RDF, ShEx and Entity Schemas

Jose Emilio Labra Gayo
WESO Research group
University of Oviedo, Spain

Overview

Why RDF, ShEx and entity schemas?

RDF data model

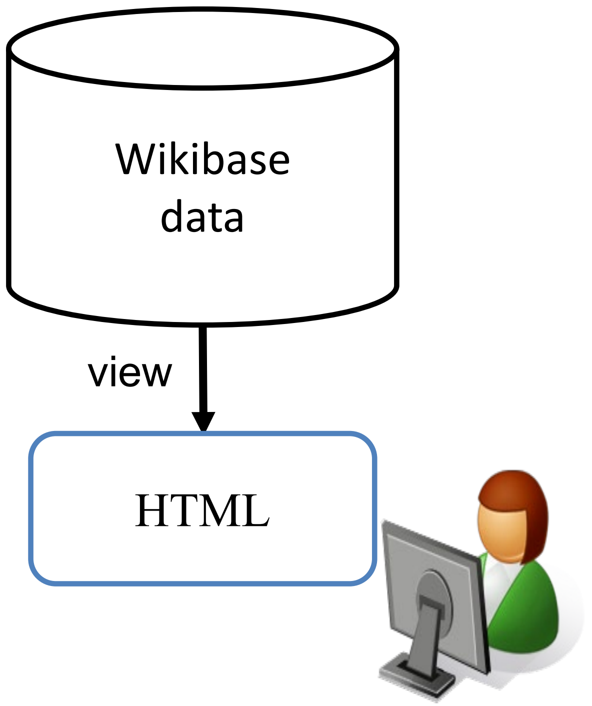
Wikibase data model and RDF

ShEx

Entity schemas

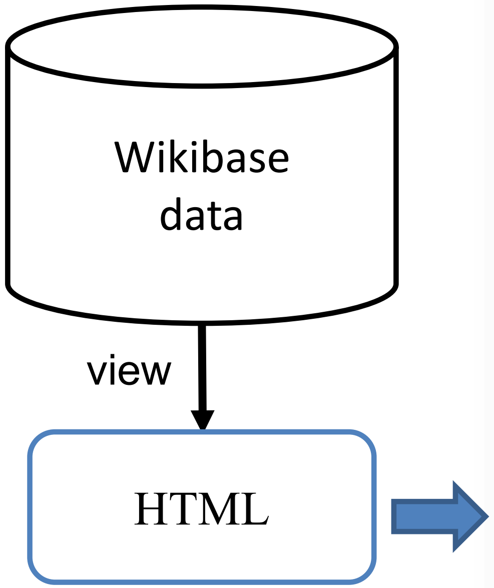
Traditional web

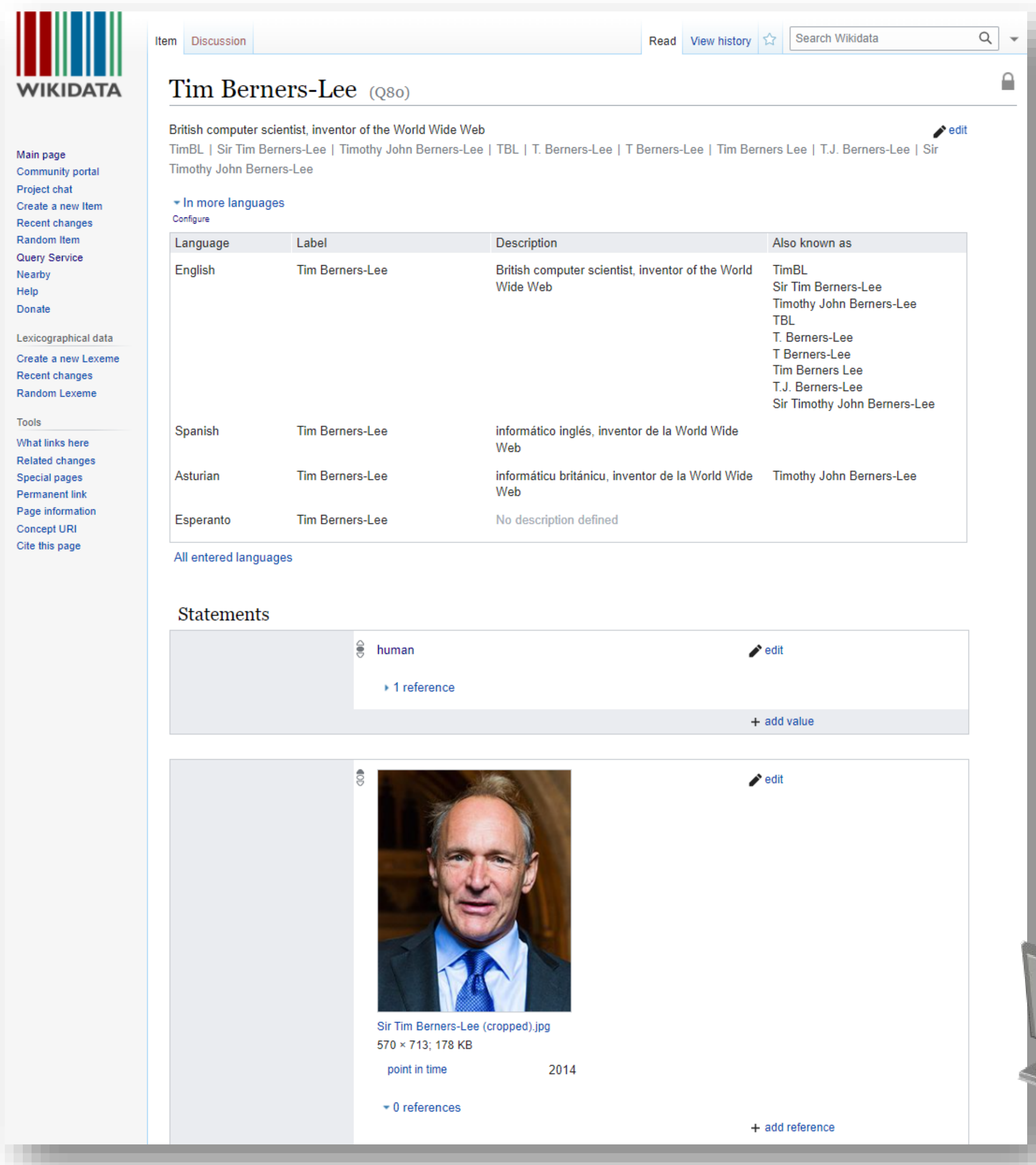
Provide HTML for humans to browse



Example (Q80)

Tim Berners-Lee



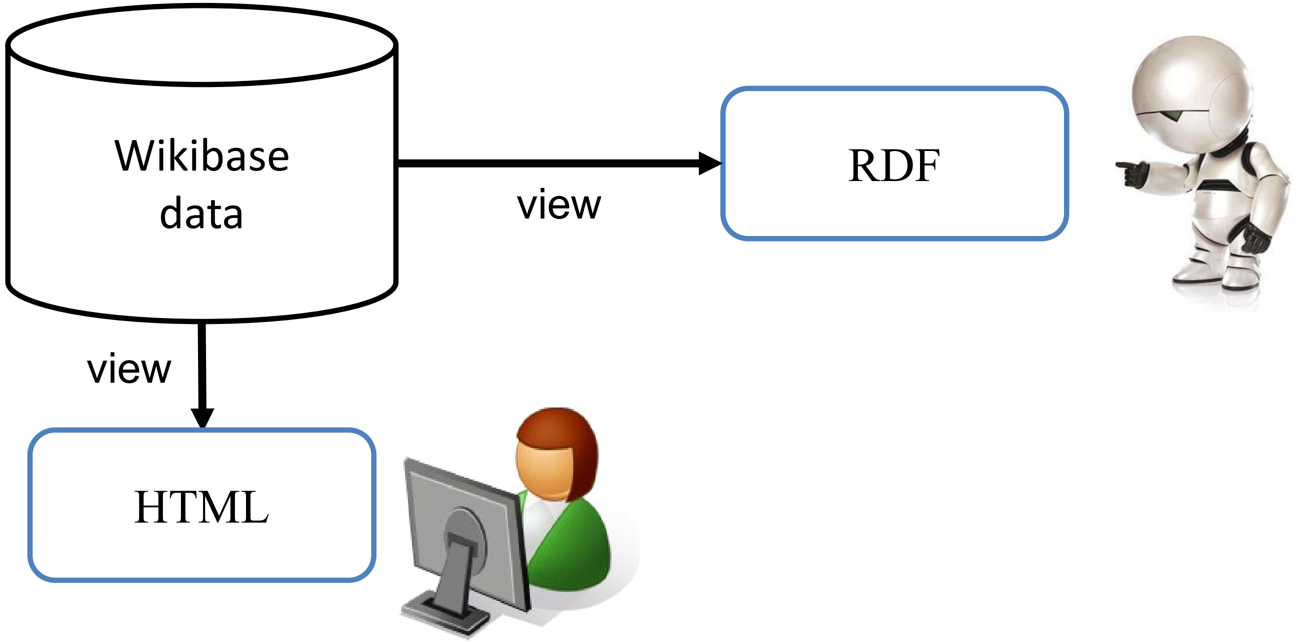


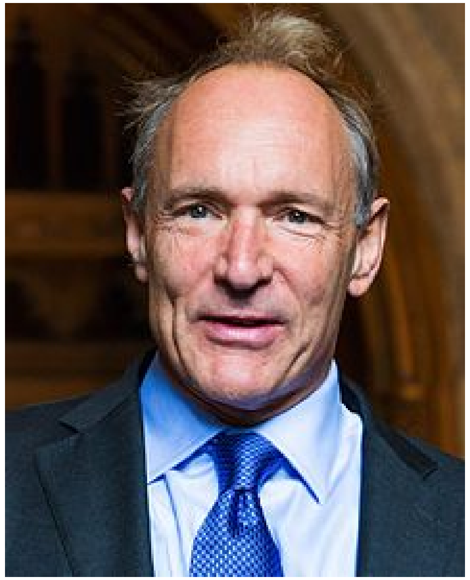
Try it: http://www.wikidata.org/entity/Q80

Semantic web

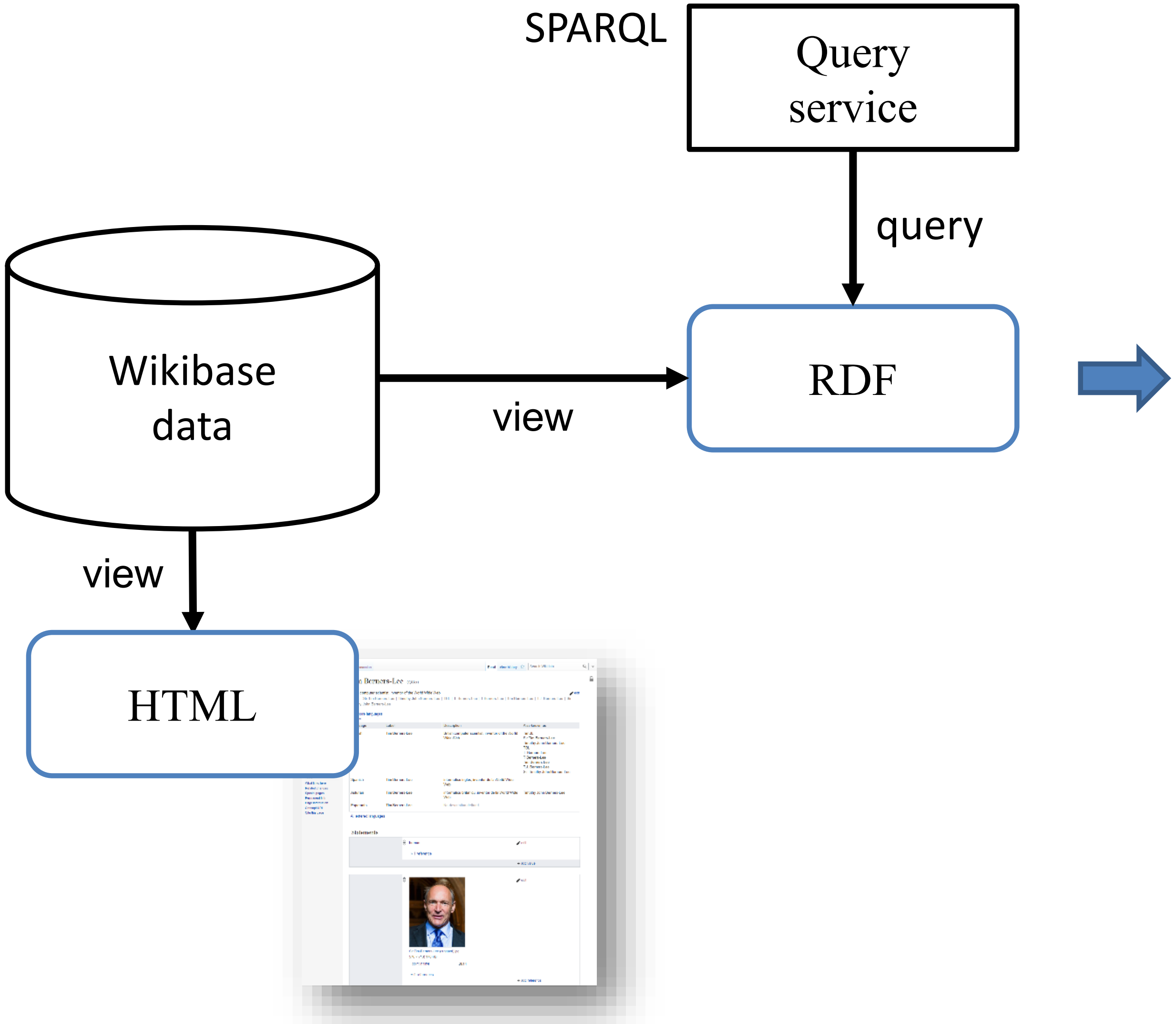
Data for humans (HTML) and for machines (RDF)

Web of data + web of documents





RDF view

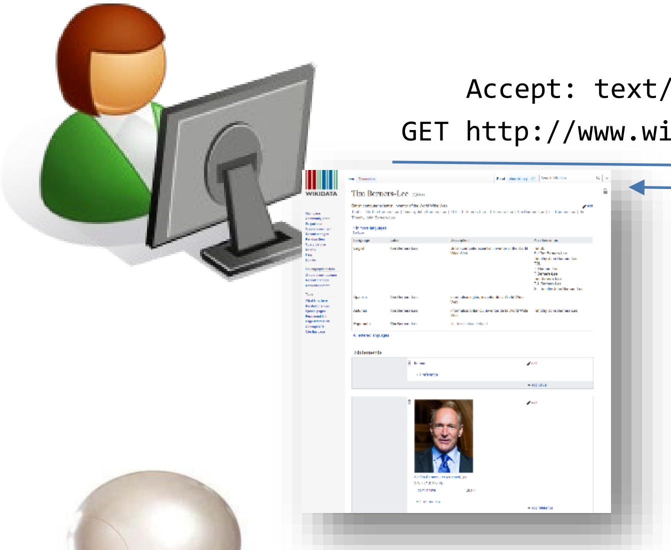



```
wd:Q80 a wikibase:Item ;
 rdfs:label "Tim Berners-Lee"@en;
 wdt:P31    wd:Q5 ;
 wdt:P18    <http://commons.wikimedia.org/...jpg> ;
 wdt:P19    wd:Q84 ;
. . .
```

Content negotiation

Each item has a concept URI (`http://www.wikidata.org/entity/Q80`)

System redirects to HTML, RDF, JSON, views depending on client



```
Accept: text/html
GET http://www.wikidata.org/entity/Q80
```

```
303 See Other
https://www.wikidata.org/wiki/Q80
```

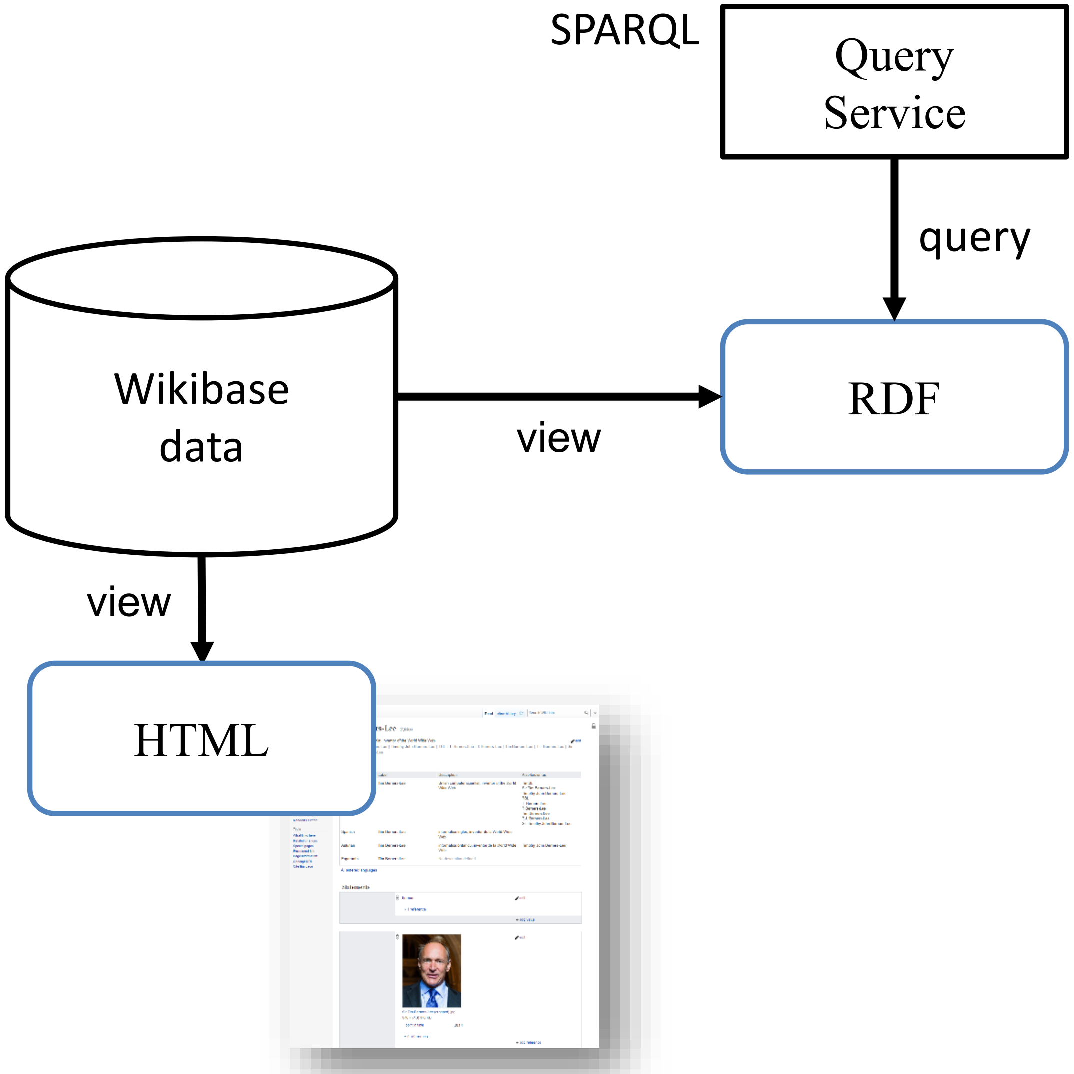
Wikibase data

```
Accept: text/turtle
GET http://www.wikidata.org/entity/Q80
```

```
wd:Q80 a     wikibase:Item ;
 rdfs:label  "Tim Berners-Lee"@en;
 wdt:P31     wd:Q5 ;
 wdt:P18     <http://commons.wikimedia.org/... jpg>;
 wdt:P19     wd:Q84 ;
```

```
303 See Other
https://www.wikidata.org/wiki/Special:EntityData/Q80.ttl
```

SPARQL query service




```
SELECT ?picure ?birthPlace WHERE {
  wd:Q80 wdt:P18 ?picture    ;
         wdt:P19 ?birthPlace .
}
```

```
wd:Q80 a wikibase:Item ;
 rdfs:label "Tim Berners-Lee"@en;
 wdt:P31    wd:Q5 ;
 wdt:P18    <http://commons.wikimedia.org/...jpg> ;
 wdt:P19    wd:Q84 ;
. . .
```

Try it: https://w.wiki/4exN

RDF and SPARQL, the good parts

Interoperability

- RDF as a communication language
- Basis for knowledge representation

Flexibility

- Graph data can be adapted to multiple models
- No need to fix a schema before adding statements

Reusability and existing tools

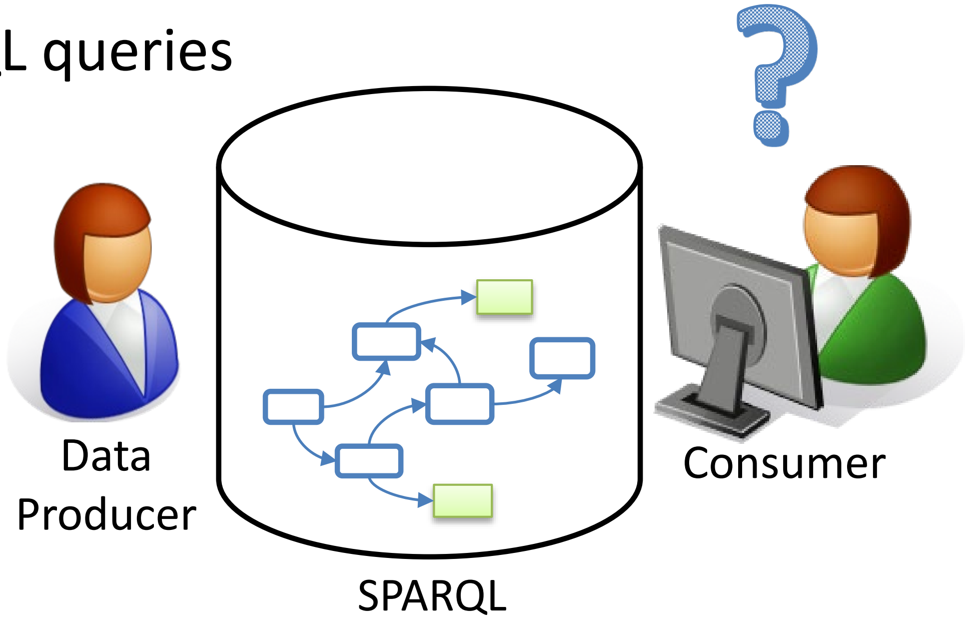
- Part of the semantic web stack: existings tools and specs
- RDF data stores and SPARQL endpoints
- Wikidata query service is really great

Wikidata is one of the best showcases of semantic web

RDF and SPARQL, the other parts...

Consuming & producing RDF

- Describing and validating RDF content
- SPARQL endpoints can be overwhelming
    - Typical documentation = set of example SPARQL queries
    - Difficult to know where to start doing queries



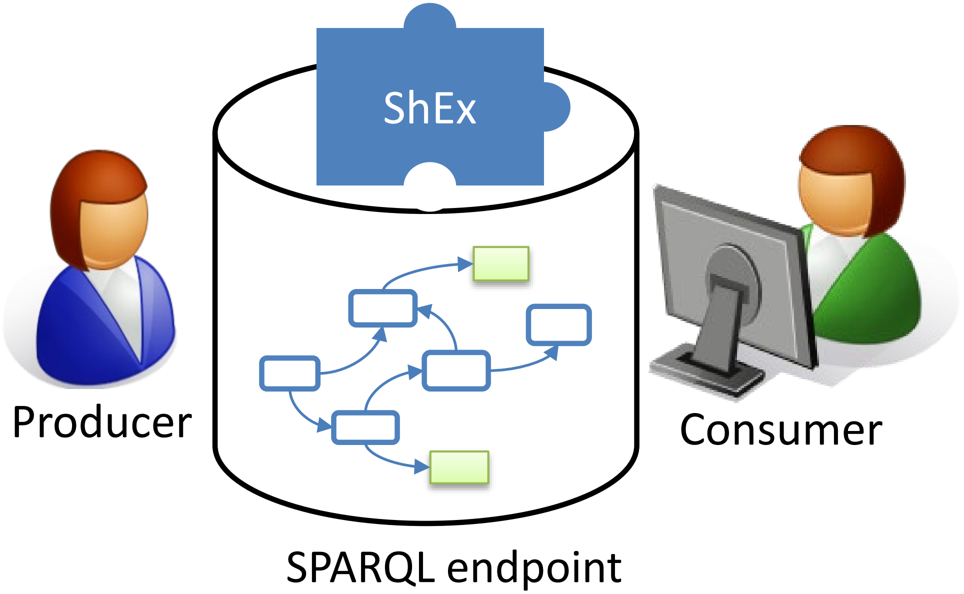
Why describe & validate RDF?

For producers

- Developers can understand the contents they are going to produce
- They can ensure they produce the expected structure
- Advertise and document that structure
- Generate interfaces

For consumers

- Understand the contents
- Verify the structure before processing it
- Query generation & optimization

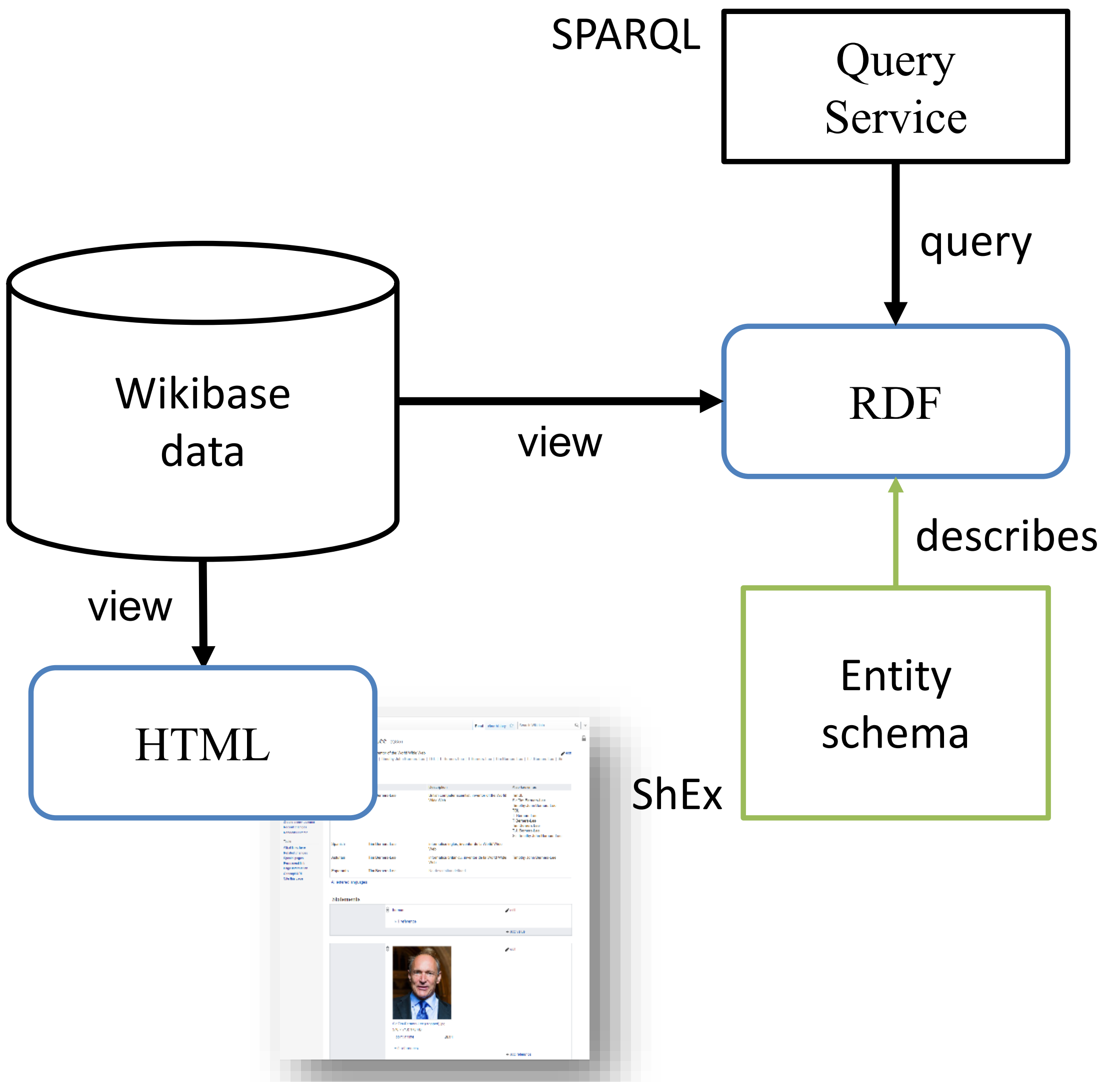


Similar technologies

Technology	Schema
Relational Databases	DDL
XML	DTD, XML Schema, RelaxNG, Schematron
Json	Json Schema
RDF	ShEx

Fill that gap

Example (Q80)




```
SELECT ?picure ?birthPlace WHERE {
  wd:Q80 wdt:P18 ?picture    ;
         wdt:P19 ?birthPlace .
}
```

```
wd:Q80 a wikibase:Item ;
 rdfs:label "Tim Berners-Lee"@en;
 wdt:P31    wd:Q5 ;
 wdt:P18    <http://commons.wikimedia.org/...jpg> ;
 wdt:P19    wd:Q84 ;
. . .
```

```
<Person> {
  wdt:P31 @<Human>   ;
  wdt:P18 @<Picture> ;
  wdt:P19 @<Place>
}
<Human>    [ wd:Q5 ]
<Place>    {}
<Picture> IRI
```

Introduction to the RDF data model

RDF

RDF = Resource description framework

Based on triples and URIs to represent properties and nodes

Short history

- Around 1997 - PICS, Dublin core, Meta Content Framework
- 1997 1st Working draft https://www.w3.org/TR/WD-rdf-syntax-971002, RDF/XML
- 1999 1st W3C Rec https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/, XML Syntax, first applications RSS, EARL
- 2004 - RDF Revised https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/, SPARQL, Turtle, Linked Data
- 2014 - RDF 1.1 https://www.w3.org/TR/rdf11-concepts/, SPARQL 1.1, JSON-LD

RDF Data Model

$$\text{subject} \xrightarrow{\text{predicate}} \text{object}$$

RDF is made from statements

Statement = a triple (subject, predicate, object)

Example:

http://www.wikidata.org/entity/Q80 —http://www.wikidata.org/prop/direct/P19→ http://www.wikidata.org/entity/Q84

subject predicate object

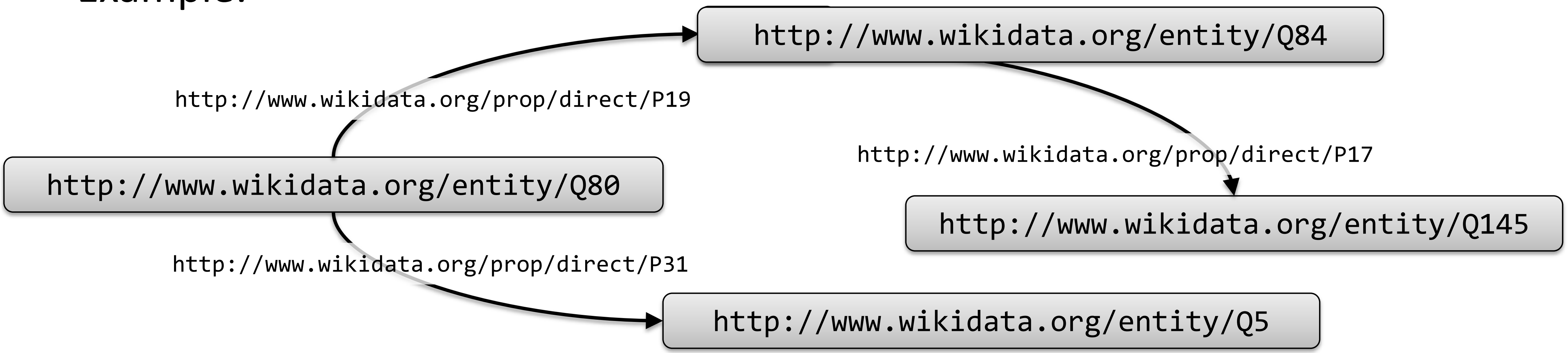
N-Triples representation

```
<http://www.wikidata.org/entity/Q80> <http://www.wikidata.org/prop/direct/P19> <http://www.wikidata.org/entity/Q84> .
```

Set of statements = RDF graph

RDF data model = directed graph

Example:



N-triples representation

```
<http://www.wikidata.org/entity/Q80> <http://www.wikidata.org/prop/direct/P19> <http://www.wikidata.org/entity/Q84> .
<http://www.wikidata.org/entity/Q84> <http://www.wikidata.org/prop/direct/P17> <http://www.wikidata.org/entity/Q145> .
<http://www.wikidata.org/entity/Q80> <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/Q5> .
```

subject predicate object

Turtle notation

Human readable notation that simplifies N-Triples

Allows namespace declarations and some abreviations

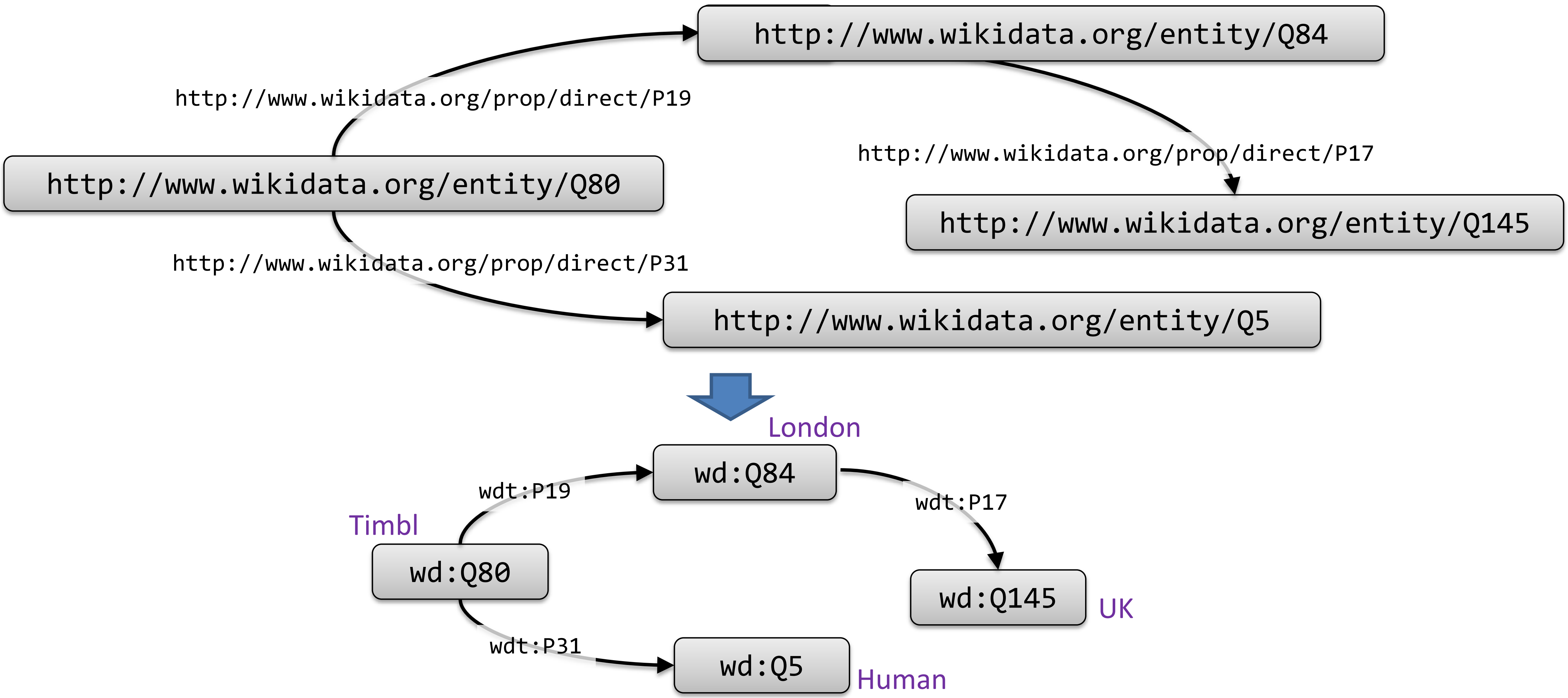
N-Triples

```
<http://www.wikidata.org/entity/Q84> <http://www.wikidata.org/prop/direct/P17> <http://www.wikidata.org/entity/Q145> .
<http://www.wikidata.org/entity/Q80> <http://www.wikidata.org/prop/direct/P19> <http://www.wikidata.org/entity/Q84> .
<http://www.wikidata.org/entity/Q80> <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/Q5> .
```

Turtle

```
prefix wd: <http://www.wikidata.org/entity/>
prefix wdt: <http://www.wikidata.org/prop/direct/>

wd:Q80 wdt:P19 wd:Q84 ;
       wdt:P31 wd:Q5 .
wd:Q84 wdt:P17 wd:Q145 .
```

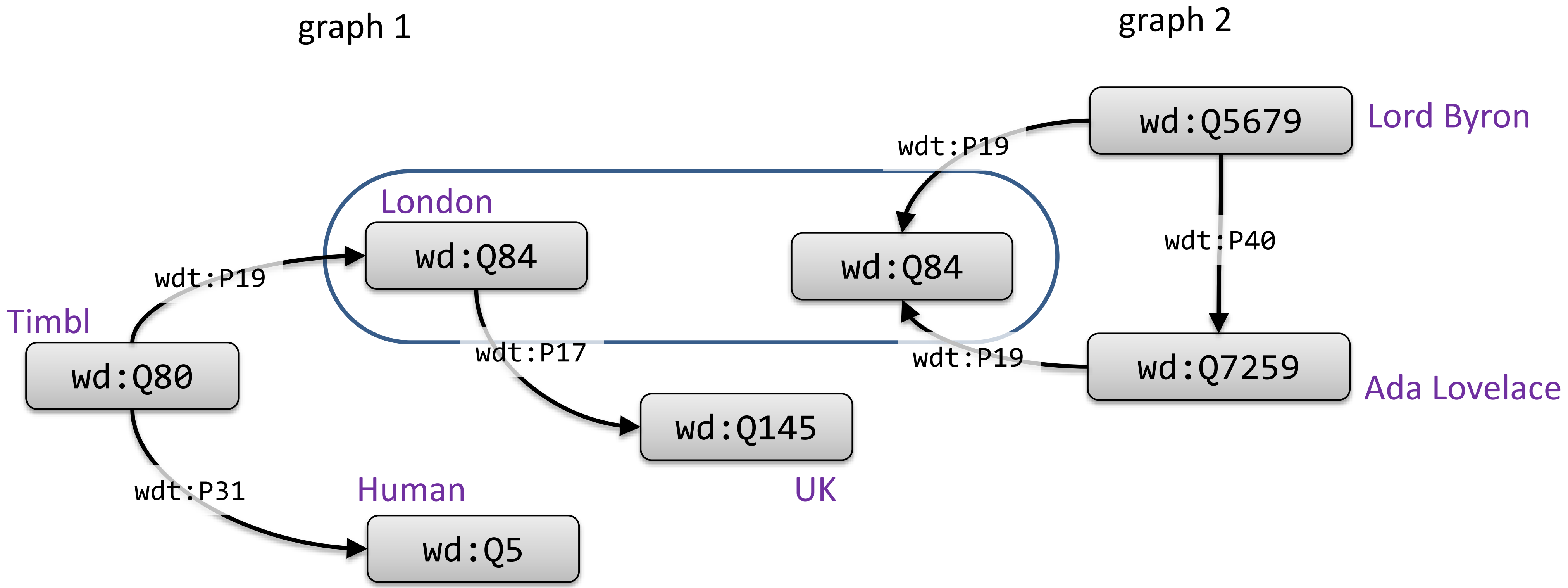
Namespaces simplification



RDF is compositional

RDF graphs can be merged to obtain a bigger RDF graphs

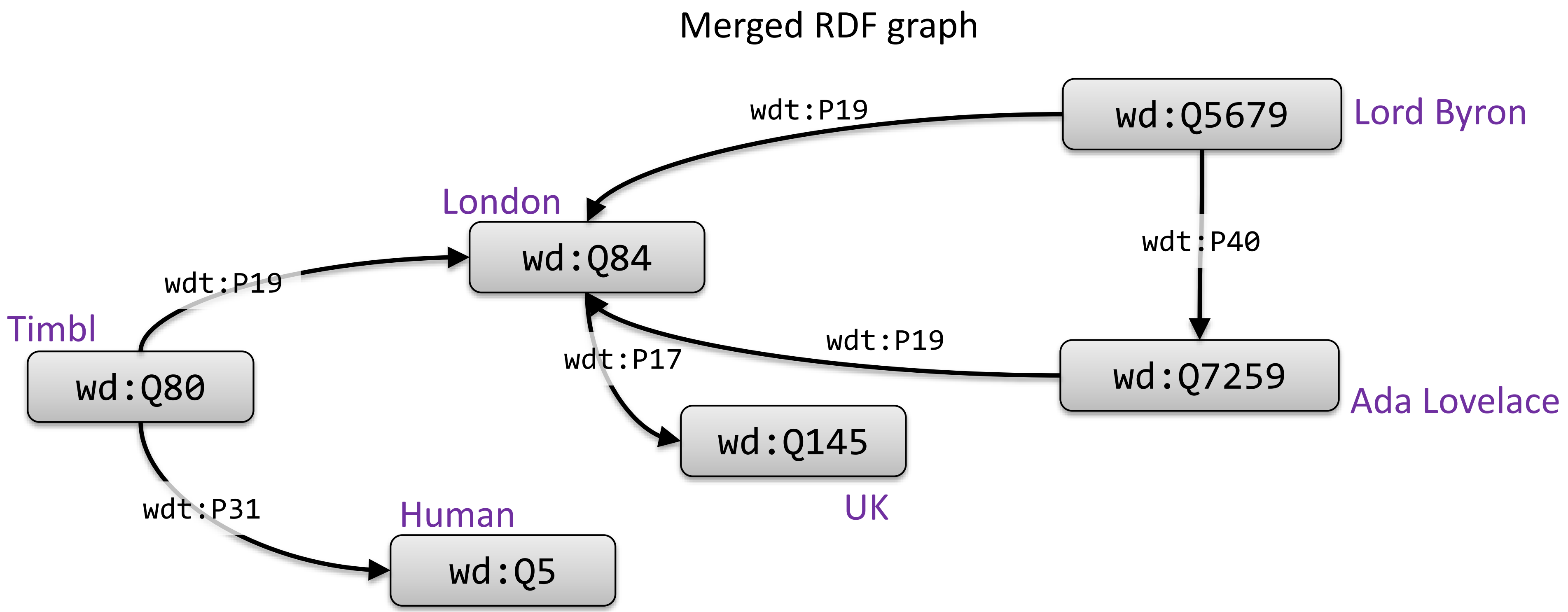
Automatic data integration



RDF is compositional

RDF graphs can be merged to obtain a bigger RDF graphs

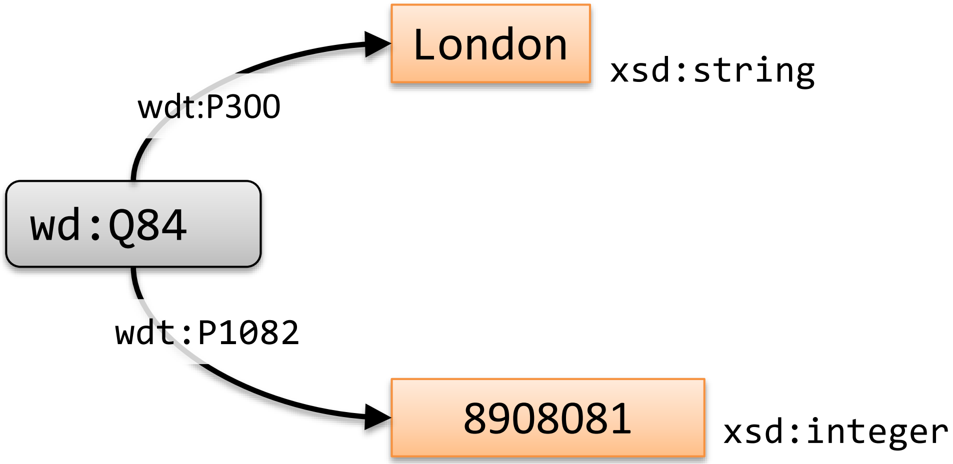
Automatic data integration



RDF Literals

Objects can also be literals

- Literals contain a lexical form and a datatype
- Typical datatypes = XML Schema primitive datatypes
  - If not specified, a literal has datatype `xsd:string`

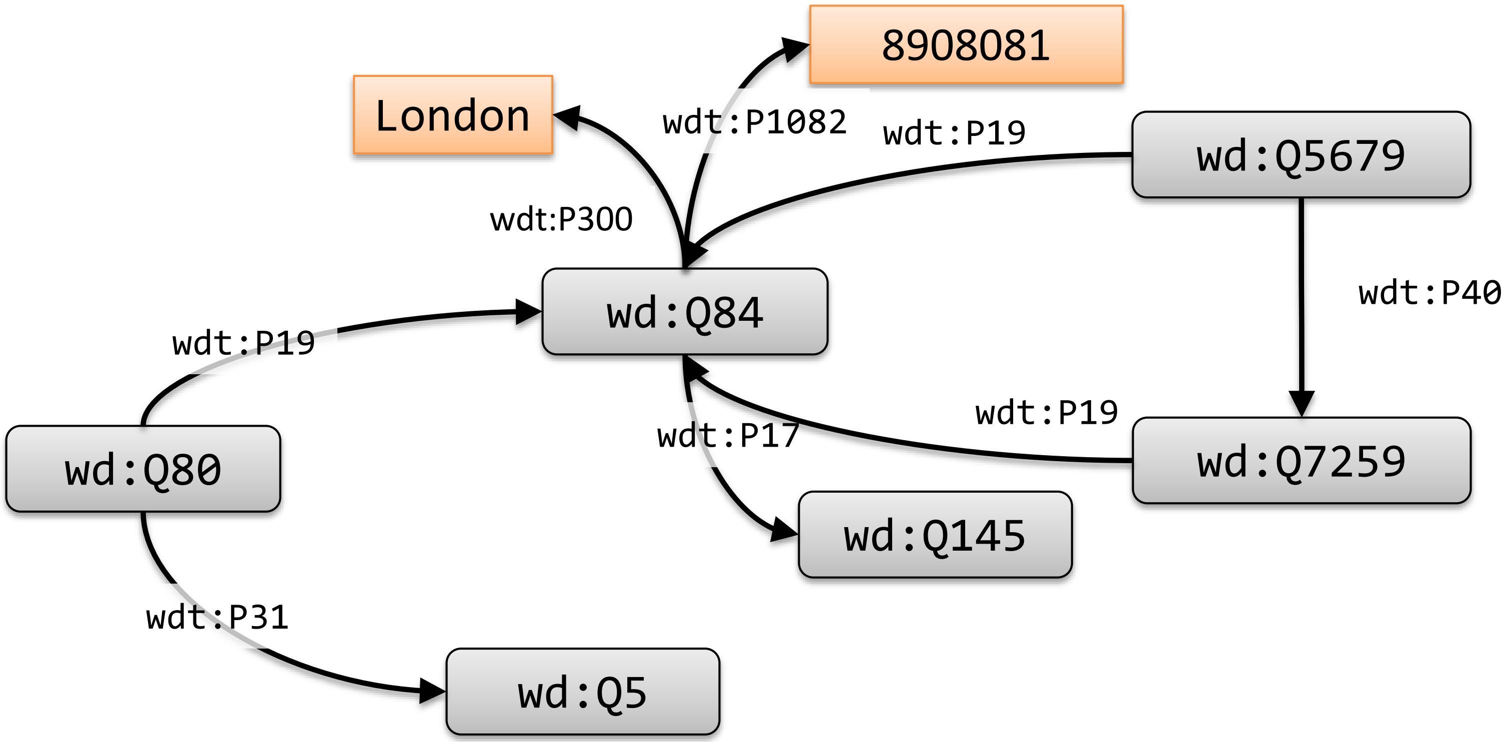


Turtle notation

```
wd:Q84 wd:P300   "GB-LND" ;
wd:Q84 wdt:P1082 "8908081"^^xsd:integer .
```

Remember...RDF is compositional

Merging previous data



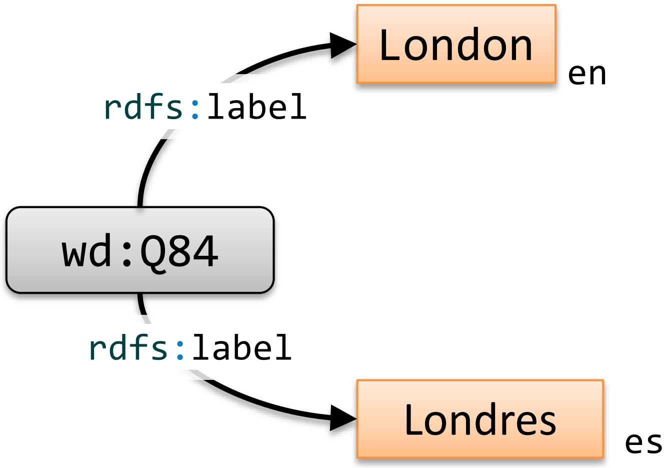

```
prefix wd: <http://www.wikidata.org/entity/>
prefix wdt: <http://www.wikidata.org/prop/direct/>

wd:Q80    wdt:P19 wd:Q84 ;
          wdt:P31 wd:Q5 .
wd:Q5679 wdt:P40 wd:Q7259 ;
          wdt:P19 wd:Q84 .
wd:Q7259 wdt:P19 wd:Q84 .
wd:Q84    wdt:P300 "GB-LND" ;
          wdt:P1082 8908081 ;
          wdt:P17 wd:Q145 .
```

https://rdfshape.weso.es/link/16417186264

Language tagged strings

String literals can be qualified by a language tag

They have datatype `rdfs:langString`

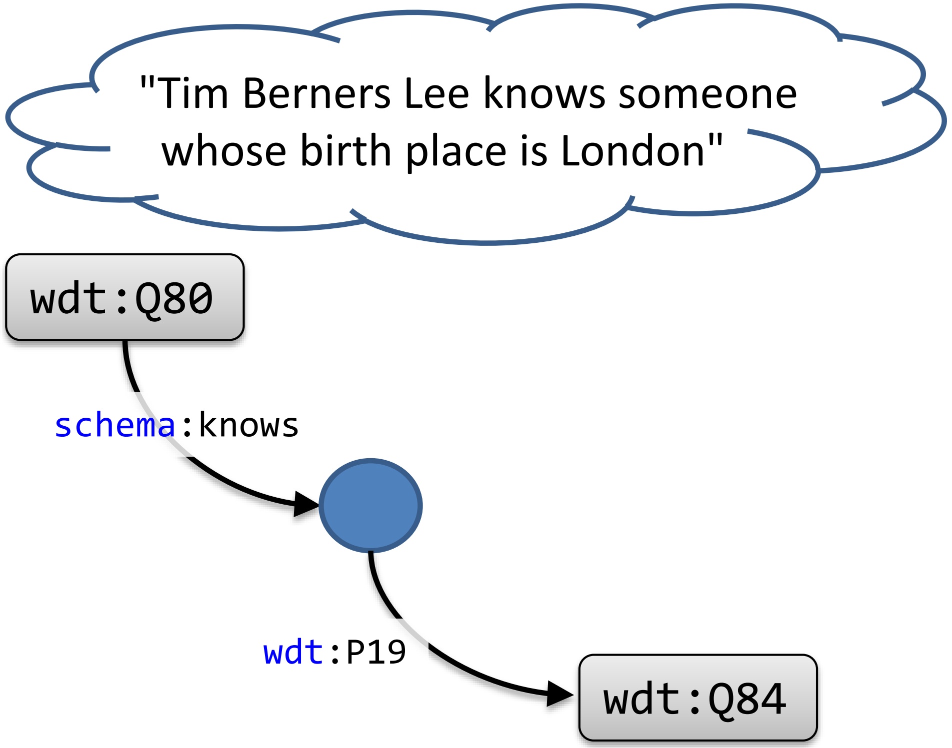


Turtle notation

```
wd:Q84 rdfs:label "London"@en  ;
       rdfs:label "Londres"@es .
```

Blank nodes

Subjects and objects can also be Blank nodes



Turtle notation with local identifier

```
wd:Q80 schema:knows _:x .
_:x     wdt:P19      wd:Q84 .
```

Turtle notation with square brackets

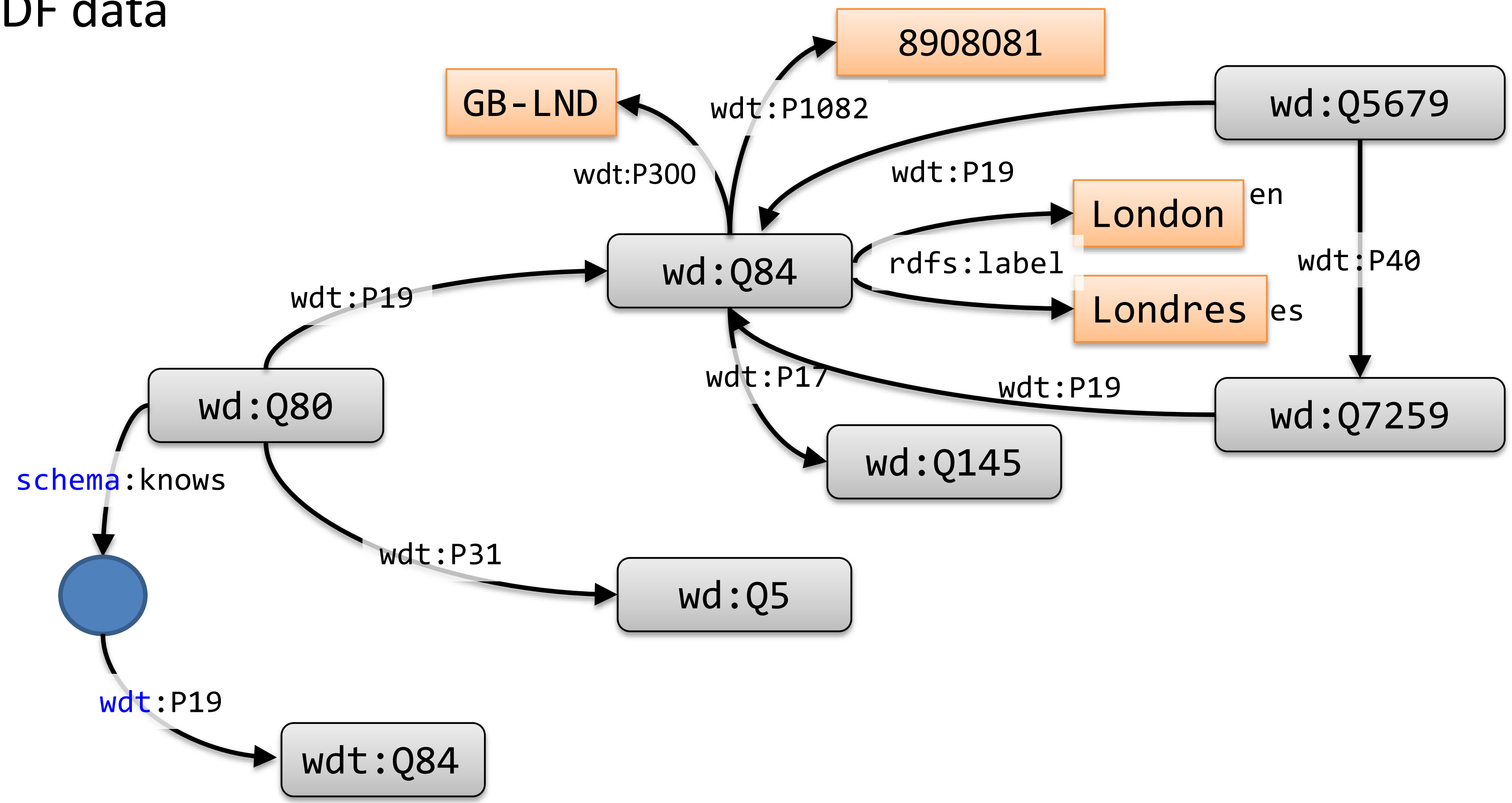
```
wd:Q80 schema:knows [ wdt:P19 wd:Q84 ] .
```

Mathematical meaning:

$\exists x(\text{schema:knows}(\text{wd:Q80}, x) \wedge \text{wdt:P19}(x, \text{wdt:Q84}))$

RDF data model

Example of RDF data



3 types of nodes

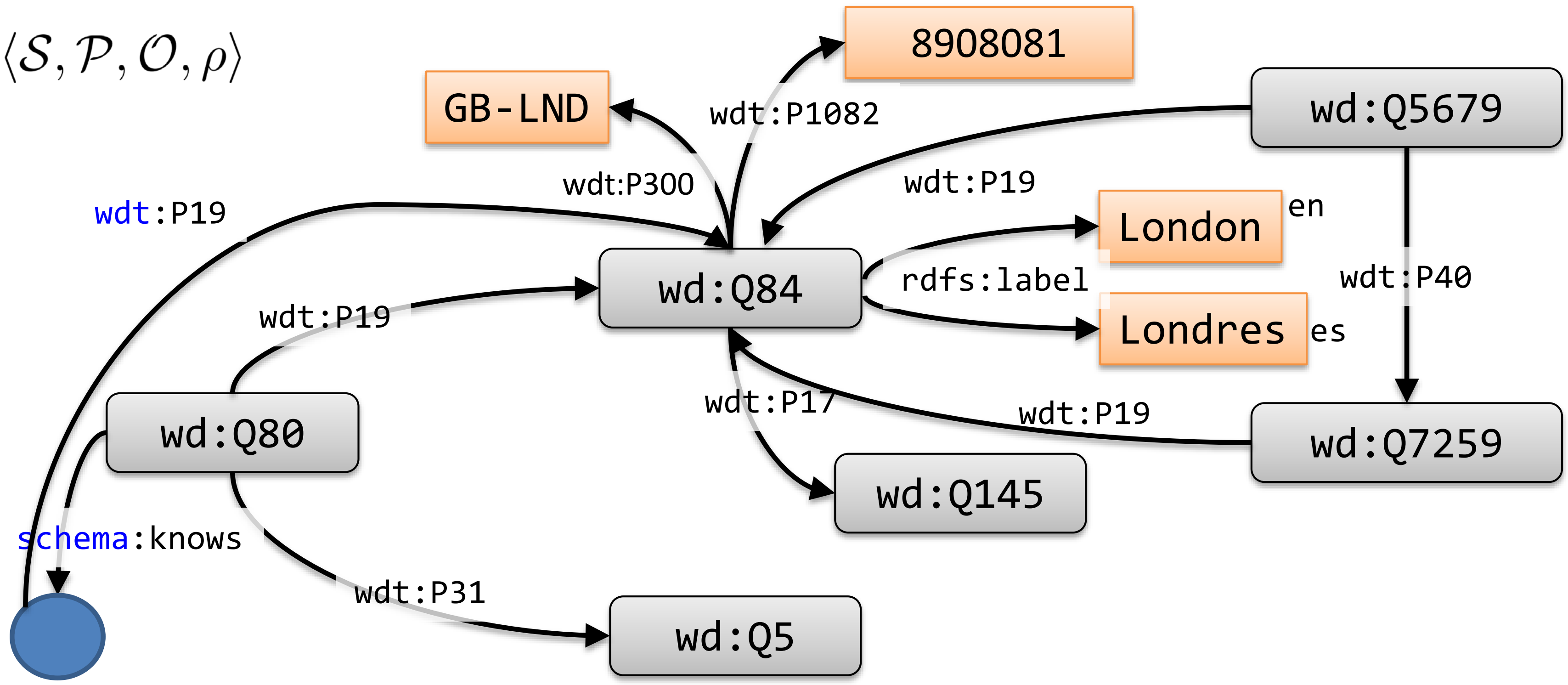
- IRIs
- Blank nodes
- Literals

Subjects: URIs or Blank nodes
Objects: URIs, Blank nodes or literals
Predicates always URIs

Formal definition of RDF data model

Given a set of IRIs $\mathcal{I}$,
a set of blank nodes $\mathcal{B}$
and a set of literals Lit
an *RDF graph* is a tuple $\mathcal{G} = \langle \mathcal{S}, \mathcal{P}, \mathcal{O}, \rho \rangle$
where

$\mathcal{S} = \mathcal{I} \cup \mathcal{B}$,
$\mathcal{P} = \mathcal{I}$,
$\mathcal{O} = \mathcal{I} \cup \mathcal{B} \cup Lit$
$\rho \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{O}$



3 types of nodes
IRIs
Blank nodes
Literals

Subjects: URIs or Blank nodes
Objects: URIs, Blank nodes or literals
Predicates always URIs

...and that's all about the RDF data model

The RDF Data model is very simple

Simple
is
better

Wikibase data model and RDF

2 different data models

Wikibase

- Intial goal: support Wikipedia
- Collaborative model based on MediaWiki
- Combines 2 models
  - Document centric (MediaWiki
  - Graph model (statements)
    - Statements can have qualifiers & references

RDF

- Initial goal: Knowledge representation
- Basis for Semantic Web
- Resources = URIs
- Graph based model
- Graph = Set of triples

Wikibase data model

Described in https://www.mediawiki.org/wiki/Wikibase/DataModel

Entities

Items (Q..), Properties (P..), Lexemes (L..)

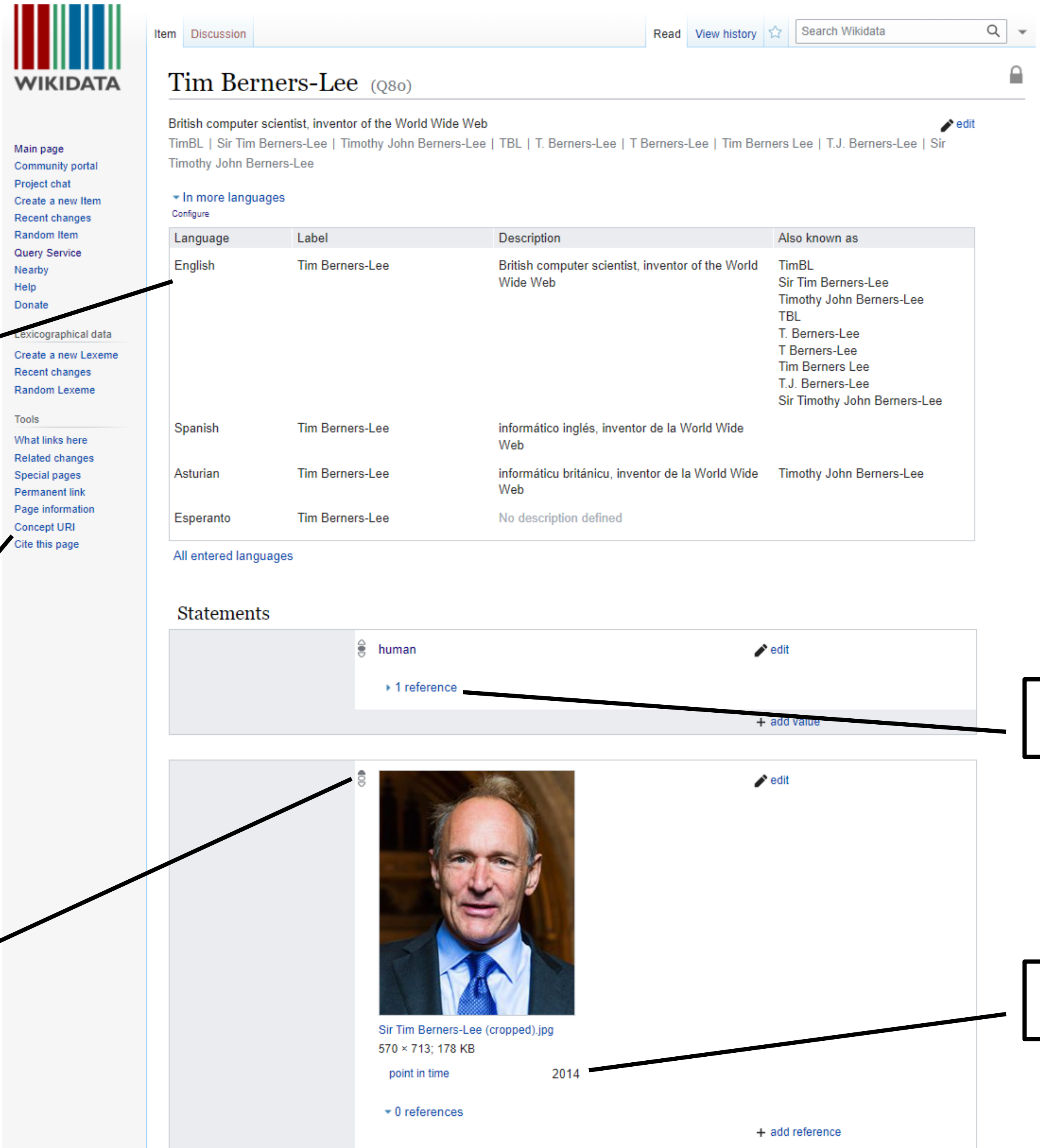
Each entity has:

- Labels, descriptions, aliases
- List of statements (Property-values)
- Each statement can have qualifiers and references

Built-in data values

Examples: strings, numbers, dates, time-values, geo-coordinates, ...

Example



Multilingual labels, descriptions and aliases

Concept URI

http://www.wikidata.org/entity/Q80

Hidden reference

Rank

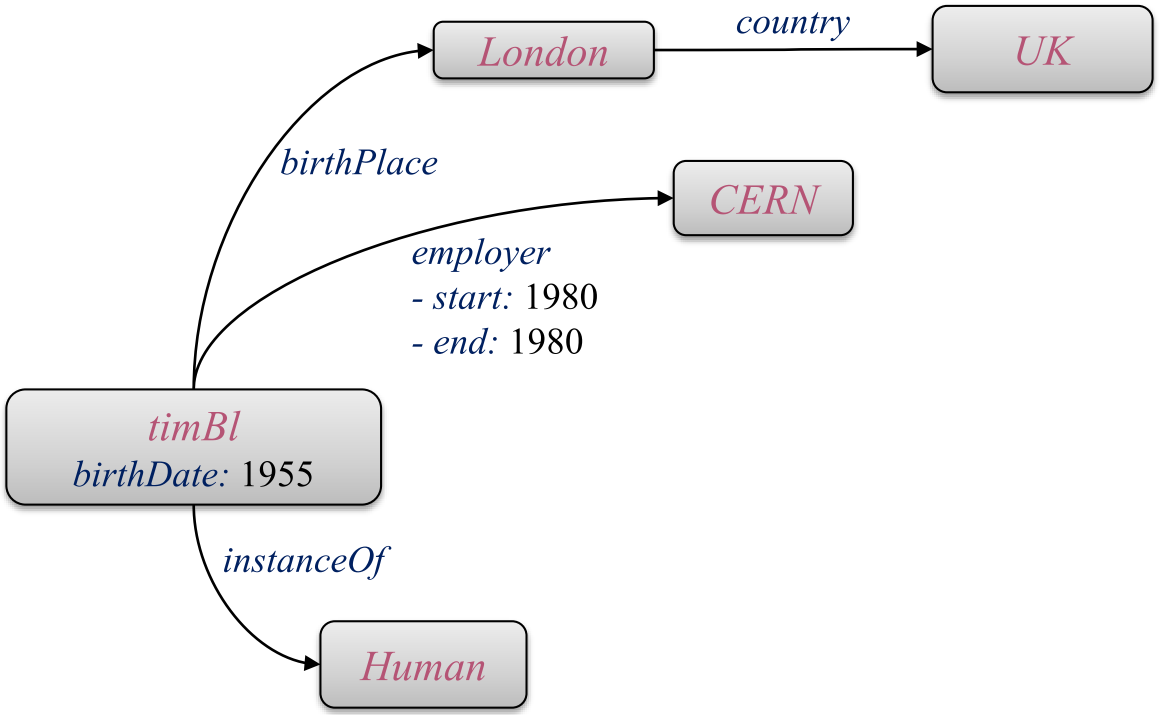
Qualifier

Wikibase graphs

Wikibase graph model similar to Property Graphs

Nodes can have a list of property-values

Statements can also have a list of property-values



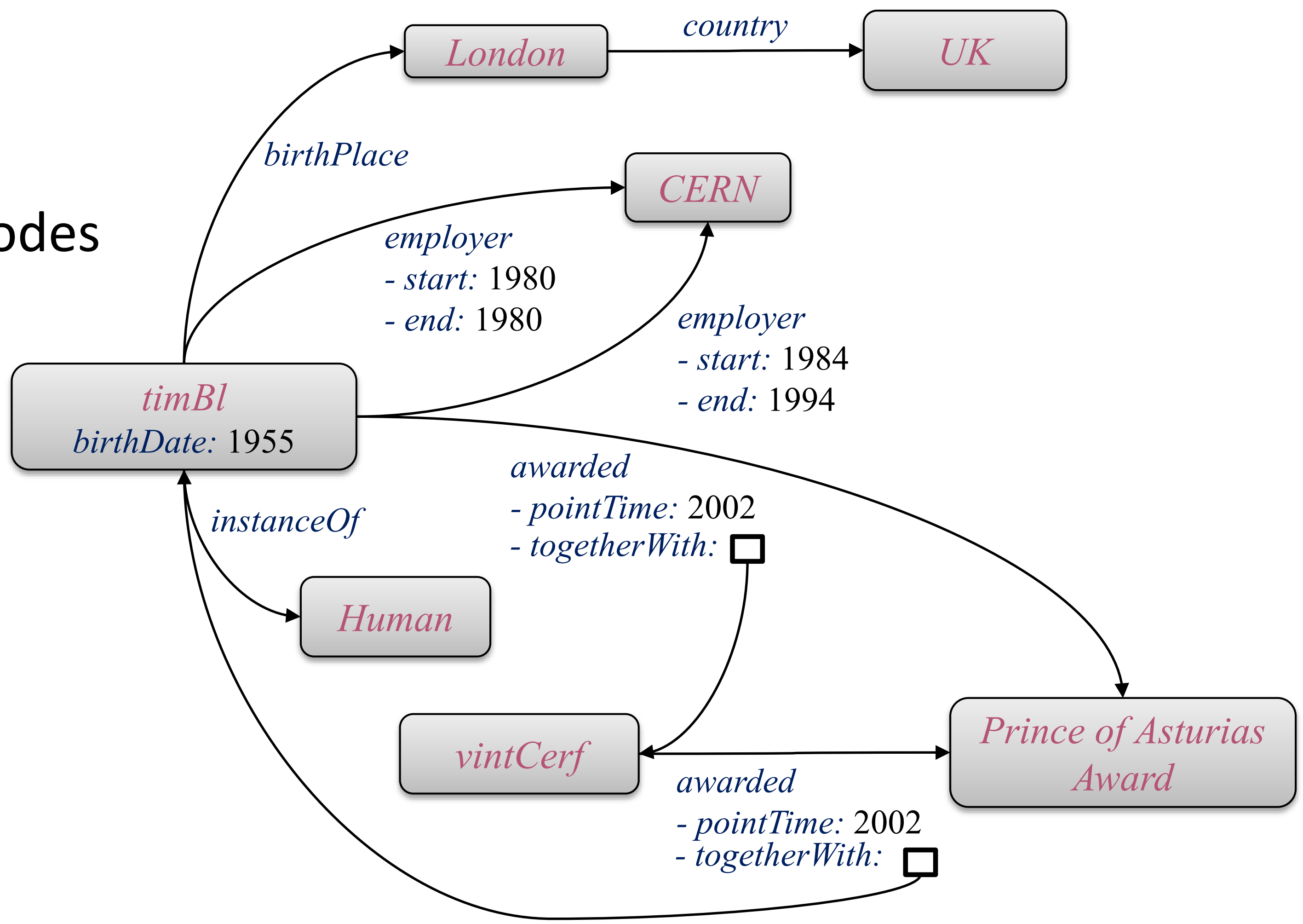
Wikibase graphs

Wikibase graphs = Multigraphs

- It is possible to have more than one statement between the same 2 nodes

Generalize property graphs

- The values can also be nodes



Wikibase ⇝ RDF: prefixes

New namespaces created for Wikidata: wd, wdt, p, ps, pr, psv ...

Reuse popular namespaces: rdf, rdfs, dct, owl, prov, skos, ...

Some popular prefix declarations

wd	http://www.wikidata.org/entity/
wdt	http://www.wikidata.org/prop/direct/
p	http://www.wikidata.org/prop/
ps	http://www.wikidata.org/prop/statement/
pq	http://www.wikidata.org/prop/qualifier/
pr	http://www.wikidata.org/prop/reference/
psv, pqv, prv	http://www.wikidata.org/prop/{statement|qualifier|reference}/value/
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
...	...

Wikidata query service assumes those prefix declarations

Wikibase ⇝ RDF: entities

Entity nodes (ex. `wd:Q80`) have type `wikibase:Item`

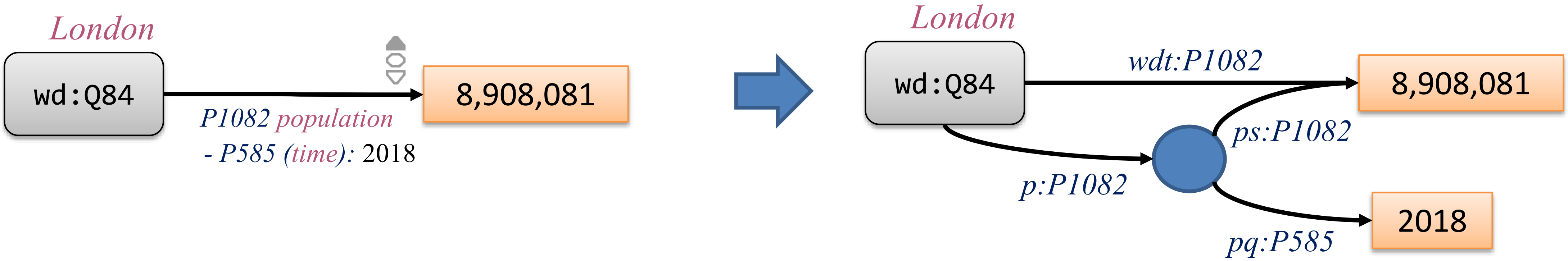
- Labels declared using `rdfs:label`, `skos:prefLabel`, `schema:name`
- Descriptions declared using `schema:description`
- Aliases declared using `skos:altLabel`

```
wd:Q80  rdf:type            wikibase:Item ;
        rdfs:label          "Tim Berners-Lee"@en , "Tim Berners-Lee"@es ;
        schema:description  "British computer scientist, inventor of the World Wide Web"@en ,
                            "informático inglés, inventor de la World Wide Web"@es ;
        schema:name         "Tim Berners-Lee"@en , "Tim Berners-Lee"@es ;
        skos:altLabel       "T Berners-Lee"@en , "Sir Timothy John Berners-Lee"@en ;
        skos:prefLabel      "Tim Berners-Lee"@en , "Tim Berners-Lee"@es
        ...
```

Wikibase ⇝ RDF: statements

Statements have 2 possibilities

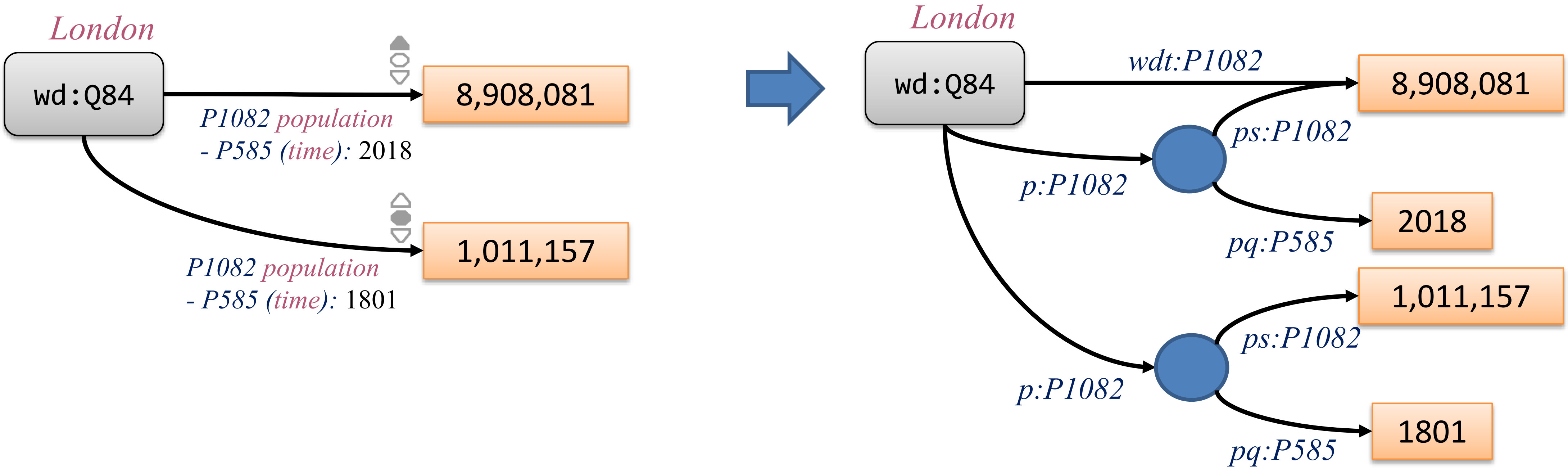
- Truthy statements: best non-deprecated rank for a property
- Full statements: contain all data about a statement



Wikibase ⇝ RDF: statements

Statements have 2 possibilities

- Truthy statements: best non-deprecated rank for a property
- Full statements: contain all data about a statement

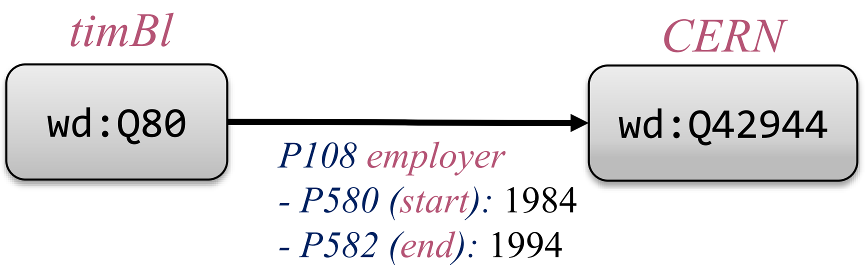


Wikibase ⇝ RDF: qualifiers

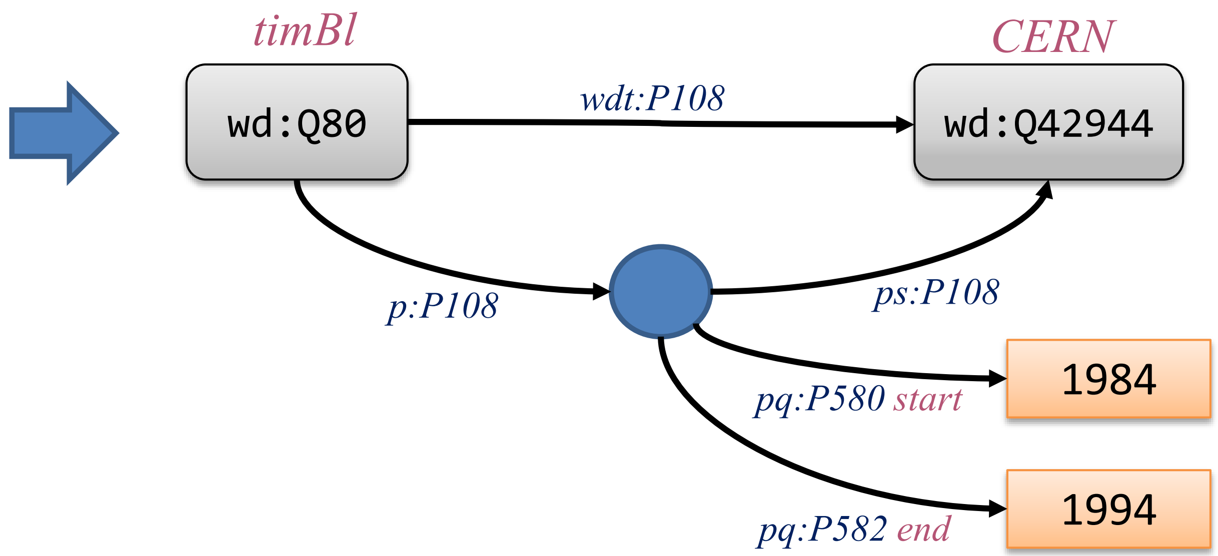
Qualifiers = statements about statements

RDF reification can be used for that

- Different RDF reification alternatives
- Wikibase approach to reification uses auxiliary nodes



Wikibase

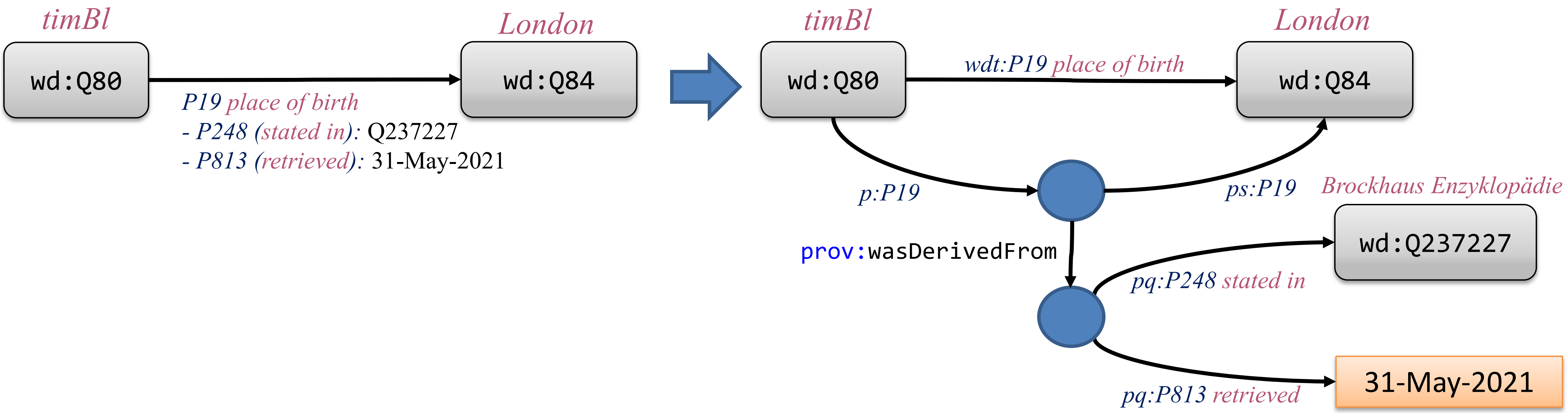


RDF

Wikibase ⇝ RDF: references

References = similar to qualifiers

Adds a new node that represents the provenance information

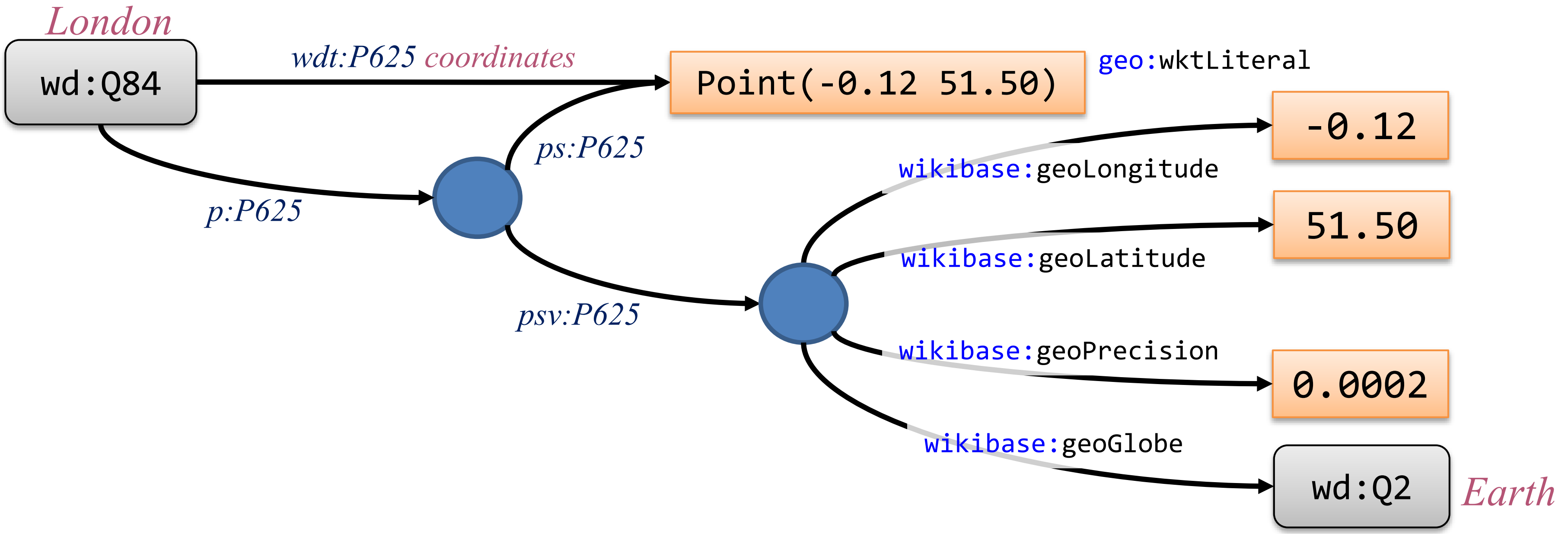


Wikibase ⇝ RDF: values

Values represented as simple and full values

- Simple values = literals or URIs
- Full values include more information

Example London's coordinates:



Wikibase ⇝ RDF: other features

Normalized values

Special values: Some values, NoValues

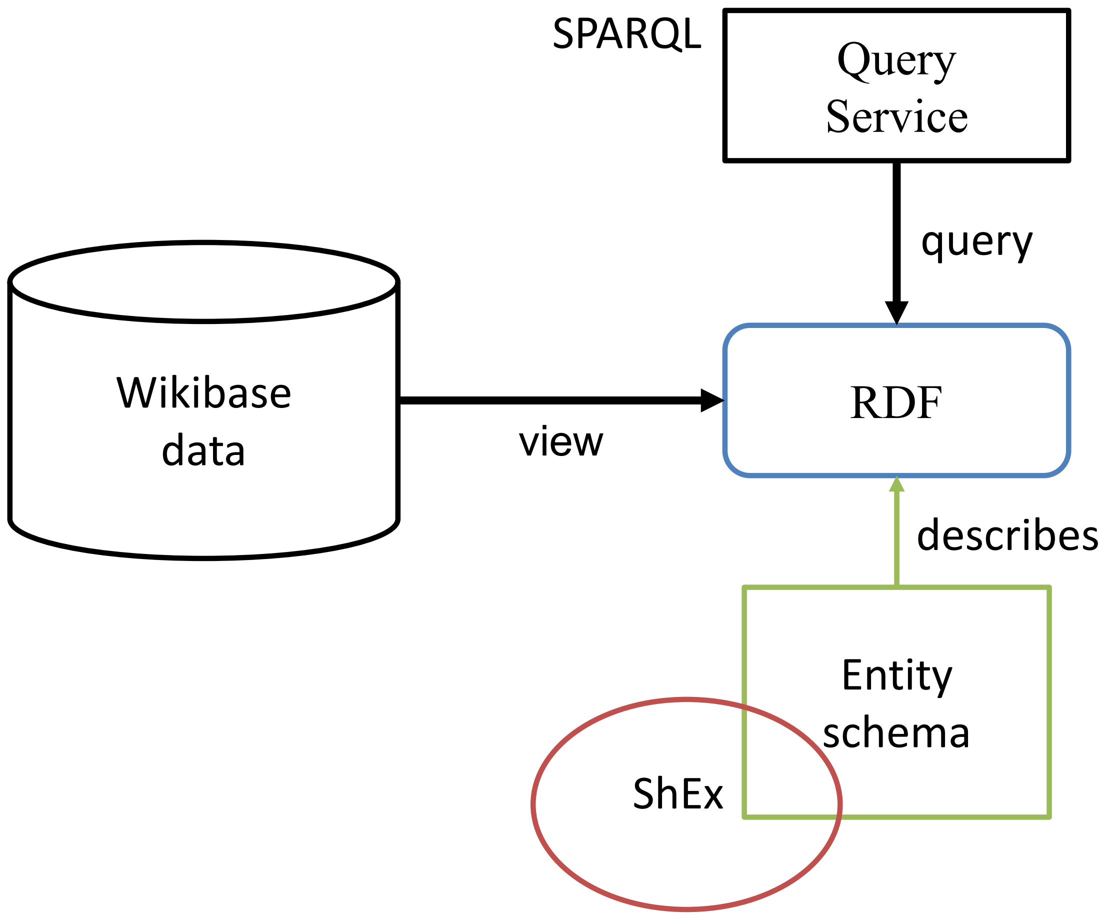
Sitelinks

Redirects

More information: https://www.mediawiki.org/wiki/Wikibase/Indexing/RDF_Dump_Format

Wikibase, RDF and ShEx

Overview



Introduction to Shape Expressions

ShEx

ShEx (Shape Expressions Language)

Goal: RDF validation & description

Design objectives: High level, concise, human-readable, machine processable language

Syntax inspired by SPARQL, Turtle

Semantics inspired by RelaxNG

Official info: http://shex.io

ShEx as a language

Language based approach

- ShEx = domain specific language for RDF validation
- Specification: http://shex.io/shex-semantics/
- Primer: http://shex.io/shex-primer
- Different serializations:
  - ShExC (Compact syntax)
  - JSON-LD (ShExJ)
  - RDF obtained from JSON-LD (ShExR)

Short history of ShEx

2013 - RDF Validation Workshop

- Conclusions: "*SPARQL queries cannot easily be inspected and understood…*"
  - Need of a higher level, concise language
  - Agreement on the term "Shape"

2014 First proposal of Shape Expressions (ShEx 1.0)

2014 - Data Shapes Working Group chartered

- Mutual influence between SHACL & ShEx

2017 - ShEx Community Group - ShEx 2.0

2018 - ShEx 2.1

ShEx implementations and demos

Implementations:

- shex.js: Javascript
- Apache Jena ShEx: Java
- shex-s: Scala (Jena/RDF4j)
- PyShEx: Python
- shex-java: Java
- Ruby-ShEx: Ruby
- ShEx-ex: Elixir

Online demos & playgrounds

- ShEx-simple
- RDFShape
- ShEx-Java
- ShExValidata
- Wikishape

Simple example

Prefix declarations as in Turtle/SPARQL

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

<User> IRI {
 schema:name   xsd:string   ;
 schema:knows @<User>     *
}
```

Nodes conforming to <User> shape must:

- Be IRIs
- Have exactly one schema:name with a value of type xsd:string
- Have zero or more schema:knows whose values conform to <User>

RDF Validation using ShEx

Schema

```
<User> IRI {
 schema:name   xsd:string   ;
 schema:knows @<User>      *
}
```

Shape map

```
:alice@<User> ✓
:bob   @<User> ✓
:carol@<User> ✗
:dave  @<User> ✗
:emily@<User> ✗
:frank@<User> ✓
:grace@<User> ✗
```

Try it (RDFShape): https://goo.gl/97bYdv
Try it (ShExDemo):https://goo.gl/Y8hBsW

Data

```
:alice schema:name  "Alice" ;
       schema:knows :alice  .

:bob   schema:knows :alice ;
       schema:name  "Robert".

:carol schema:name   "Carol", "Carole" .

:dave  schema:name   234 .

:emily foaf:name     "Emily" .

:frank schema:name "Frank" ;
       schema:email <mailto:frank@example.org> ;
       schema:knows :alice, :bob .

:grace schema:name "Grace" ;
       schema:knows :alice, _:1 .

_:1 schema:name  "Unknown" .
```

Validation process

Input: RDF data, ShEx schema, Shape map

Output: Result shape map

ShEx Schema

```
:User {
 schema:name  xsd:string ;
 schema:knows @:User *
}
```

Shape map

```
:alice@:User, :bob@:User, :carol@:User
```

RDF data

```
:alice schema:name  "Alice" ;
       schema:knows :alice  .

:bob   schema:knows :alice ;
       schema:name  "Robert".

:carol schema:name   "Carol", "Carole" .
```

ShEx Validator

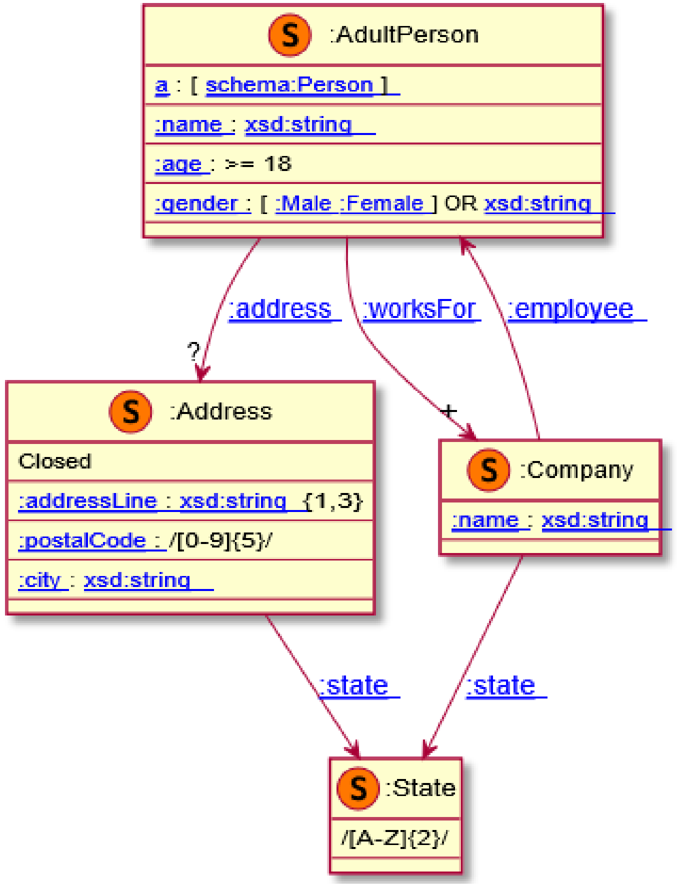
Result shape map

```
:alice@:User,
:bob@:User,
:carol@!:User
```

Example with more ShEx features

```
:AdultPerson EXTRA rdf:type {
 rdf:type    [ schema:Person ]              ;
 :name       xsd:string                     ;
 :age        MinInclusive 18                ;
 :gender     [:Male :Female] OR xsd:string ;
 :address    @:Address ?                    ;
 :worksFor   @:Company +
}
:Address CLOSED {
 :addressLine xsd:string {1,3}
 :postalCode  /[0-9]{5}/
 :state       @:State
 :city        xsd:string
}
:Company {
 :name     xsd:string
 :state    @:State
 :employee @:AdultPerson *
}
:State   /[A-Z]{2}/
```

```
:alice rdf:type :Student, schema:Person ;
 :name       "Alice" ;
 :age        20 ;
 :gender     :Male ;
 :address    [
  :addressLine  "Bancroft Way" ;
  :city         "Berkeley" ;
  :postalCode   "55123" ;
  :state        "CA"
 ] ;
 :worksFor  [
  :name        "Company" ;
  :state       "CA"      ;
  :employee    :alice
] .
```



Try it: https://tinyurl.com/yd5hp9z4

ShExC - Compact syntax

BNF Grammar: http://shex.io/shex-semantics/#shexc

Shares terms with Turtle and SPARQL

- Prefix declarations
- Comments starting by #
- a keyword = rdf:type
- Keywords aren't case sensitive (MinInclusive = MININCLUSIVE)

Shape Labels can be URIs or BlankNodes

ShEx-Json

JSON-LD serialization for Shape Expressions and validation results

```
prefix schema: <http://schema.org/>
prefix xsd:    <http://www.w3.org/2001/XMLSchema#>
base   <http://example.com/>

<User> {
  schema:name  xsd:string ;
}
```

equivalent

```
{ "type" : "Schema",
  "@context" : "http://www.w3.org/ns/shex.jsonld",
  "shapes" :[{"type" : "Shape",
    "id" : "http://a.example/UserShape",
    "expression" : {
    "type" : "TripleConstraint",
    "predicate" : "http://schema.org/name",
    "valueExpr" : { "type" : "NodeConstraint",
      "datatype" : "http://www.w3.org/2001/XMLSchema#string"
     }
   }
  }]
}
```

Some definitions

Schema = set of Shape Expressions

Shape Expression = labeled pattern

```
<label> {
  ...pattern...
}
```

Shape Label

Shape Expression

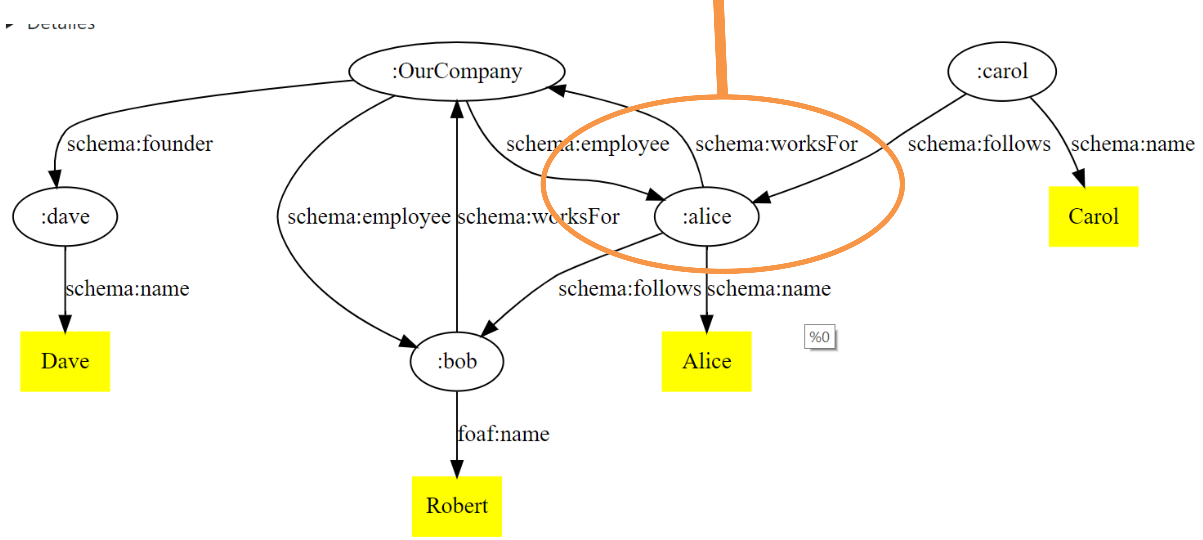
```
<UserShape> {
  schema:name  xsd:string
}
```

Focus Node and Neighborhood

Focus Node = node that is being validated

```
:alice      schema:name     "Alice";
            schema:follows  :bob;
            schema:worksFor :OurCompany .

:bob        foaf:name       "Robert" ;
            schema:worksFor :OurCompany .

:carol      schema:name     "Carol" ;
            schema:follows  :alice .

:dave       schema:name      "Dave" .

:OurCompany schema:founder  :dave ;
            schema:employee :alice, :bob .
```

```
Neighbourhood of :alice = {
  (:alice,      schema:name,     "Alice")
  (:alice,      schema:follows,  :bob),
  (:alice,      schema:worksFor, :OurCompany),
  (:carol,      schema:follows,  :alice),
  (:OurCompany, schema:employee, :alice)
}
```



Shape maps

Shape maps declare which node/shape pairs are selected

- They declare the queries that ShEx engines solve
  - Example: Does `:alice` conform to `<User>` ?
    - `:alice@<User>`
  - Example: Do all subjects of `schema:knows` conform to `<User>` ?
    - `{FOCUS schema:knows _ }@<User>`

3 types of shape maps:

- Query shape maps: Input shape maps
- Fixed shape maps: Simple pairs of node/shape
- Result shape maps: Shape maps generated by the validation process

Shape map resolver

Converts query shape maps to fixed shape maps

Query shape map

```
{FOCUS schema:worksFor _ }@:User
{FOCUS rdf:type schema:Person}@:User,
{_ schema:worksFor FOCUS }@:Company
```

ShapeMap Resolver

Fixed shape map

```
:alice@:User,
:bob@:User,
:carol@:User,
:c1@:Company,
:c2@:Company,
```

RDF Graph

```
:alice a :User .

:bob   schema:worksFor :c1,
                       :c2 .

:carol a schema:Person ;
       schema:worksFor :c1 .
```

ShEx validator

Input: schema, rdf data and fixed shape map, Output: result shape map

ShEx Schema

```
:User {
 schema:name  xsd:string ;
 schema:knows @:User*
}
```

Fixed shape map

```
:alice@:User,
:bob  @:User
```

ShEx Validator

Result Shape map

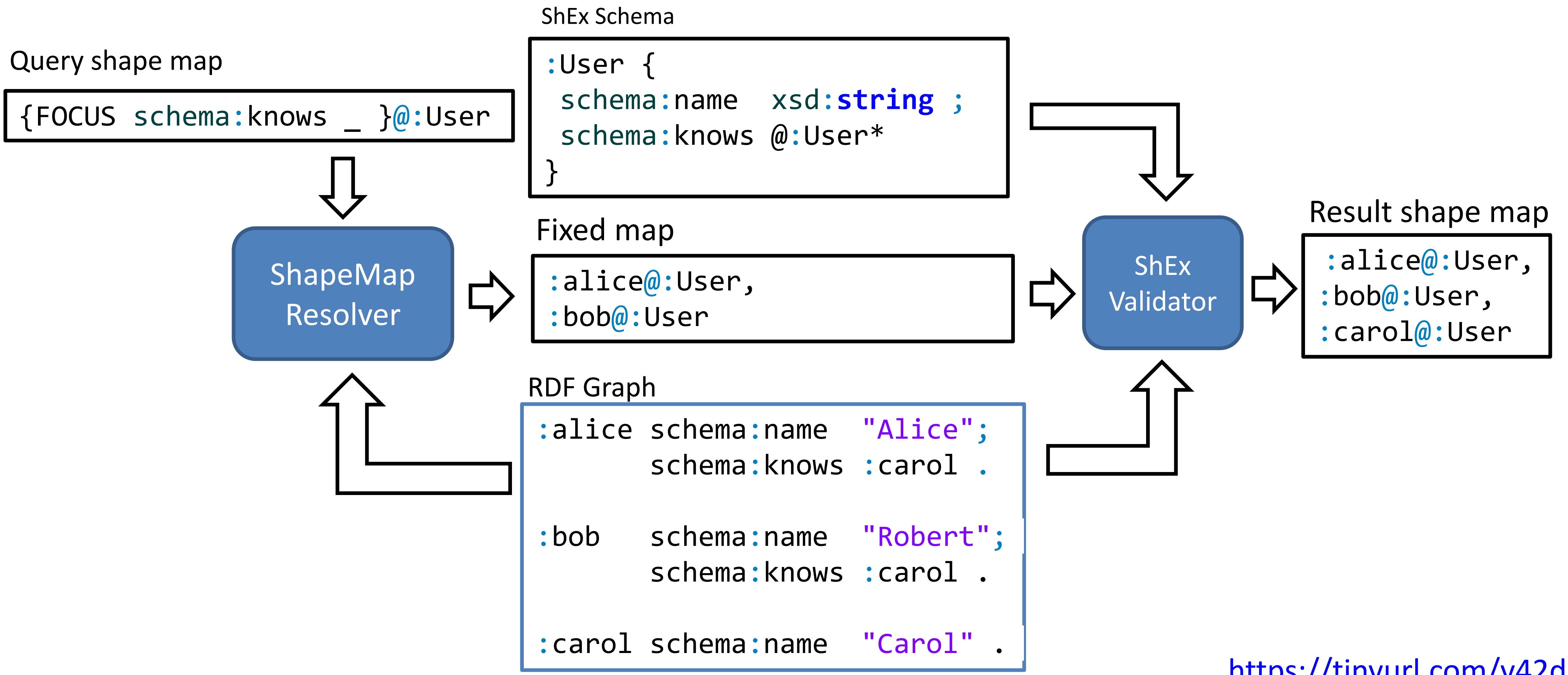
```
:alice@:User,
:bob@:User,
:carol@:User
```

RDF data

```
:alice schema:name  "Alice";
       schema:knows :carol .

:bob   schema:name  "Robert" .

:carol schema:name  "Carol" .
```

Validation process

2 stages: 1) ShapeMap resolver
2) ShEx validator



https://tinyurl.com/y42dclaa

Query maps

A simple language that can be used to generate fixed shape maps

Specification: http://shex.io/shape-map/

Examples:

`:alice@:User`	Checks `:alice` as `:User`
`:alice@:User, :cmp@:Company`	Checks `:alice` as `:User` and `:cmp` as `:Company`
`{_ schema:knows FOCUS }@:User`	Checks nodes who `schema:know` some node
`{FOCUS schema:knows _ }@:User`	Checks nodes who are `schema:known` by some node

SPARQL """
prefix schema: <http://schema.org/>
 	
select ?node where {
 ?node schema:knows ?someone
}
"""@:User
	The same as before
	Any SPARQL query can be used to obtain focus nodes

Node constraints

Constraints over an RDF node

Node constraints

```
:User {
 schema:name        xsd:string ;
 schema:birthDate   xsd:date? ;
 schema:gender      [schema:Male schema:Female] OR xsd:string;
 schema:knows       IRI @:User*
}
```

Node constraints

Triple constraints

Constraints about the incoming/outgoing arcs of a node

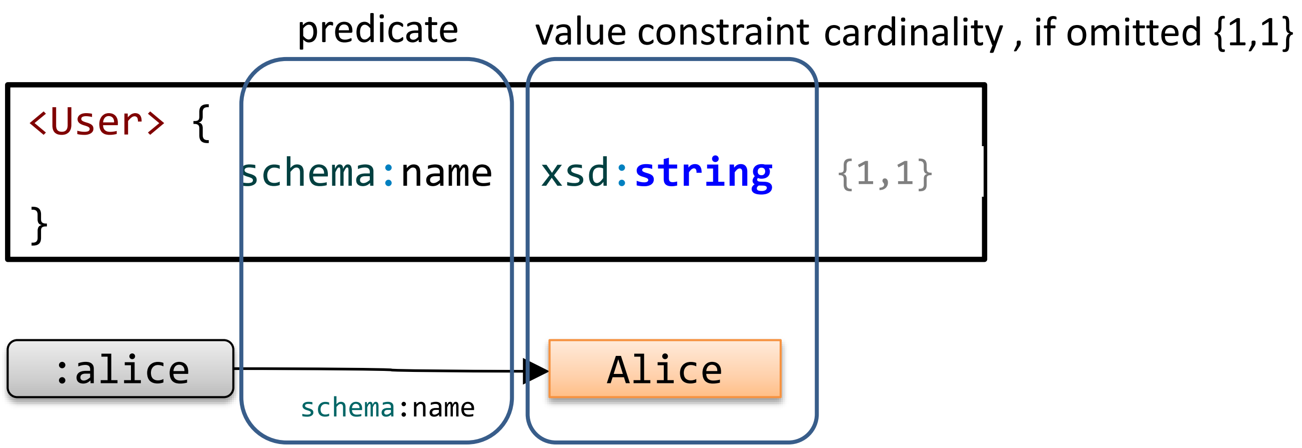
```
:User {
  schema:name        xsd:string ;
  schema:birthDate   xsd:date ? ;
  schema:gender      [schema:Male schema:Female] OR xsd:string;
  schema:knows       IRI @:User *
}
```

Triple constraints

Triple constraints

A basic expression consists of a Triple Constraint

Triple constraint ≈ predicate + value constraint + cardinality



Shape expressions

Labelled rules

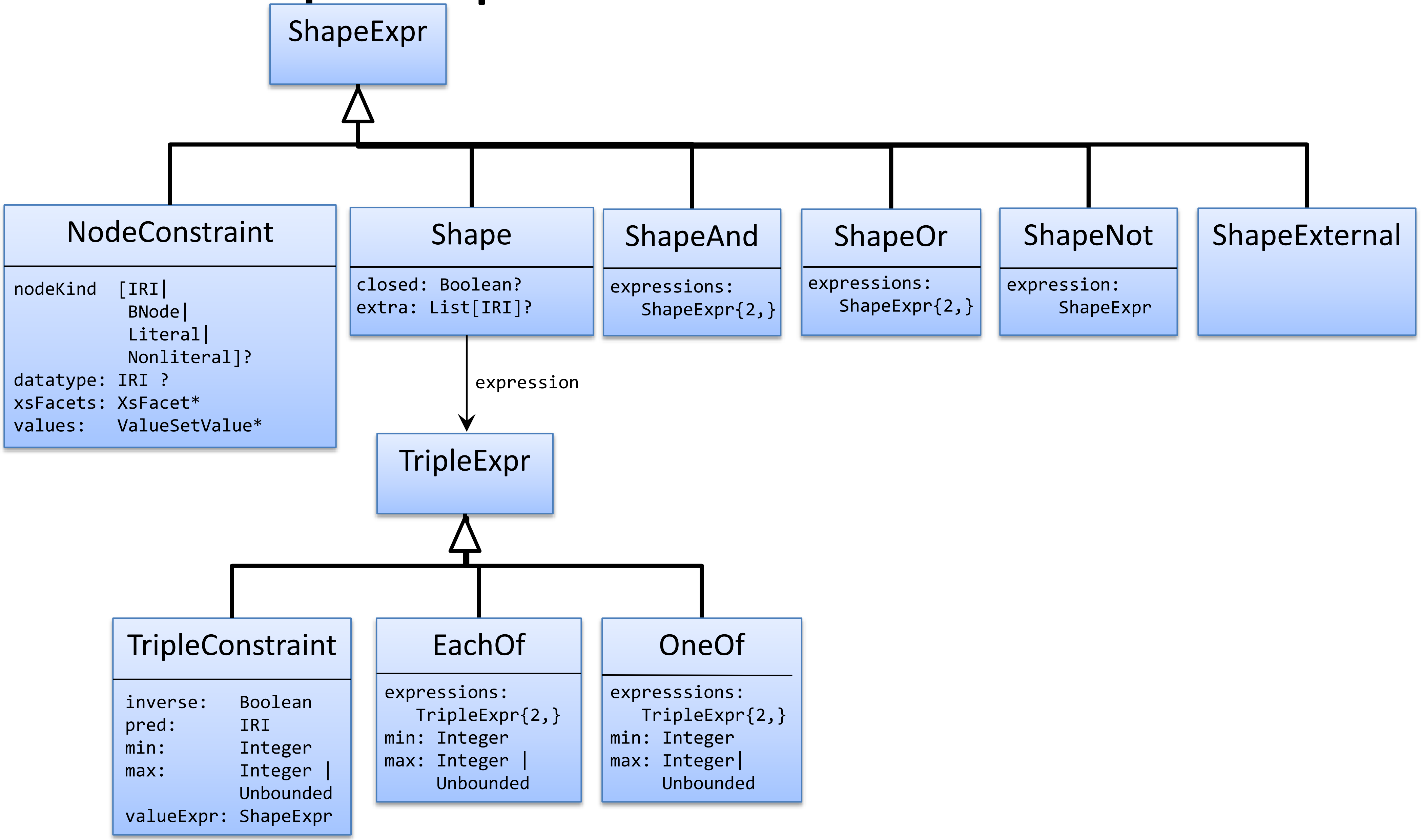
Shape expression label

Shape expression

Triple expression

```
:User {
 schema:name        xsd:string ;
 schema:birthDate   xsd:date ? ;
 schema:gender      [schema:Male schema:Female] OR xsd:string;
 schema:knows       IRI @:User *
}
```

Structure of Shape Expressions



Simple expressions and grouping

The each-of operator ; combines triple expressions

- Unordered sequence

```
:User {
 schema:name   xsd:string  ;
 foaf:age      xsd:integer ;
 schema:email  xsd:string  ;
}
```

```
:alice schema:name  "Alice";
       foaf:age     10 ;
       schema:email "alice@example.org" .

:bob   schema:name  "Robert Smith" ;               ☹
       foaf:age     45 ;
       schema:email <mailto:bob@example.org> .

:carol schema:name   "Carol" ;                     ☹
       foaf:age      56, 66 ;
       schema:email  "carol@example.org" .
```

Repeated properties

A repeated property indicates that **each of** the expressions must be satisfied

```
<User> {
 schema:name   xsd:string;
 schema:parent @<Male>;
 schema:parent @<Female>
}

<Male> {
 schema:gender [schema:Male ]
}

<Female> {
 schema:gender [schema:Female]
}
```

Means that <User> must have two parents, one male and another female

```
:alice schema:name   "Alice" ;
       schema:parent :bob, :carol .

:bob   schema:name   "Bob" ;
       schema:gender schema:Male .

:carol schema:name   "Carol" ;
       schema:gender schema:Female .
```

Try it (RDFShape): https://goo.gl/d3KWPJ

Cardinalities

Inspired by regular expressions

- Traditional operators: *, +, ?
- ...plus cardinalities {m, n}
- If omitted {1,1} = default cardinality

`*`	0 or more
`+`	1 or more
`?`	0 or 1
`{m}`	m repetitions
`{m,n}`	Between m and n repetitions
`{m,}`	m or more repetitions

Example with cardinalities

```
<User> {
 schema:name xsd:string          ;
 schema:worksFor @<Company>    ? ;
 schema:follows  @<User>       *
}

<Company> {
 schema:founder  @<User>       ? ;
 schema:employee @<User> {1,100}
}
```

```
:alice       schema:name     "Alice";
             schema:follows  :bob;
             schema:worksFor :OurCompany .

:bob         schema:name     "Robert" ;
             schema:worksFor :OurCompany .

:carol       schema:name     "Carol" ;
             schema:follows  :alice .

:dave        schema:name      "Dave" .

:OurCompany schema:founder :dave ;
            schema:employee :alice, :bob .
```

Try it: https://goo.gl/ddQHPo

Choices - OneOf

The one-of operator | represents alternatives (either one or the other)

```
:User {
  schema:name  xsd:string ;
| schema:givenName xsd:string + ;
  schema:lastName  xsd:string
}
```

```
:alice schema:name      "Alice Cooper" .

:bob   schema:givenName "Bob", "Robert" ;
       schema:lastName  "Smith" .

:carol schema:name      "Carol King" ;   ☹
       schema:givenName "Carol" ;
       schema:lastName  "King" .

:dave  foaf:name        "Dave" .         ☹
```

Try it (RDFShape): https://goo.gl/hCvyRN

Node constraints

Type	Example	Description
Anything	.	The value can be anything
Datatype	`xsd:string`	Matches a literal with datatype `xsd:string`
Kind	`IRI BNode Literal NonLiteral`	The object must have that kind
Value set	`[:Male :Female ]`	The value must be `:Male` or `:Female`
Reference	`@<User>`	The value must have shape `<User>`
Composed with OR AND NOT	`xsd:string OR IRI`	The value must have datatype `xsd:string` or be an IRI
IRI Range	`foaf:~`	The value must start with the IRI associated with `foaf`
Any except...	`- :Checked`	Any value except `:Checked`

No constraint

A dot (.) matches anything ⇒ no constraint on values

```
:User {
 schema:name        . ;
 schema:affiliation . ;
 schema:email       . ;
 schema:birthDate   .
}
```

```
:alice
   schema:name        "Alice";
   schema:affiliation [ schema:name "OurCompany" ] ;
   schema:email       <mailto:alice@example.org> ;
   schema:birthDate   "2010-08-23"^^xsd:date .
```

Try it: https://goo.gl/LNVg4p

Datatypes

Datatypes are directly declared by their URIs

Predefined datatypes from XML Schema:

`xsd:string xsd:integer xsd:date ...`

```
:User {
 schema:name      xsd:string;
 schema:birthDate xsd:date
}
```

```
:alice schema:name      "Alice";
       schema:birthDate "2010-08-23"^^xsd:date.

:bob   schema:name      "Robert" ;        ☹
       schema:birthDate "Unknown" .

:carol schema:name      _:unknown ;       ☹
       schema:birthDate 2012 .
```

Try it: https://goo.gl/neVWeC

Facets on Datatypes

It is possible to qualify the datatype with XML Schema facets

See: http://www.w3.org/TR/xmlschema-2/#rf-facets

Facet	Description
MinInclusive, MaxInclusive MinExclusive, MaxExclusive	Constraints on numeric values which declare the min/max value allowed (either included or excluded)
TotalDigits, FractionDigits	Constraints on numeric values which declare the total digits and fraction digits allowed
Length, MinLength, MaxLength	Constraints on string values which declare the length allowed, or the min/max length allowed
/… /	Regular expression pattern

Facets on Datatypes

```
:User {
 schema:name  xsd:string  MaxLength 10 ;
 foaf:age     xsd:integer MinInclusive 1 MaxInclusive 99 ;
 schema:phone xsd:string  /\\d{3}-\\d{3}-\\d{3}/
}
```

```
:alice schema:name  "Alice";
       foaf:age     10 ;
       schema:phone "123-456-555" .

:bob   schema:name  "Robert Smith" ;       ☹
       foaf:age     45 ;
       schema:phone "333-444-555" .

:carol schema:name  "Carol" ;              ☹
       foaf:age     23 ;
       schema:phone "23-456-555" .
```

Try it: https://goo.gl/8KanuJ

Node Kinds

Define the kind of RDF nodes: Literal, IRI, BNode, ...

Value	Description	Examples

`Literal`	Literal values	`"Alice"`
		`"Spain"@en`
		`23`
		`true`

`IRI`	IRIs	`<http://example.org/alice>` `ex:alice`
`BNode`	Blank nodes	`_:1`

`NonLiteral`	Blank nodes or IRIs	`_:1`
		`<http://example.org/alice>`
		`ex:alice`

Example with node kinds

```
:User {
 schema:name    Literal ;
 schema:follows IRI
}
```

```
:alice schema:name    "Alice" ;
       schema:follows :bob .

:bob   schema:name    :Robert ;   ☹
       schema:follows :carol .

:carol schema:name    "Carol" ;   ☹
       schema:follows "Dave"  .
```

Try it: https://goo.gl/B6x2rE

Value sets

The value must be one of the values of a given set

Denoted by [and]

```
:Product {
 schema:color        [ "Red" "Green" "Blue" ] ;
 schema:manufacturer [ :OurCompany :AnotherCompany ]
}
```

```
:x1 schema:color "Red";
    schema:manufacturer :OurCompany .

:x2 schema:color "Cyan" ;
    schema:manufacturer :OurCompany .       ☹

:x3 schema:color "Green" ;
    schema:manufacturer :Unknown .          ☹
```

Try it: https://goo.gl/AJ1eQX

Single value sets

Value sets with a single element

A very common pattern

```
<SpanishProduct> {
 schema:country [ :Spain ]
}

<FrenchProduct> {
 schema:country [ :France ]
}

<VideoGame> {
 a [ :VideoGame ]
}
```

```
:product1 schema:country :Spain .

:product2 schema:country :France .

:product3 a :VideoGame ;
          schema:country :Spain .
```

Note: ShEx doesn't interact with inference

It just checks if there is an `rdf:type` arc

- Inference can be done before/after validating
- It can even be used to validate inference systems

Try it: https://goo.gl/NpZN9n

Language tagged literals

```
:FrenchProduct {
 schema:label [ @fr ]
}

:SpanishProduct {
 schema:label [ @es @es-AR @es-ES ]
}
```

```
:car1 schema:label "Voiture"@fr .    # Passes as  :FrenchProduct

:car2 schema:label "Auto"@es .       # Passes as :SpanishProduct

:car3 schema:label "Carro"@es-AR .   # Passes as  :SpanishProduct

:car4 schema:label "Coche"@es-ES .   # Passes as  :SpanishProduct
```

Shape references

Defines that the value must match another shape

- References are marked as @

```
:User {
 schema:name xsd:string     ;
 schema:worksFor @:Company
}

:Company {
 schema:founder xsd:string
}
```

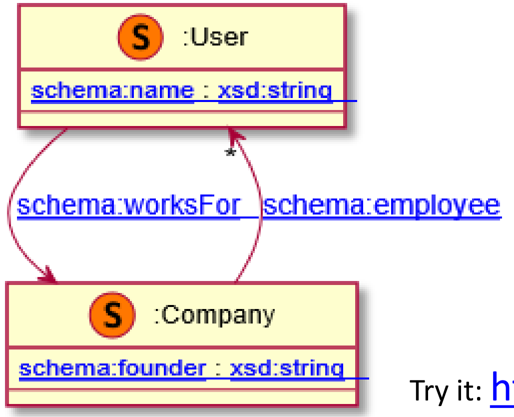
```
:alice a :User;
       schema:worksFor :OurCompany .

:bob   a :User;                          ☹
       schema:worksFor :Another .

:OurCompany
       schema:name     "OurCompany" .

:Another                                 ☹
       schema:name      23 .
```

Try it: https://goo.gl/Q3SriH

Recursion and cyclic data models

```
:User {
 schema:name      xsd:string  ;
 schema:worksFor @:Company ;
}

:Company {
 schema:founder  xsd:string  ;
 schema:employee @:User *
}
```

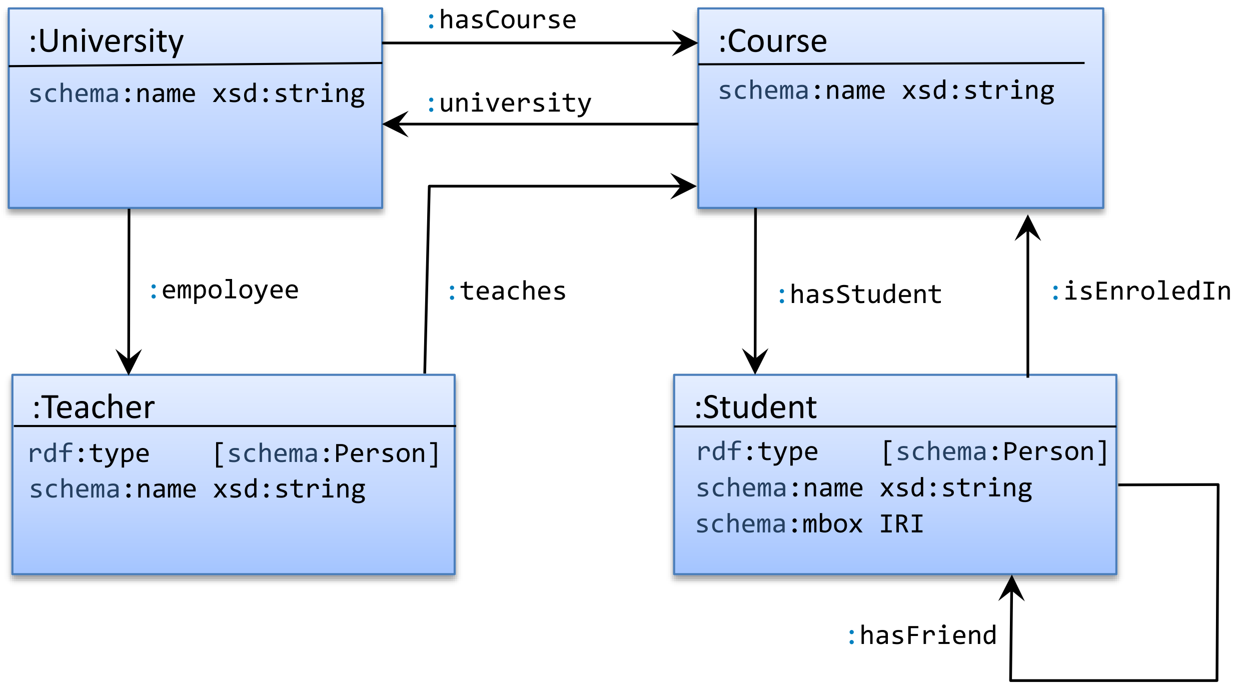
```
:alice    schema:name     "Alice";;
          schema:worksFor :OurCompany .

:bob      schema:name     "Robert";        ☹
          schema:worksFor :Another .

:companyA schema:founder     "Carol";
          schema:employee    :alice .

:companyB schema:founder     "Another" .   ☹
          schema:employee    :unknown .
```



Try it: https://goo.gl/eMNiyR

Exercise

Define a Schema for the following domain model



IRI ranges

`uri:~` represents the set of all URIs that start with stem `uri`

```
prefix codes: <http://example.codes/>

:User {
  :status [ codes:~ ]
}
```

```
prefix codes: <http://example.codes/>
prefix other: <http://other.codes/>

:x1 :status codes:resolved .

:x2 :status other:done .                  ☹

:x3 :status <http://example.codes/pending> .
```

Try it: https://goo.gl/EC521J

IRI Range exclusions

The operator - excludes IRIs or IRI ranges from an IRI range

```
prefix codes: <http://example.codes/>

:User {
  :status [
      codes:~ - codes:deleted
  ]
}
```

```
:x1 :status codes:resolved .

:x2 :status other:done.          ☹

:x3 :status <http://example.codes/pending> .

:x4 :status codes:deleted .      ☹
```

Try it: https://goo.gl/pU8u4b

Nested shapes

Syntax simplification to avoid defining two shapes

- Internally, the inner shape is identified using a blank node

```
User {
  schema:name      xsd:string ;
  schema:worksFor _:1
}

_:1 a [ schema:Company ] .
```

≡

```
:User {
  schema:name      xsd:string ;
  schema:worksFor {
    a [ schema:Company ]
  }
}
```

```
:alice schema:name      "Alice" ;
       schema:worksFor :OurCompany .

:OurCompany a schema:Company .
```

Try it (RDFShape): https://goo.gl/2Eoehi

Labeled constraints

`$label` <constraint> associates a constraint to a label

It can later be used as `&label`

```
:User {
 $:name ( schema:name .
        | schema:givenName . ;
          schema:familyName .
        ) ;
 schema:email IRI
}
:Employee {
  &:name ;
  :employeeId .
}
```

```
:Employee {
  ( schema:name .
  | schema:givenName . ;
    schema:familyName .) ;
  :employeeId .
}
```

Inverse triple constraints

^ reverses the order of the triple constraint

```
:User {
 schema:name      xsd:string ;
 schema:worksFor @:Company
}

:Company {
 a         [schema:Company] ;
 ^schema:worksFor @:User+
}
```

```
:alice schema:name "Alice";
       schema:worksFor :OurCompany .

:bob schema:name "Bob" ;
     schema:worksFor :OurCompany .

:OurCompany a schema:Company .
```

Try it (RDFShape): https://goo.gl/9FbHi3

Allowing other triples

Triple constraints limit all triples with a given predicate to match one of the constraints

This is called *closing a property*

Example:

```
<Company> {
 a [ schema:Organization ] ;
 a [ org:Organization ]
}
```

```
:OurCompany a org:Organization,
              schema:Organization .

:OurUniversity a org:Organization,      ☹
                 schema:Organization,
                 schema:CollegeOrUniversity .
```

Sometimes we would like to permit other triples (open the property)

Allowing other triples

EXTRA `<listOfProperties>` declares that a list of properties can contain extra values

```
<Company> EXTRA a {
 a [ schema:Organization ] ;
 a [ org:Organization ]
}
```

```
:OurCompany a org:Organization,
              schema:Organization .

:OurUniversity a org:Organization,
                 schema:Organization,
                 schema:CollegeOrUniversity .
```

Try it: https://goo.gl/MxZVts

Closed Shapes

CLOSED can be used to limit the appearance of any predicate not mentioned in the shape expression

```
<User> {
 schema:name IRI;
 schema:knows @<User>*
}
```

By default open, so all match <User>

```
:alice schema:name "Alice" ;
       schema:knows :bob .

:bob schema:name "Bob" ;
     schema:knows :alice .

:dave schema:name "Dave" ;
      schema:knows :emily ;
      :link2virus <virus> .

:emily schema:name "Emily" ;
       schema:knows :dave .
```

```
<User> CLOSED {
 schema:name IRI;
 schema:knows @<User>*
}
```

With closed, only `:alice` and `:bob` match <User>

Node constraints

Constraints on the focus node

```
<User> IRI {
  schema:name xsd:string ;
  schema:worksFor IRI
}
```

```
:alice schema:name "Alice";
  :worksFor :OurCompany .

_:1 schema:name "Unknown";
  :worksFor :OurCompany .
```

☹

Composing Shape Expressions

It is possible to use AND , OR and NOT to compose shapes

```
:User {
 schema:name      xsd:string ;
 schema:worksFor IRI AND @:Company ?;
 schema:follows   IRI OR BNode *
}

:Company {
 schema:founder   IRI ?;
 schema:employee IRI {1,100}
}
```

```
:alice       schema:name      "Alice";
             schema:follows   :bob;
             schema:worksFor :OurCompany .

:bob         schema:name      "Robert" ;
             schema:worksFor [
              schema:Founder "Frank" ;
              schema:employee :carol ;
             ] .

:carol       schema:name      "Carol" ;
             schema:follows   [
              schema:name "Emily" ;
             ] .

:OurCompany schema:founder   :dave ;
             schema:employee :alice, :bob .
```

☹

Try it: https://goo.gl/auLBiu

Implicit AND

AND can be omitted between a node constraint and a shape

```
:User {
  schema:name xsd:string ;
  schema:worksFor IRI AND @:Company
}
```

⇔

```
:User {
  schema:name xsd:string ;
  schema:worksFor IRI @:Company
}
```

Conjunction of Shape Expressions

AND can be used to define conjunction on Shape Expressions

```
<User> { schema:name xsd:string ;
         schema:worksFor IRI
       }
       AND {
         schema:worksFor @<Company>
       }
```

Conjunctions are the default operator in SHACL

Using AND to extend shapes

AND can be used as a basic form of inheritance

```
:Person {
 a                      [ schema:Person ] ;
 schema:name            xsd:string  ;
}

:User @:Person AND {
 schema:name            MaxLength 20 ;
 schema:email           IRI
}

:Student @:User AND {
 :course                IRI *;
}
```

```
:alice a                schema:Person ;
       schema:name      "Alice" .

:bob schema:name        "Robert";
     schema:email       <bob@example.org> .

:carol a                schema:Person;
       schema:name      "Carol" ;
       schema:email     <carol@example.org> .

:dave  a     schema:Person;
       schema:name      "Carol" ;
       schema:email     <carol@example.org>;
       :course          :algebra .
```

Disjunction of Shape Expressions

OR can be used to define disjunction of Shape Expressions

```
:User { schema:name xsd:string }
   OR { schema:givenName xsd:string ;
        schema:familyName xsd:string
      }
```

Inclusive-or

```
:User { schema:name xsd:string
      | schema:givenName xsd:string ;
        schema:familyName xsd:string
      }
```

Exclusive-or

Disjunction of datatypes

```
:Product {
 rdfs:label          xsd:string OR rdf:langString;
 schema:releaseDate xsd:date OR xsd:gYear OR
                     [ "unknown-past" "unknown-future" ]
}
```

```
:p1 a :Product ;                                    #Passes as a :Product
    rdfs:label "Laptop";
    schema:releaseDate "1990"^^xsd:gYear .

:p2 a :Product ;                                    #Passes as a :Product
    rdfs:label "Car"@en ;
    schema:releaseDate "unknown-future" .

:p3 a :Product ;                                    #Fails as a :Product
    rdfs:label :House ;
    schema:releaseDate "2020"^^xsd:integer .
```

Exercise

Emulate recursive property paths in ShEx

- A node conforms to :Person if it has rdf:type schema:Person or if it has a type that is a rdfs:subClassOf some type that has rdf:type schema:Person

```
:alice     a schema:Person .               #Passes as :Person

:bob       a :Teacher .                    #Passes as :Person

:carol     a :Assistant .                  #Passes as :Person

:Teacher   rdfs:subClassOf schema:Person .
:Assistant rdfs:subClassOf :Teacher .
```

Negation

NOT s creates a new shape expression from a shape s.

Nodes conform to NOT s when they do not conform to s.

```
:NoName Not {
   schema:name .
}
```

```
:alice schema:givenName  "Alice" ;
       schema:familyName "Cooper" .

:bob   schema:name       "Robert" .          ☹

:carol schema:givenName  "Carol" ;           ☹
       schema:name       "Carol" .
```

Try it: https://goo.gl/GMvXy7

IF-THEN pattern

All products must have a schema:productID and if a product has type schema:Vehicle, then it must have the properties schema:vehicleEngine and schema:fuelType.

```
:kitt schema:productID "C21";      # Passes as :Product
      a schema:Vehicle;
      schema:vehicleEngine :x42 ;
      schema:fuelType :electric .

:bad  schema:productID "C22";      # Fails as :Product
      a schema:Vehicle;
      schema:fuelType :electric .

:c23  schema:productID "C23" ;     # Passes as :Product
      a schema:Computer .
```

IF-THEN-ELSE pattern

If a product has type schema:Vehicle, then it must have the properties schema:vehicleEngine and schema:fuelType, otherwise, it must have the property schema:category with a xsd:s

```
:kitt a schema:Vehicle;              # Passes as :Product
        schema:vehicleEngine :x42 ;
        schema:fuelType :electric .

:c23  a schema:Computer ;            # Passes as :Product
        schema:category "Laptop" .

:bad1 a schema:Vehicle;              # Fails as :Product
        schema:fuelType :electric .

:bad2 a schema:Computer .            # Fails as :Product
```

Cyclic dependencies with negation

One problem of combining NOT and recursion is the possibility of declaring ill-defined shapes

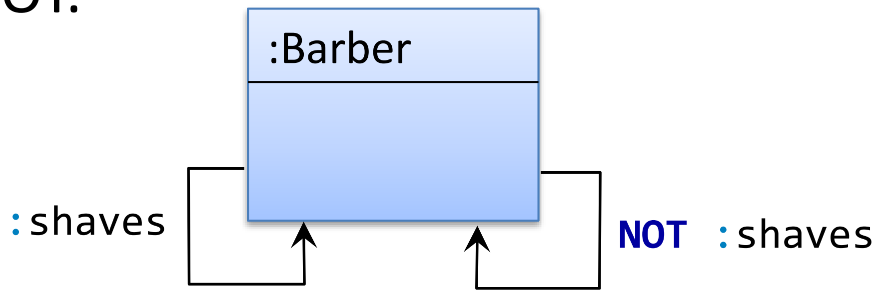
```
:Barber {                    # Violates the negation requirement
 :shaves     @:Person
} AND NOT {
 :shaves     @:Barber
}

:Person {
 schema:name xsd:string
}
```

```
:albert :shaves :dave .           # Passes as a :Barber

:bob schema:name "Robert" ;       # Passes as a :Person
     :shaves :bob .               # Passes :Barber?

:dave schema:name "Dave" .        # Passes as a :Person
```

Restriction on cyclic dependencies and negation

Requirement to avoid ill formed data models:

Whenever a shape refers to itself either directly or indirectly, the chain of references cannot traverse an occurrence of the negation operation NOT.



:Barber shape is rejected

Semantic Actions

- Arbitrary code attached to shapes
  - Can be used to perform operations with side effects
  - Independent of any language/technology
    - Several extension languages: GenX, GenJ (http://shex.io/extensions/)

```
<Person> {
 schema:name xsd:string,
 schema:birthDate xsd:dateTime
 %js:{ report = _.o; return true; %},
 schema:deathDate xsd:dateTime
 %js:{ return _[1].triple.o.lex > report.lex; %}
 %sparql:{
  ?s schema:birthDate ?bd . FILTER (?o > ?bd) %}
}
```

```
:alice schema:name "Alice" ;
  schema:birthDate "1980-01-23"^^xsd:date ;
  schema:deathDate "2013-01-23"^^xsd:date .

:bob schema:name "Robert" ;
  schema:birthDate "2013-08-12"^^xsd:date ;
  schema:deathDate "1990-01-23"^^xsd:date .
```

Importing schemas

The import statement allows to import schemas

http://example.org/Person.shex

```
:Person {
   $:name ( schema:name .
          | schema:givenName . ; schema:familyName .
          ) ;
   schema:email .
}
```

```
import <http://example.org/Person.shex>

:Employee {
    &:name ;
    schema:worksFor <CompanyShape>
}

:Company {
    schema:employee @:Employee ;
    schema:founder  @:Person ;
}
```

```
:alice schema:name      "Alice";
       schema:worksFor :OurCompany .

:OurCompany schema:employee :alice ;
            schema:founder  :bob .

:bob schema:name "Robert" ;
     schema:email <mailto:bob@example.com> .
```

Annotations

Annotations are lists (predicate, object) that can be associated to an element

Specific annotations can be defined for special purposes, e.g. forms, UI, etc.

```
:Person {
 schema:name       xsd:string
  // rdfs:label     "Name"
  // rdfs:comment   "Name of person" ;

 schema:birthDate xsd:date
   // rdfs:label     "birthDate"
   // rdfs:comment   "Birth of date" ;
}
```

Other features

Current ShEx version: 2.1

Some features postponed for next version

- Inheritance (extends/abstract)
- UNIQUE

Future work & contributions

More info http://shex.io

- ShEx currently under active development
- Curent work
  - Improve error messages
  - Inheritance of shape expressions

If you are interested, you can help

- List of issues: https://github.com/shexSpec/shex/issues

Shapes tools

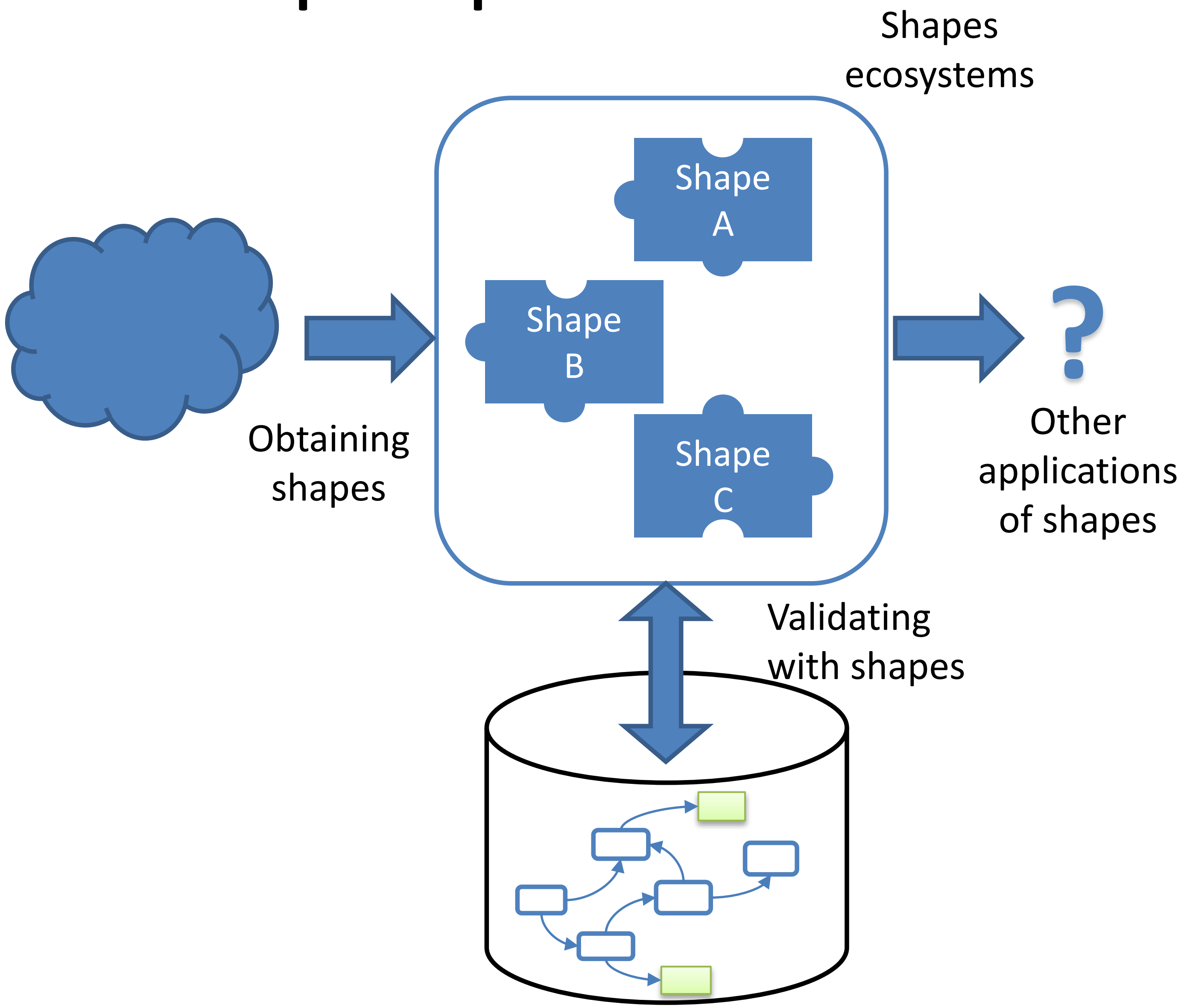
Tools: challenges and perspectives

Validating with shapes

Obtaining shapes

Other applications of shapes

Shapes ecosystems



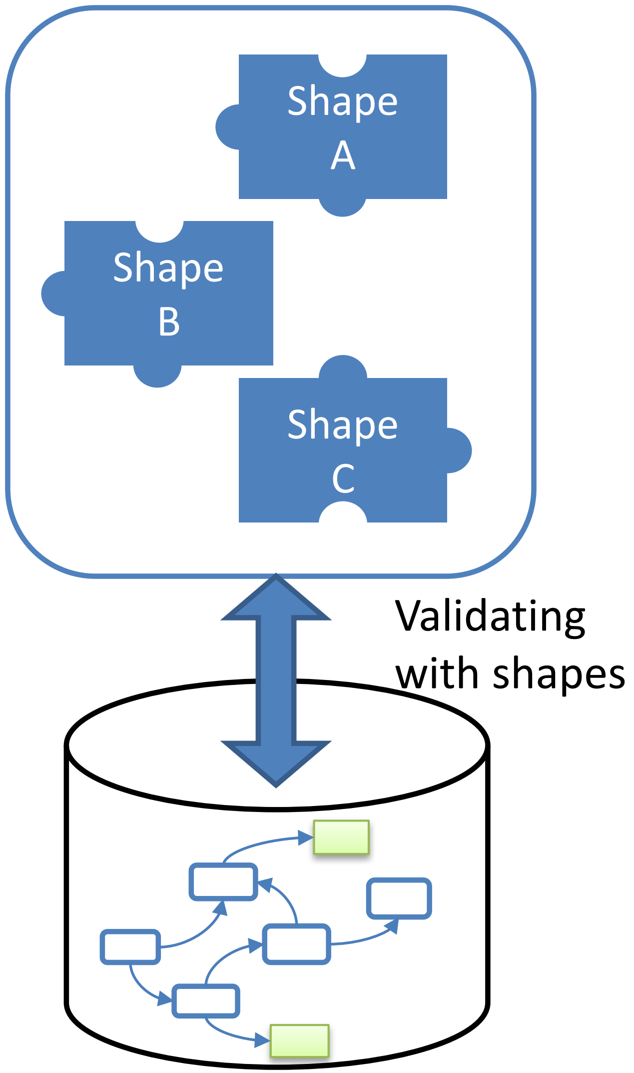
Validating with shapes

Libraries and command line validators

Online demos

Integrated in ontology editors

Continuous integration with Shapes



Libraries and command line validators

Examples at WESO

SHaclEx

RDF validation
Generic schemas: ShEx/SHACL
Converter ShEx ↔ SHACL

ShEx-s

Scala implementation of ShEx
Supports ShEx 2.1 (testsuite)

SHACL-s

Scala implementation of SHACL
Supports SHACL core (testsuite)

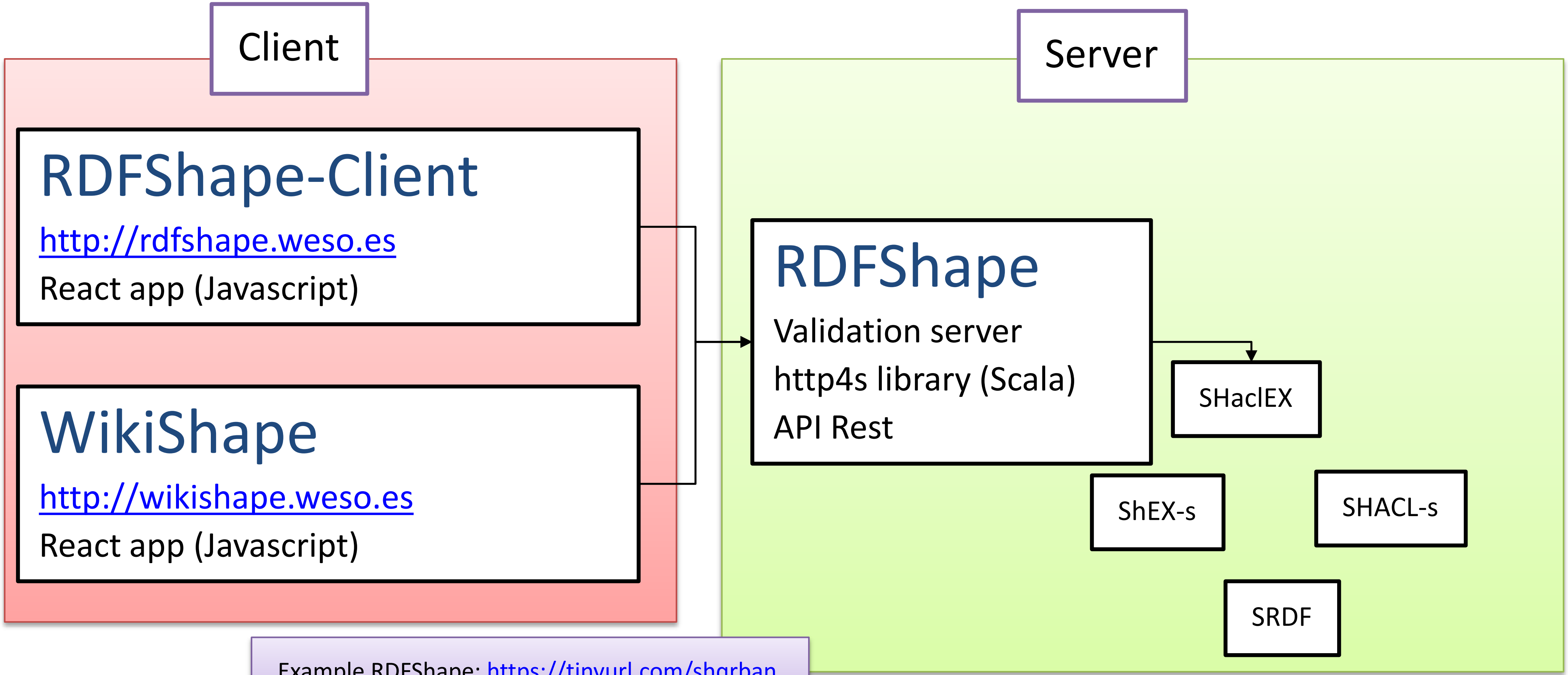
SRDF

Simple RDF - Generic RDF interface
Implementation for RDF4j and Apache Jena

All libraries are available at: https://github.com/weso/

Online demos

Web Demos and playgrounds



Example RDFShape: https://tinyurl.com/shqrban

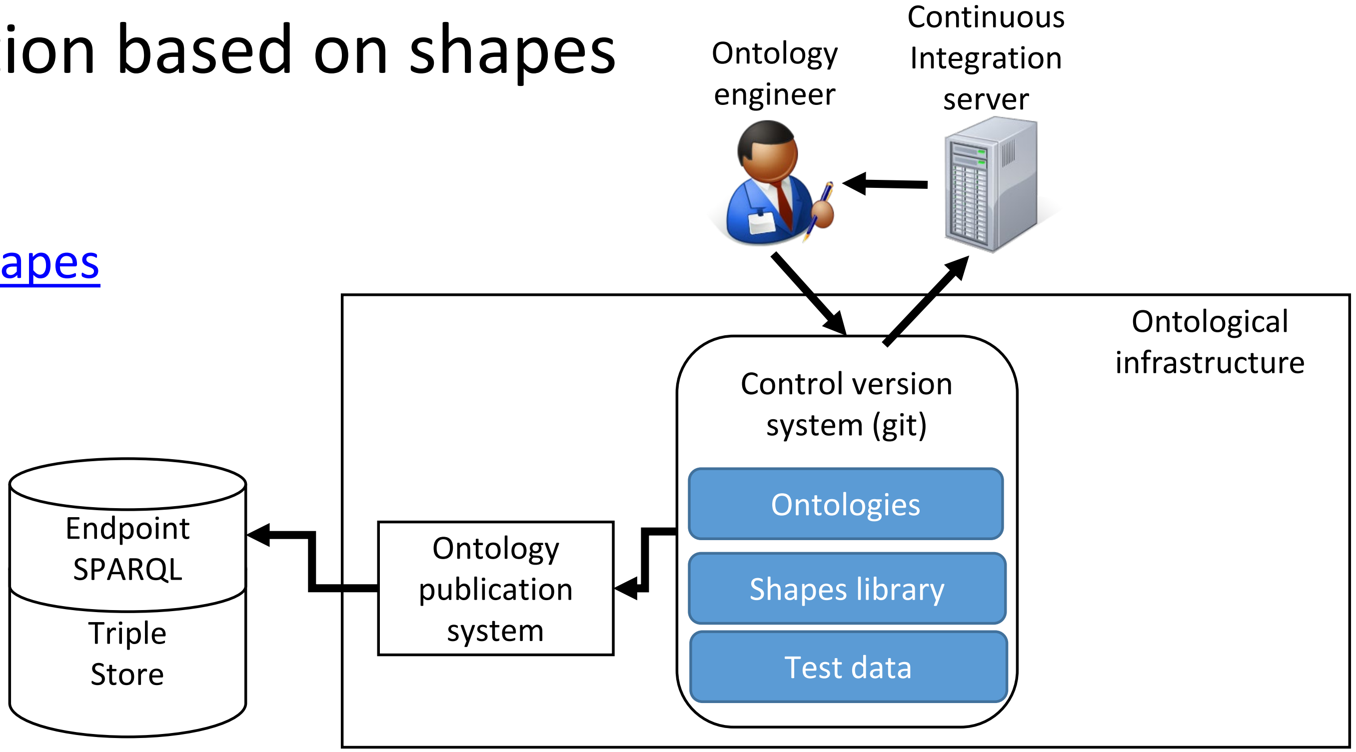
Continuous integration with Shapes

Coexistence between ontologies/shapes

- Shapes can validate the behaviour of inference systems
- Shapes pre- and post- inference
- TDD and continuous integration based on shapes

Gene Ontology Shapes:

- https://github.com/geneontology/go-shapes



Continuous integration with Shapes

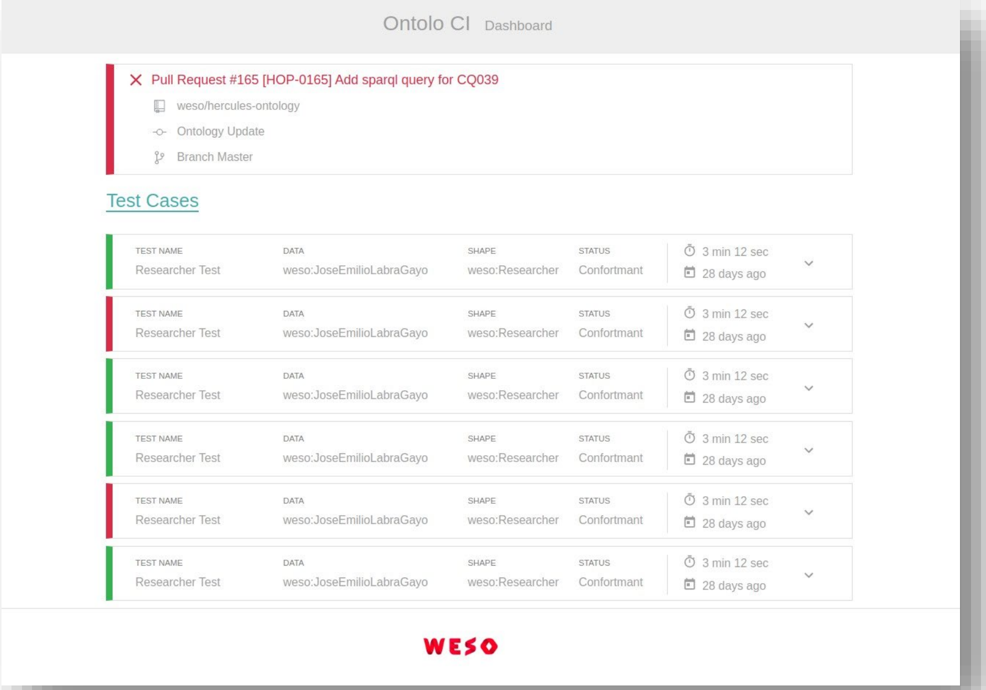
Ontolo-ci: https://www.weso.es/ontolo-ci/

Developed as part of HERCULES-Ontology

Test-Driven-Development applied to Ontologies

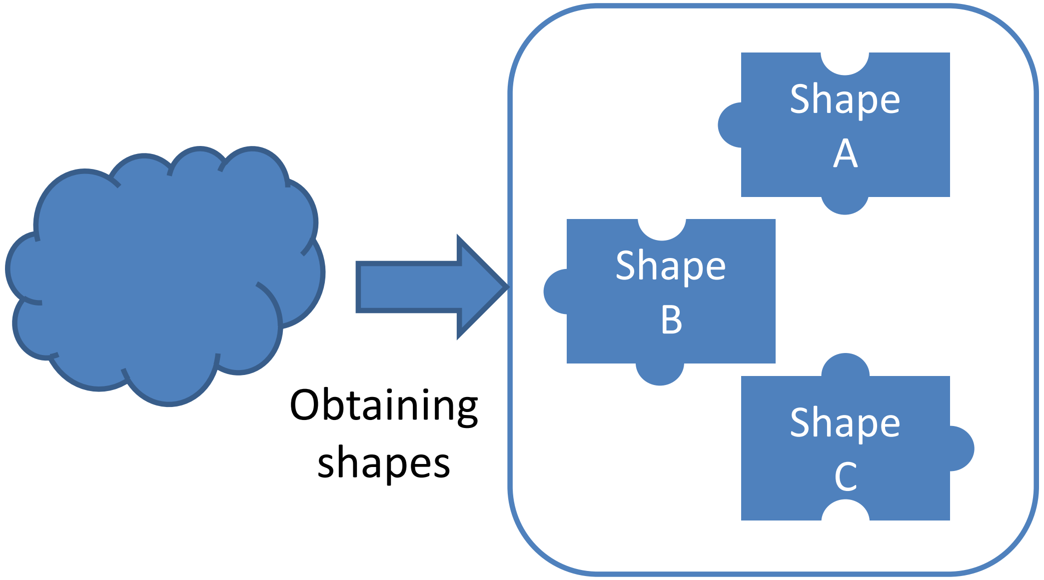
Input:

- Ontologies
- Shapes
- Test data
- Input shape map (SPARQL competency question)
- Expected result shape map



Obtaining shapes

- Shapes editors
  - Text-based editors
  - Visual editors and visualizers
- Obtaining shapes from...
  - Spreadsheets
  - RDF data
  - Ontologies
  - Other schemas (XML Schema)



Text-based editors

YaSHE: Forked from YASGUI: http://www.weso.es/YASHE/

- Syntax highlighting
- Auto-completion

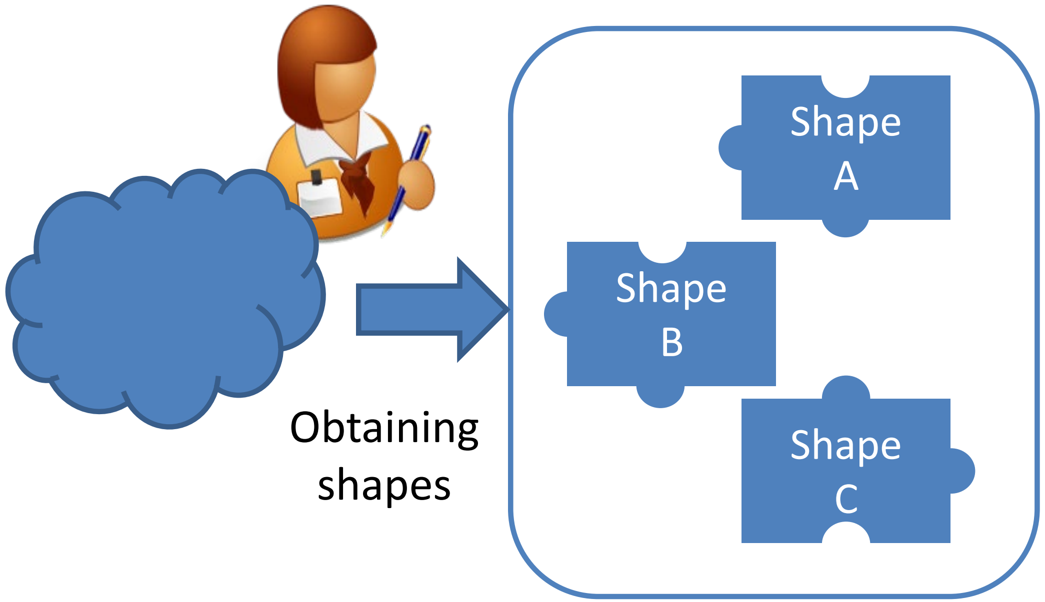
```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
prefix wd: <http://www.wikidata.org/entity/>
prefix wdt: <http://www.wikidata.org/prop/direct/>

# Example SPARQL query: select ?researcher where { ?researcher wdt:P106 wd:Q1650915 } limit 5

<Researcher> EXTRA wdt:P31 wdt:P106 {
  wdt:P31  [ wd:Q5 ]            ; # Instance of = human
  wdt:P106 [ wd:Q1650915 ]      ; # Occupation = researcher
  wdt:P101 @<Discipline>    *   ; # Field of work
  wdt:P496 xsd:string       ?   ; # ORCID-ID
  wdt:P1153 xsd:string      ?   ; # Scopus-Author ID
}
```

Scopus Author ID (P1153)

identifier for an author assigned in Scopus bibliographic database

Shapes author tools

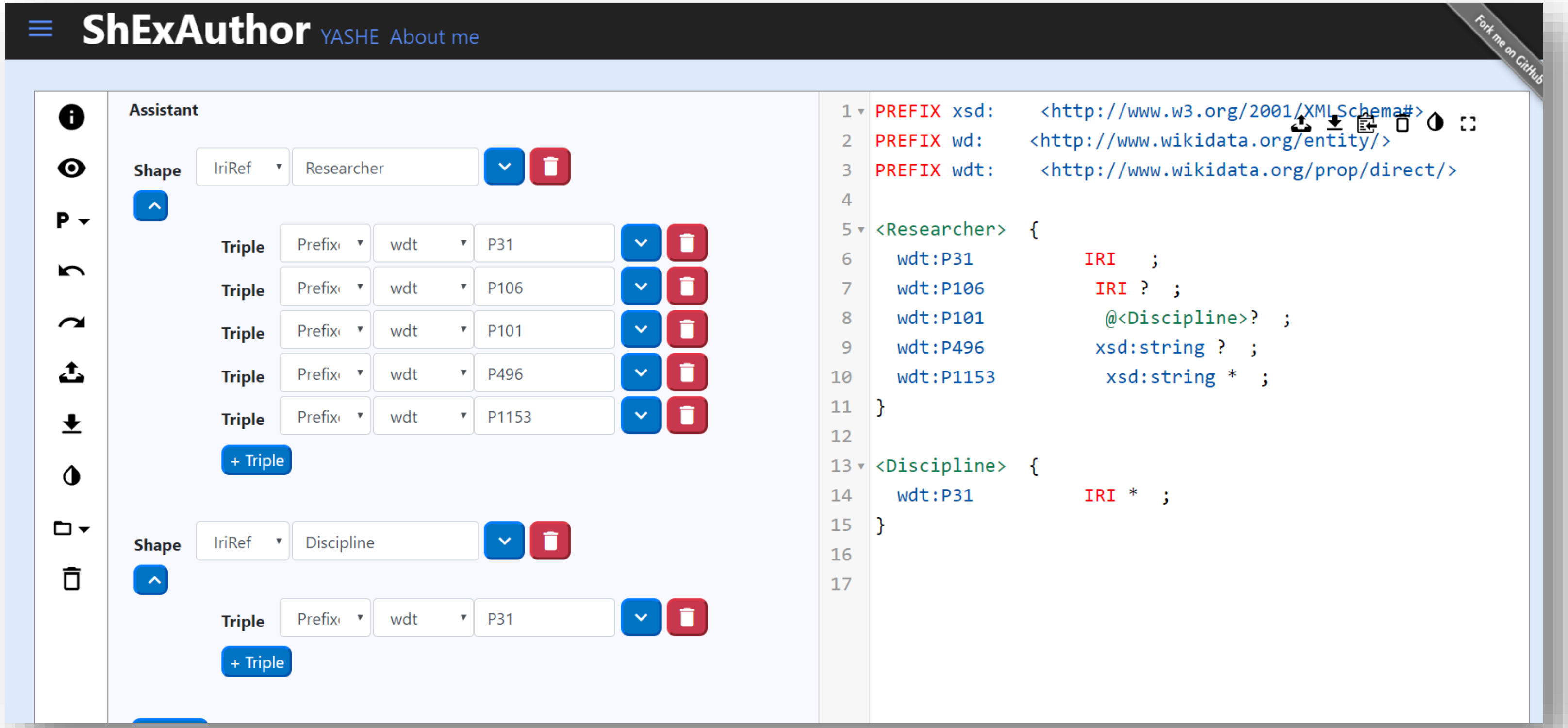
ShEx-Author



Shapes author tools: ShEx Author

ShEx-Author: Inspired by Wikidata Query Service

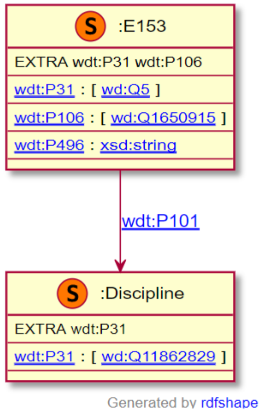
2 column: Visual one synchronized with text based

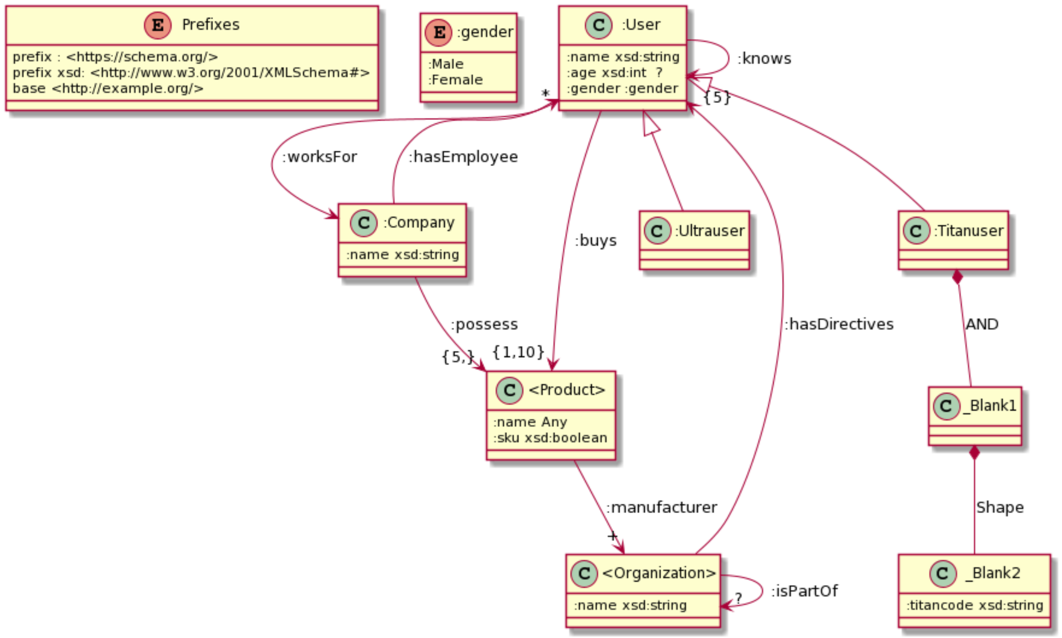


Shapes visualization

Integrated in RDFShape/Wikishape

- UMLSHaclEX UML diagrams for ShEx
- ShUMLex: Conversion to UML through XMI





Shapes from spreadsheets

ShExstatements: https://shexstatements.toolforge.org/

ShExCSV: CSV representation of Shapes

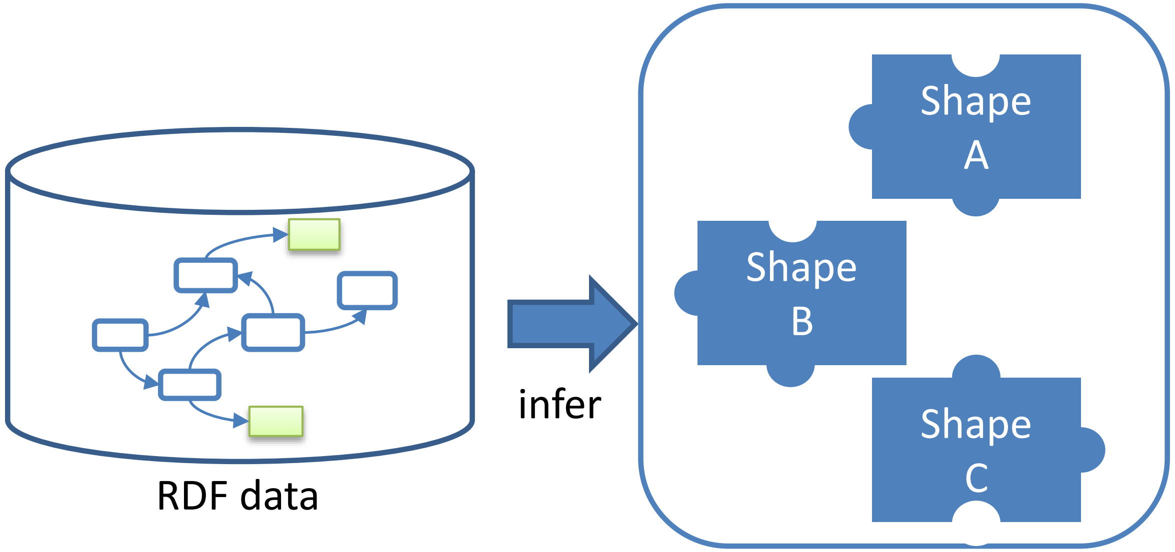
Hermes: ShExCSV processor, https://github.com/weso/hermes

Generating Shapes from RDF data

Useful use case in practice

Some prototypes

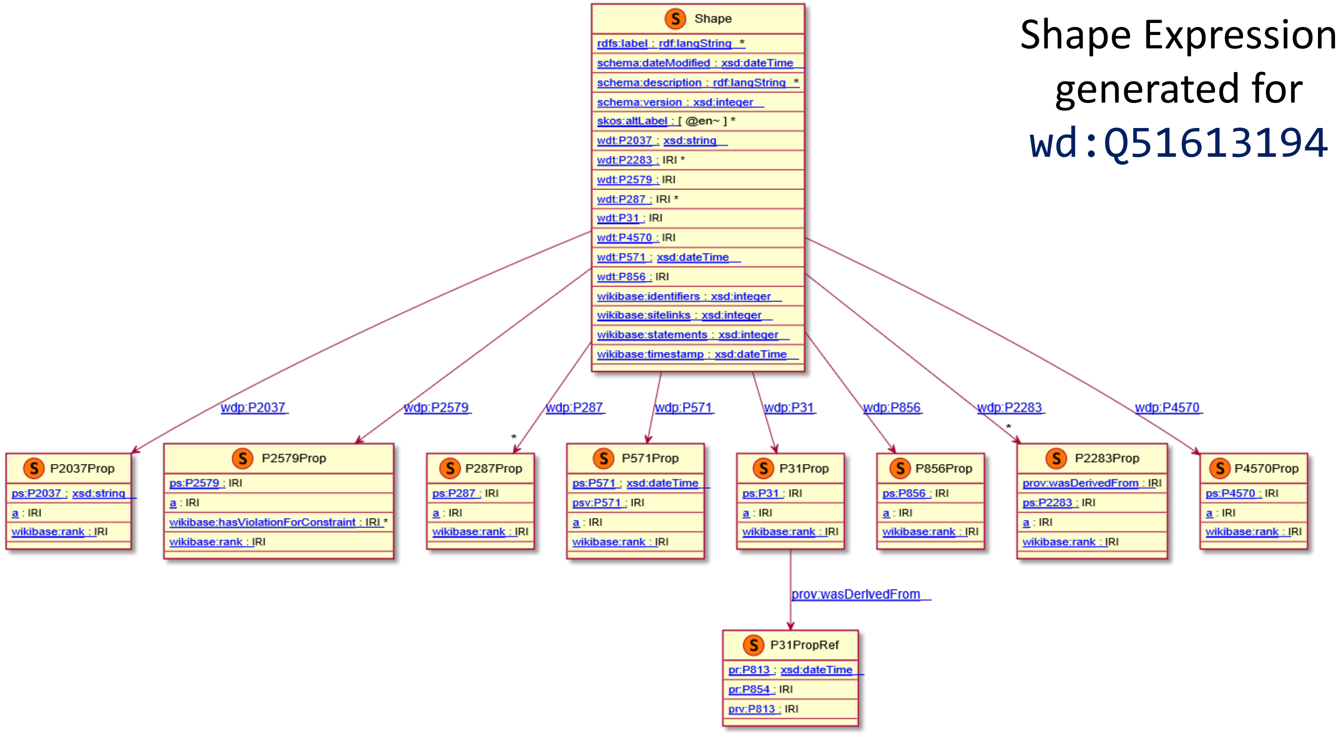
- sheXer: http://shexer.weso.es/
- RDFShape: http://rdfshape.weso.es
- ShapeDesigner: https://gitlab.inria.fr/jdusart/shexjapp



Try it with RDFShape: https://tinyurl.com/y8pjcbyf

Shapes from data: RDFShape

RDFShape/Wikishape implement a basic prototype to derive Shapes from RDF data



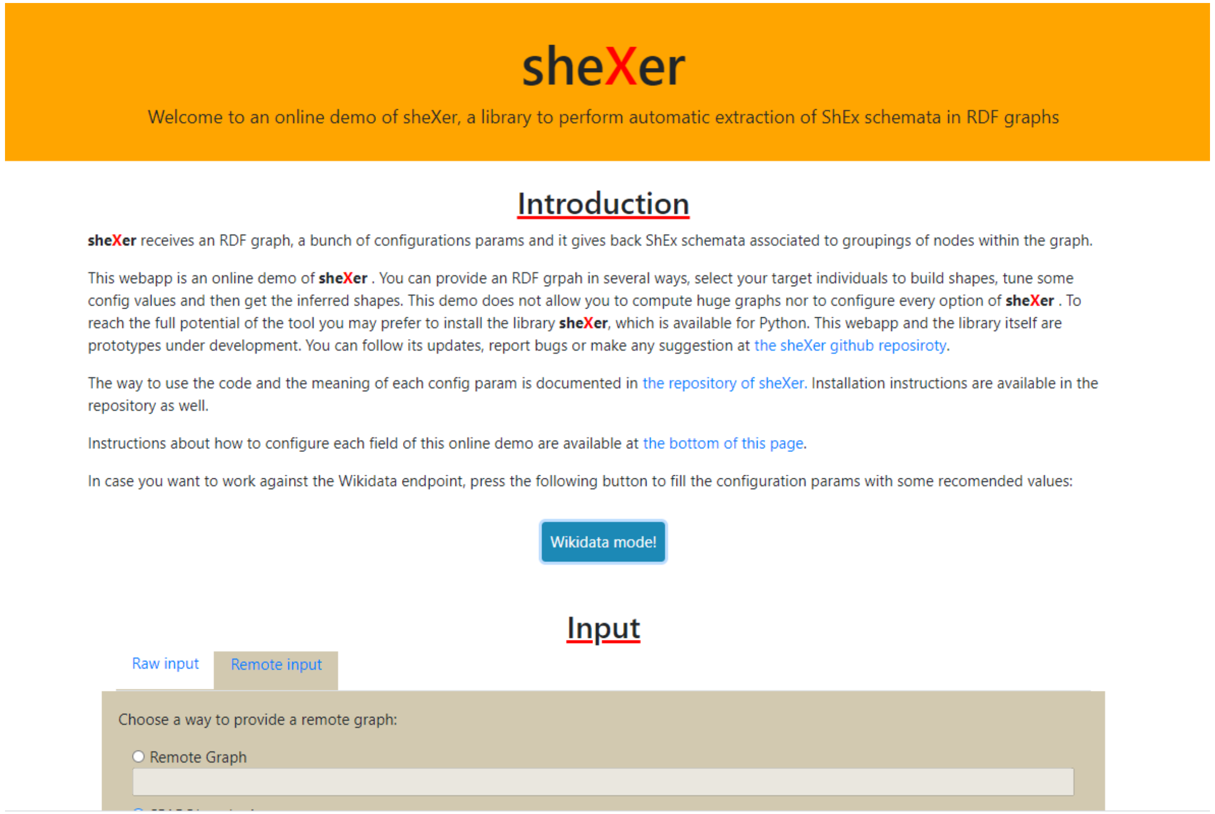
Shape Expression generated for `wd:Q51613194`

Shapes from data: sheXer

sheXer: http://shexer.weso.es/

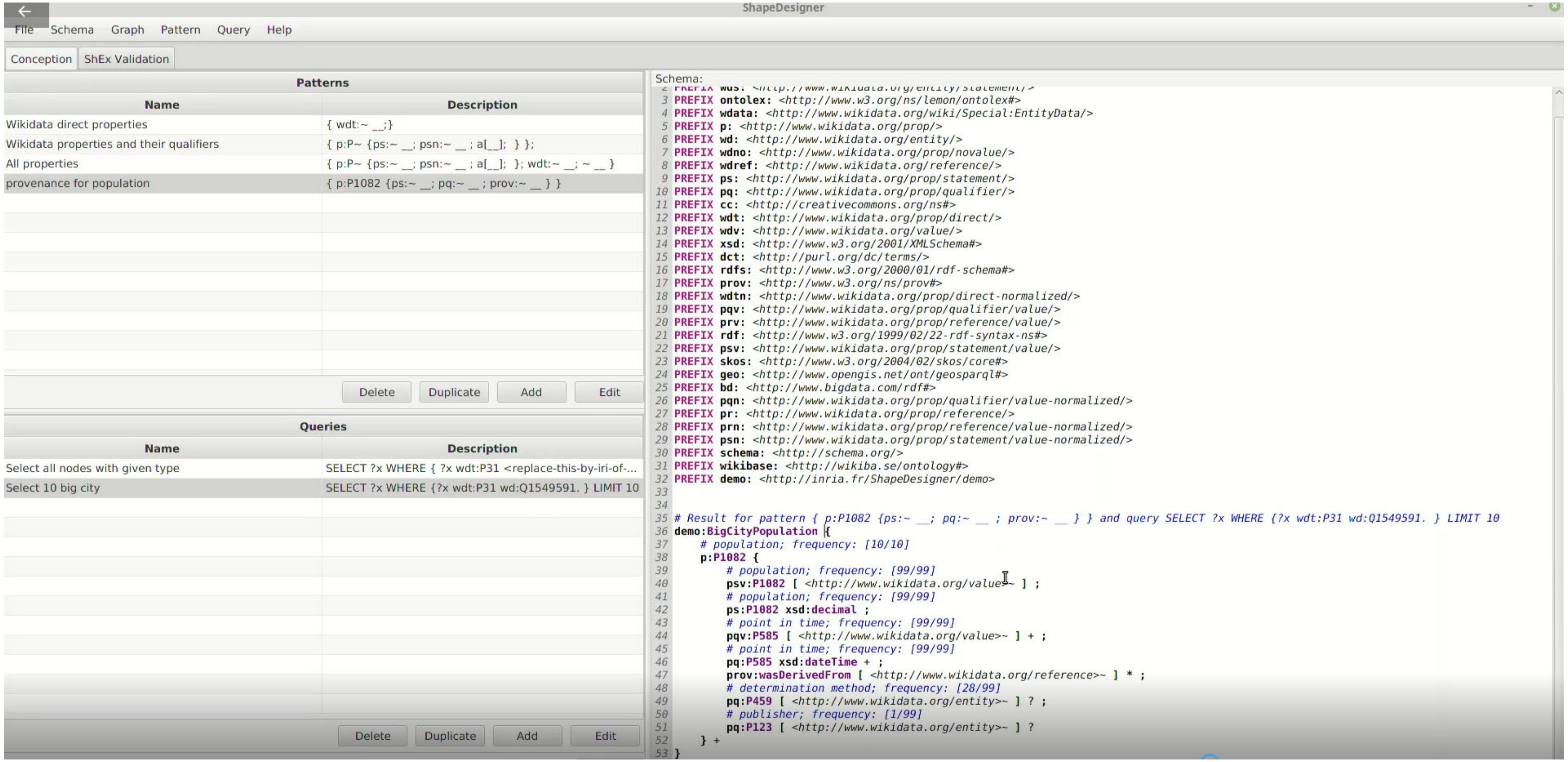
Implemented in Python

Configuration options



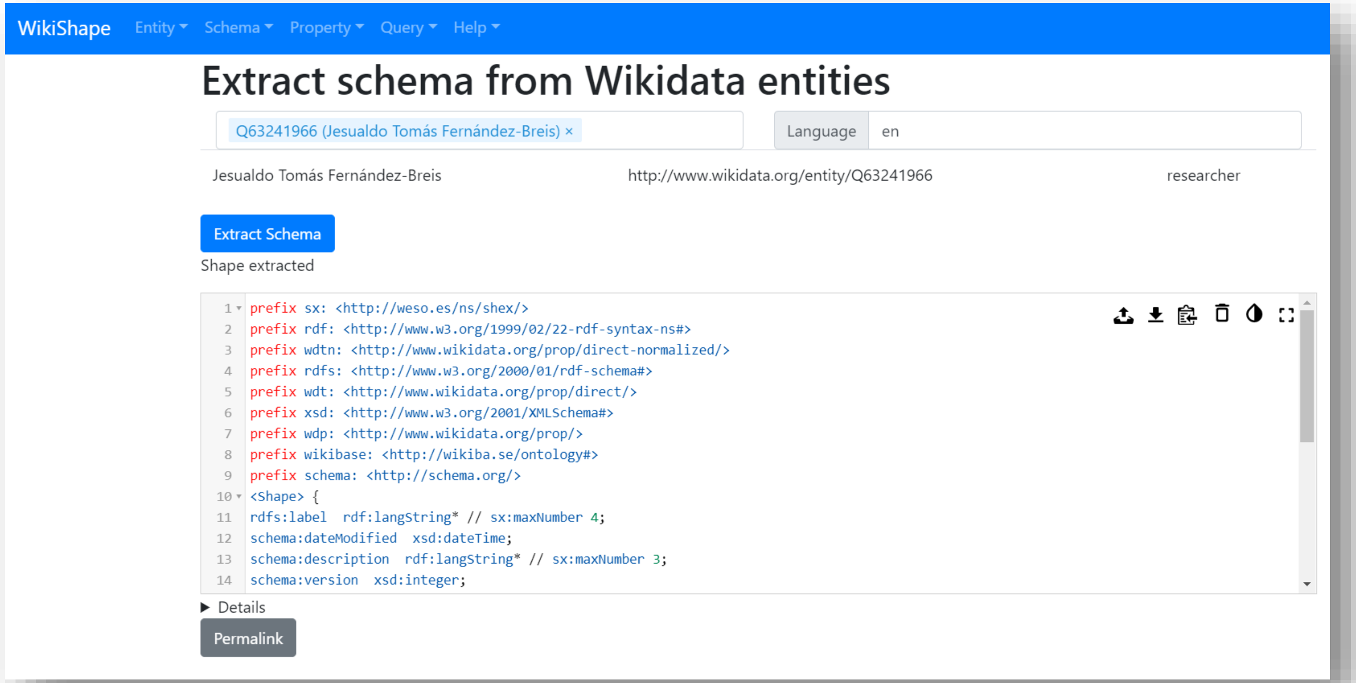
Shapes from data: ShapeDesigner

https://gitlab.inria.fr/jdusart/shexjapp



Shapes from RDF data

RDFShape allows to infer basic shapes automatically



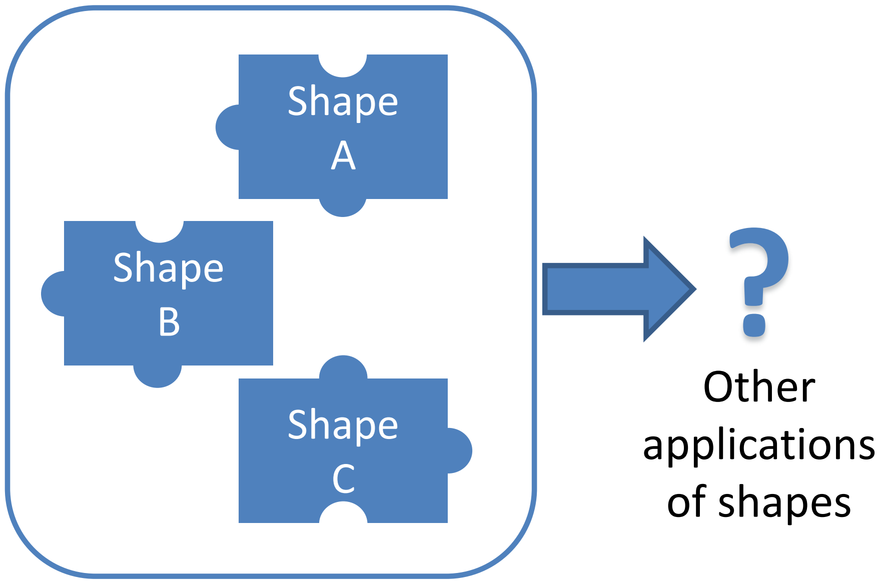
Other applications of Shapes

UIs and shapes

Generating code from Shapes

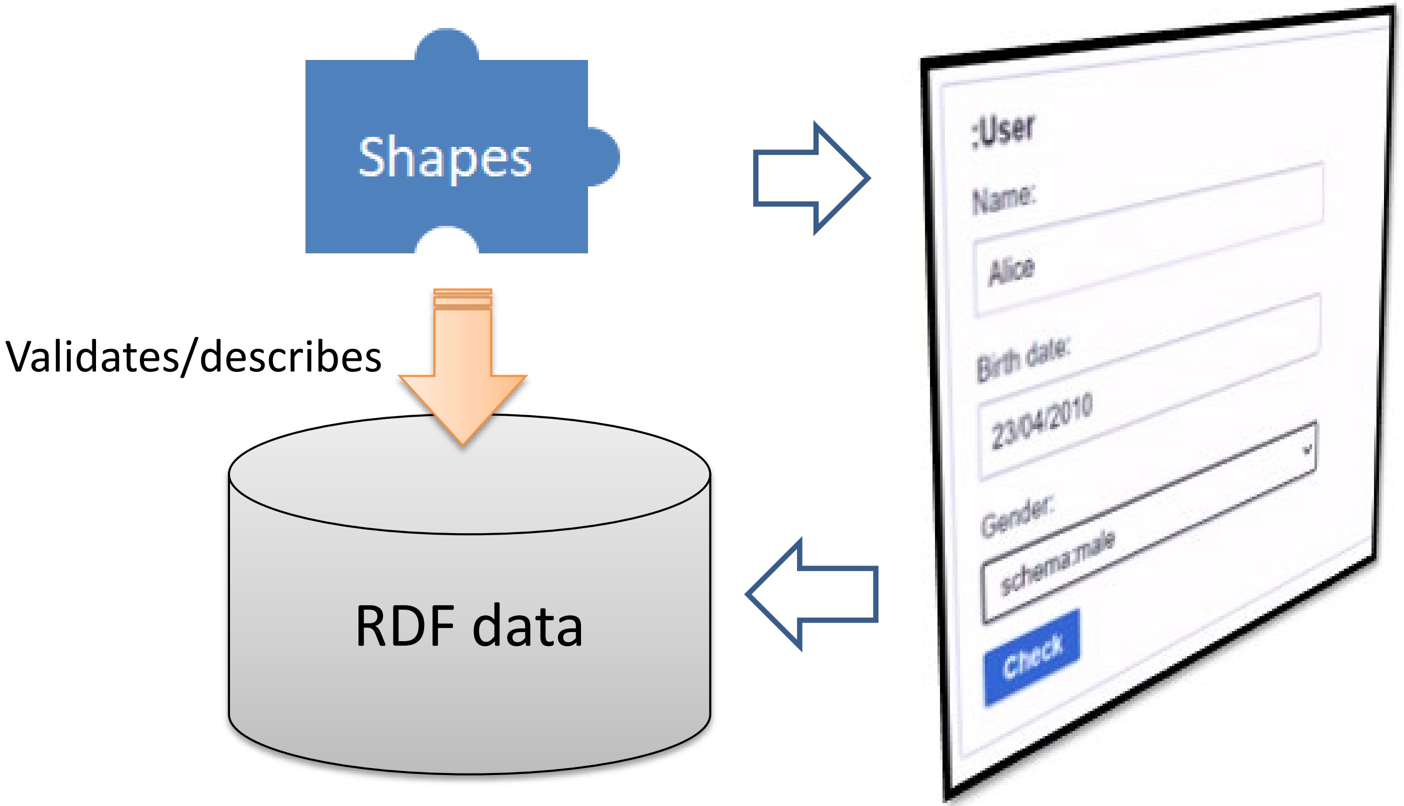
Shapes and rules

Generate subsettings



UIs and shapes

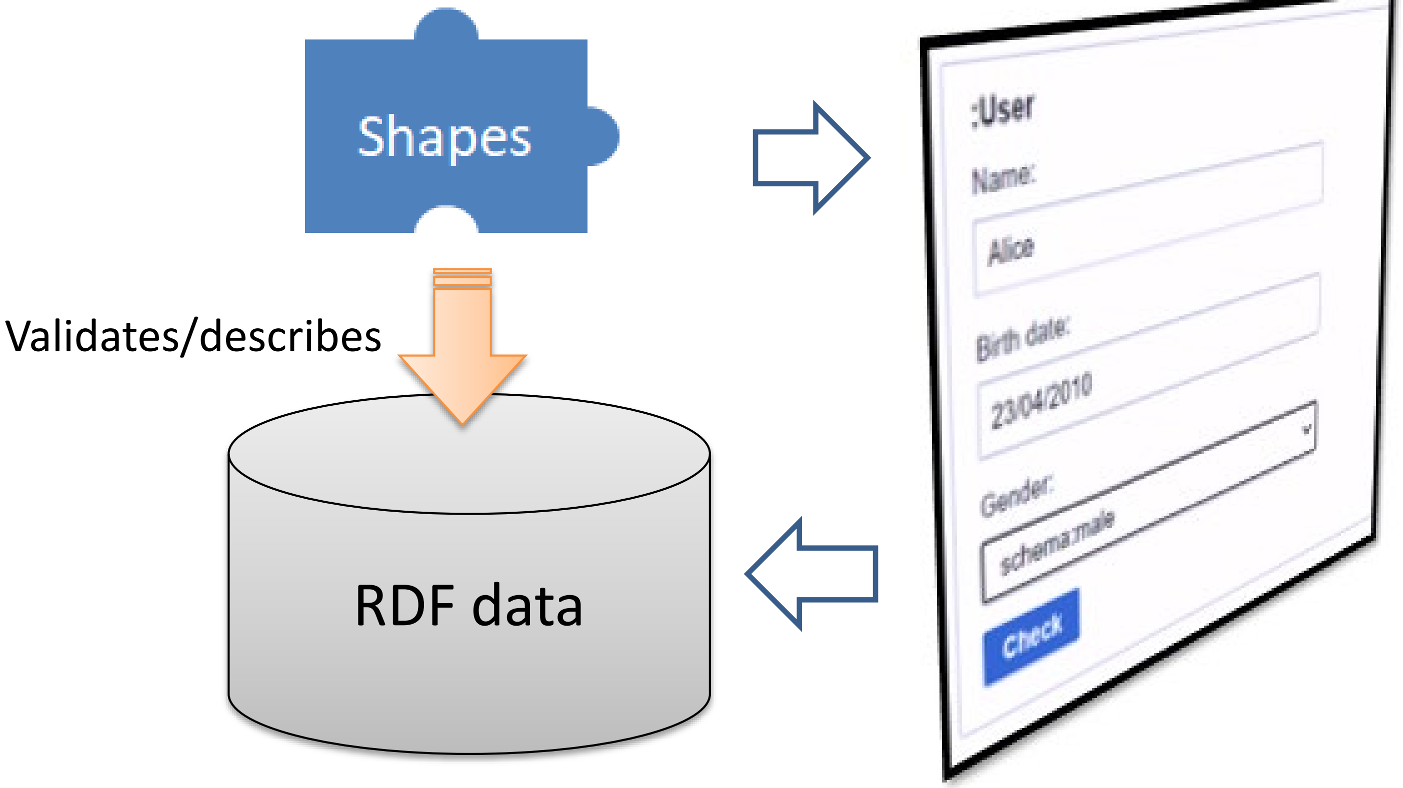
Shapes can provide hints to generate user interfaces/forms



UI and shapes: ShapeForms

ShapeForms

```
prefix schema: <http://schema.org/>
prefix : <http://example.org/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix ui: <http://www.w3.org/ns/ui#>

start = @:User

:User {
 schema:name  xsd:string       // ui:label "name" ;
 schema:birthDate xsd:dateTime // ui:label "Birth date" ;
 schema:gender [ schema:Male schema:Female ] // ui:label "Gender" ;
}
```



UIs and Shapes: ShapePath and ShapeForms

ShEx Path can be used to point to parts of a ShEx schema

https://shexspec.github.io/spec/ShExPath

ShEx generated forms demo based on UI ontology:

https://ericprud.github.io/shex-form/?manifestURL=examples/manifest.json

ShapeForms

https://github.com/weso/shapeForms

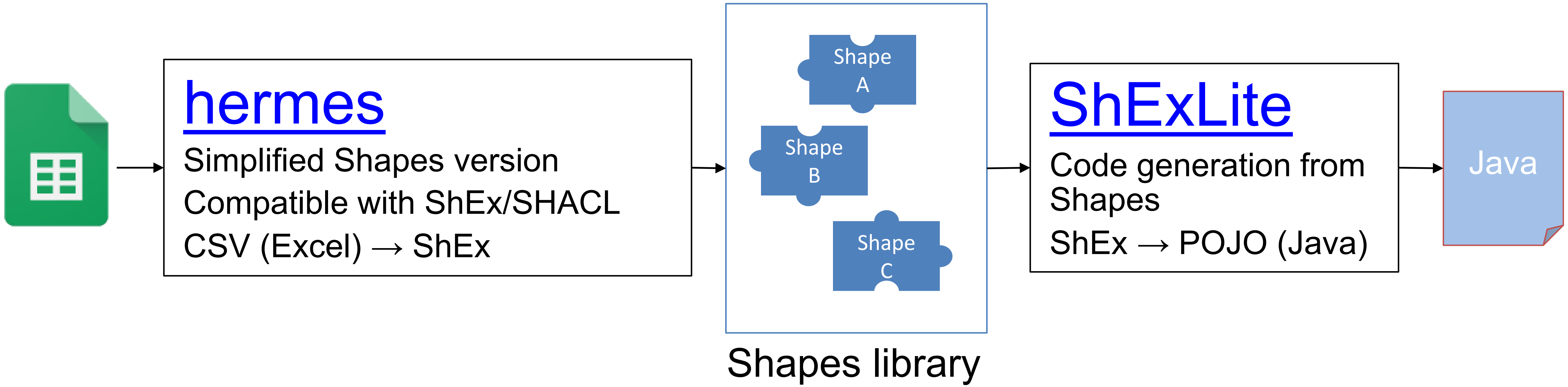
Generating code from shapes

Generate domain model from shapes

Entities (pseudo-shapes) defined with Excel (Google spreadsheets)

Shapes generation from those templates

Java code generation (POJOs) from those shapes

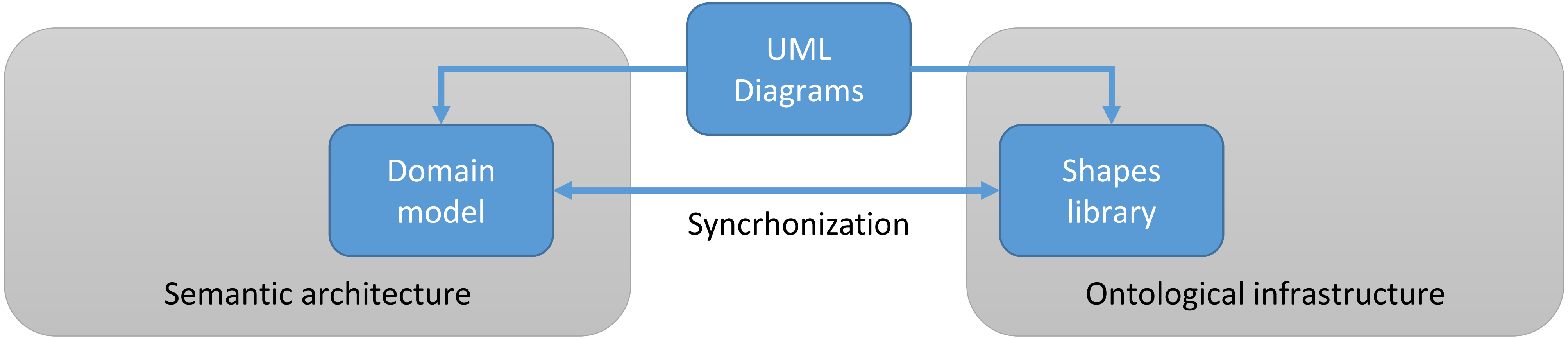


Generating code from shapes

Domain model based on Shapes

Clean architecture pattern

- Domain model as central element
- Simple classes (POJO): Plain Old Java Objects
- Shapes synchronization
- Application logic and services based on domain model

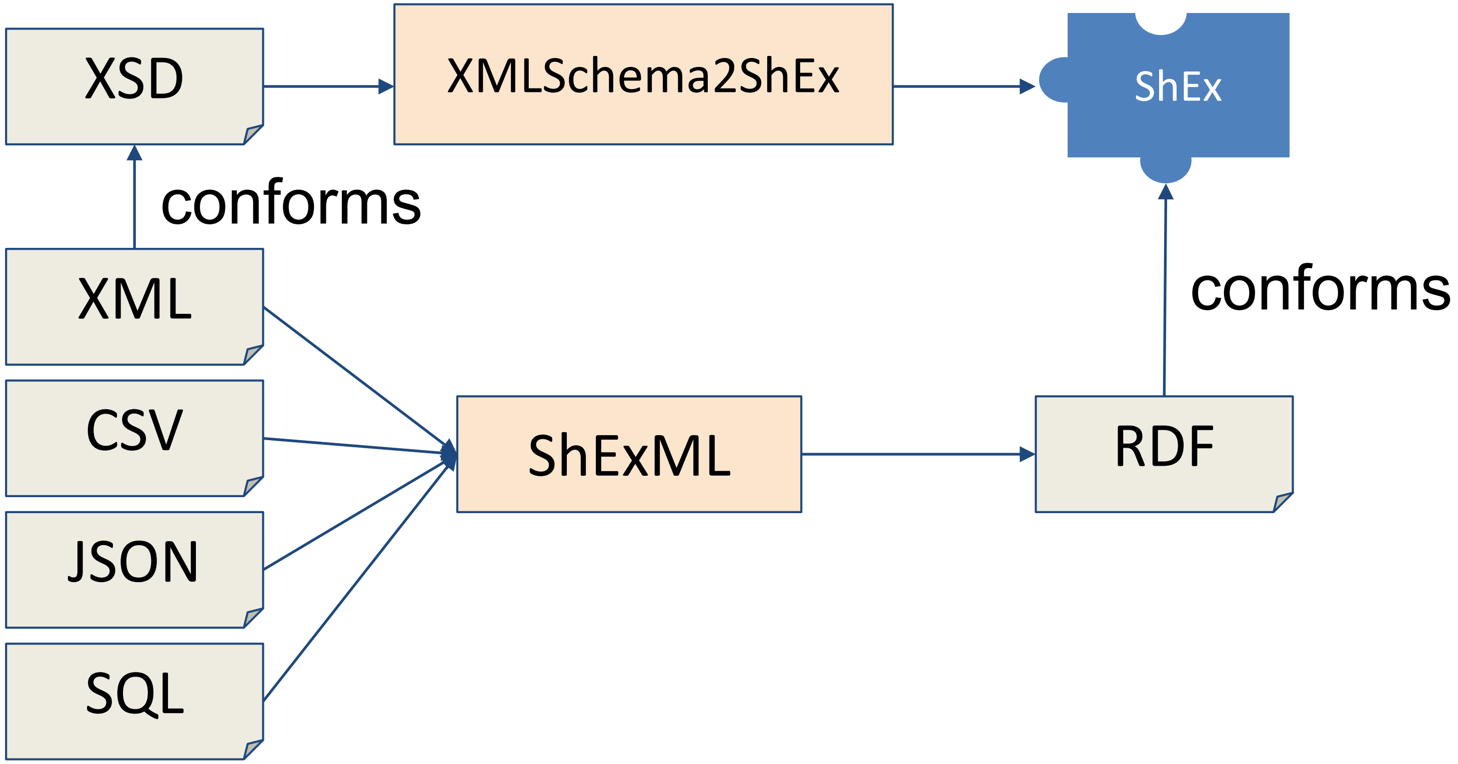


Shapes for data integration

XMLSchema2ShEx: Convert XML Schemas to shapes

ShExML: Domain specific language to convert data to RDF

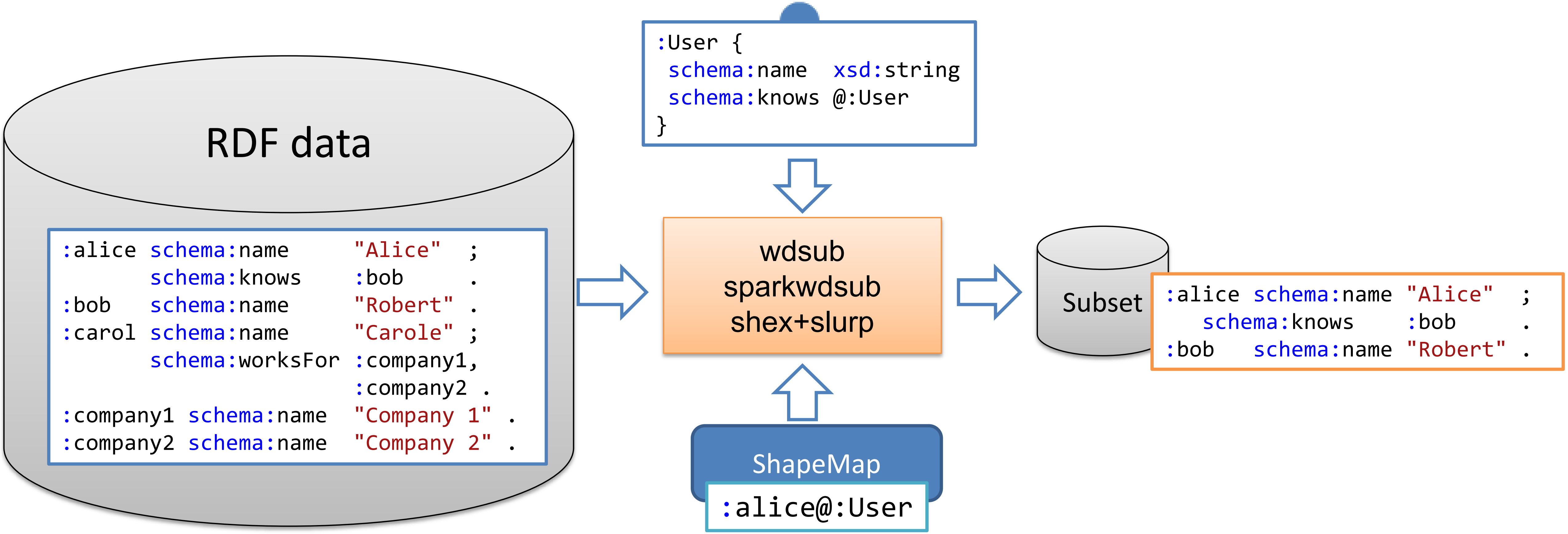
Input formats: CSV, XML, JSON, SQL



Subsetting based on Shapes

Generate subsets from ShEx

Slurp option: when validating, collect the affected nodes/triples



WShEx

ShEx extension for Wikibase graphs

Built-in support for qualifiers/references, data values

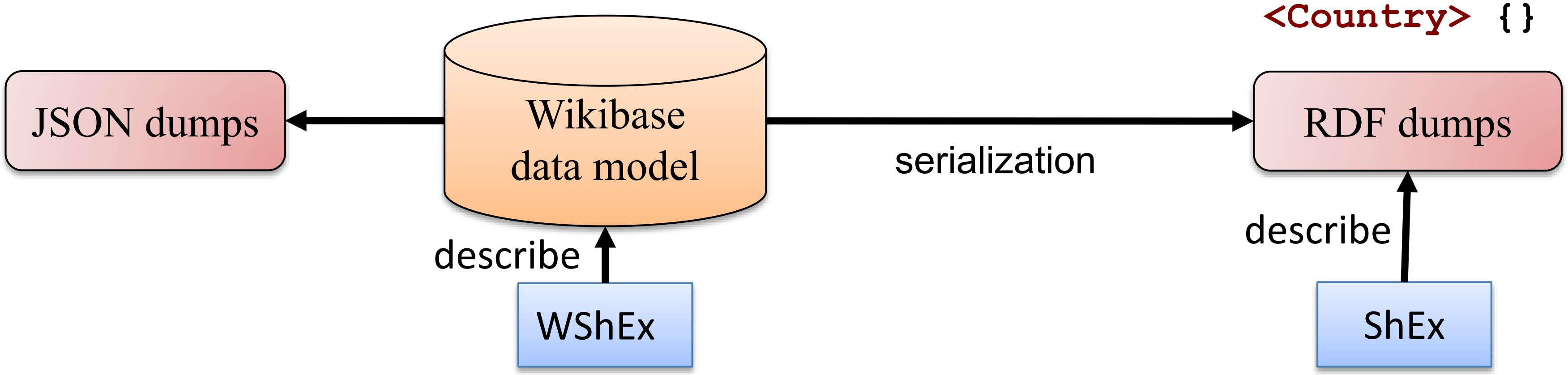
Work in progress

WShEx

```
<Researcher> {
 <birthPlace>        @<Place> ;
 <awarded>           @<Award> {{
    <togetherWith> @<Researcher> *
   }} *
}
<Place> {
 <country>           @<Country>
}
<Country> {}
```

ShEx

```
<Researcher> {
 wdt:birthPlace @<Place> ;
 p:birthPlace    {
    ps:birthPlace   @<Place>
 } ;
 wdt:awarded          @<Awarded> ;
 p:awarded {
   ps:awarded         @<Awarded> ;
   pq:togetherWith @<Researcher> *;
 } *
}
<Place> {
 wdt:country          @<Country> ;
 p:country {
  ps:country          @<Country> }
}
<Country> {}
```



Shapes ecosystems

- Wikidata provides a whole ShEx ecosystem
- Entity schemas can evolve and relate between each other
  - Directory: https://www.wikidata.org/wiki/Wikidata:Database_reports/EntitySchema_directory
  - Different schemas for the same entities?
    - Some schemas stress some aspects while others stress others
  - Evolution of schemas
  - Searching entity schemas

End of presentation