CITS3005 Knowledge Representation

Lecture 8: Knowledge Graph Schema

Tim French

The University of Western Australia

2023

Knowledge Graph Schemas

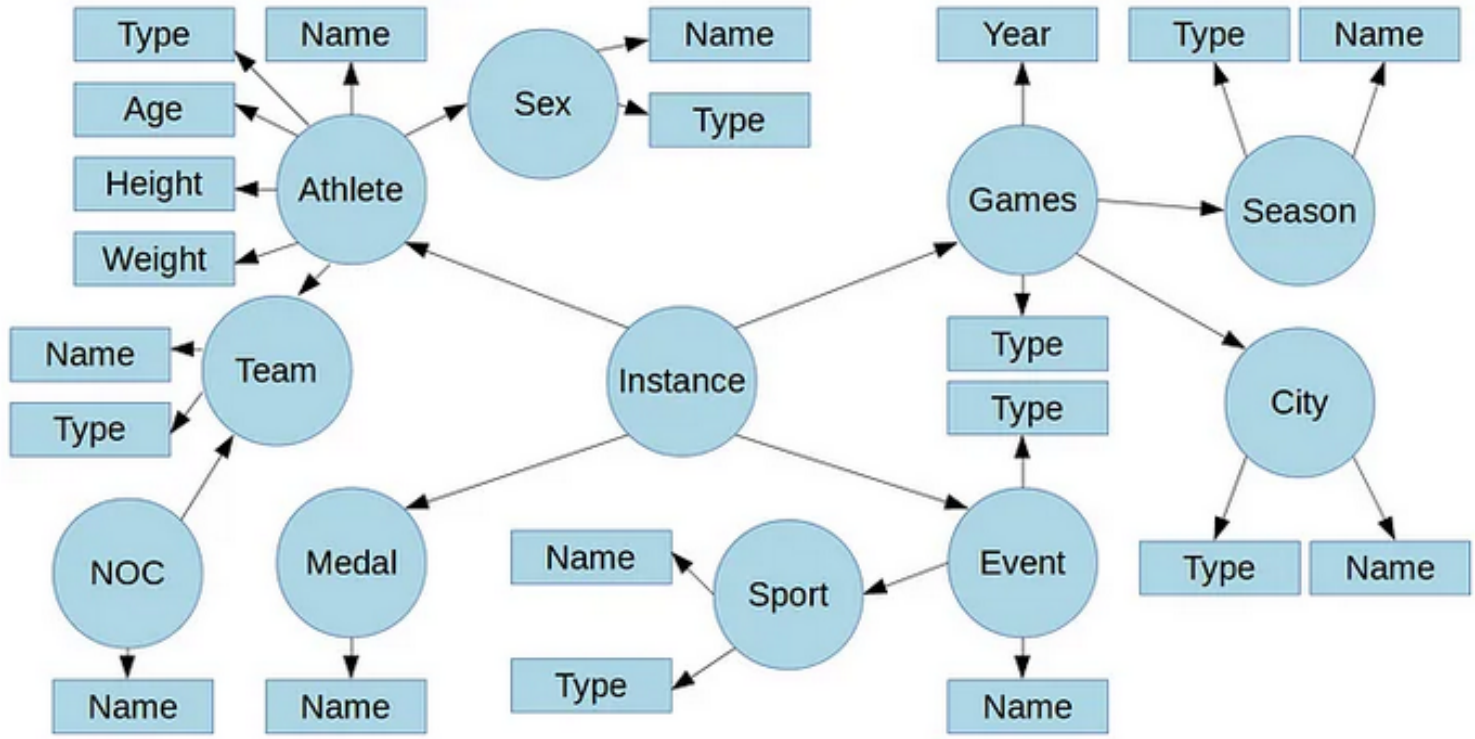
In this lecture we will look into the creation and use of knowledge graphs in some more detail, particularly focusing on the creation of semantic value through schema and constraints.
In particular, we will:

- Review creation querying and editing of RDF knowledge graphs using rdflib in python.
- Consider the problem of identity and context in knowledge.
- Consider *Semantic Schema* and the specification of knowledge under the *open world assumption*
- Use the extension RDFS for specifying semantic schema.
- Consider *Validating Schema* for specifying and checking constraints over knowledge graphs.
- Use SHACL to specify shapes and validate graphs.

This lecture is based on Chapter 3 of *Knowledge Graphs* by Hogan et al.

Example Knowledge Graph

Consider the following knowledge graph taken from `https://medium.com/wallscope/creating-linked-data-31c7dd479a9e` for representing all information taken from the modern Olympics.



The extraction and development process for this is described in the linked article.

Prefixes, URIs and Identity

The Olympics knowledge uses a range of prefixes to represent the names of concepts:

```
@prefix ns0:    <http://wallscope.co.uk/resource/olympics/team/> .
@prefix ns1:    <http://dbpedia.org/ontology/> .
@prefix ns2:    <http://wallscope.co.uk/resource/olympics/athlete/> .
@prefix ns3:    <http://xmlns.com/foaf/0.1/> .
@prefix ns4:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ns5:    <http://www.w3.org/2001/XMLSchema#> .
@prefix ns6:    <http://wallscope.co.uk/resource/olympics/gender/> .
@prefix ns7:    <http://wallscope.co.uk/resource/olympics/city/> .
@prefix ns8:    <http://wallscope.co.uk/resource/olympics/event/> .
@prefix ns9:    <http://wallscope.co.uk/resource/olympics/sport/> .
@prefix ns10:   <http://dbpedia.org/property/> .
@prefix ns11:   <http://wallscope.co.uk/resource/olympics/games/1896/> .
@prefix ns12:   <http://wallscope.co.uk/resource/olympics/season/> .
@prefix ns13:   <http://wallscope.co.uk/ontology/olympics/> .
@prefix ns14:   <http://wallscope.co.uk/resource/olympics/games/1900/> .
@prefix ns15:   <http://wallscope.co.uk/resource/olympics/games/1904/> .
@prefix ns16:   <http://wallscope.co.uk/resource/olympics/games/1906/> .
@prefix ns17:   <http://wallscope.co.uk/resource/olympics/games/1908/> .
@prefix ns18:   <http://wallscope.co.uk/resource/olympics/games/1912/> .
```

These are *Uniform Resource Identifiers* (URIs) which are essentially URLs but without the requirement to be network-accessible.
Uniformity is a powerful tool for reusing and communicating concepts.

Some common URIs

There are several open vocabularies describing common concepts:

1. RDF/RDFS (Resource Description Framework): basic definitions for types, subclasses, properties etc.
2. XSD (XML Schema Datatypes): Common datatypes such as integer, date, list, that can be specified in XML.
3. FOAF (Friend of a Friend): Concepts for describing people and their contact details.
4. SKOS (Simple Knowledge Organisation Schema): concepts describing knowledge mappings.
5. GeoNames: Geographical concepts, aligned with a crowd-sourced data base.
6. Music Ontology: Concepts describing music, artists, recordings etc.

Summary of Terms

This vocabulary defines 54 classes and 153 properties.

Term Name	Type	Definition
Activity	Class	activity
AnalogSignal	Class	analogue signal
Arrangement	Class	arrangement
AudioFile	Class	audio file
CD	Class	CD
Composition	Class	composition
CorporateBody	Class	corporate body
DAT	Class	DAT
DCC	Class	DCC
DVDA	Class	DVDA
DigitalSignal	Class	digital signal
ED2K	Class	ED2K
Festival	Class	Festival
Genre	Class	Genre
Instrument	Class	Instrument

Interacting with a knowledge graph

The Graph class in rdflib can be used to load sets of triples in various formats:

```
from rdflib import Graph


g = Graph()
g.parse("http://www.w3.org/People/Berners-Lee/card")
g.serialize(destination="tbl.ttl")
```

There are a variety of ways to represent the RDF:

RDF Format	Keyword	Notes
Turtle	turtle, ttl or turtle2	turtle2 is just turtle with more spacing & linebreaks
RDF/XML	xml or pretty-xml	Was the default format, rdflib < 6.0.0
JSON-LD	json-ld	There are further options for compact syntax and other JSON-LD variants
N-Triples	ntriples, nt or nt11	nt11 is exactly like nt, only utf8 encoded
Notation-3	n3	N3 is a superset of Turtle that also caters for rules and a few other things
Trig	trig	Turtle-like format for RDF triples + context (RDF quads) and thus multiple graphs
Trix	trix	RDF/XML-like format for RDF quads
N-Quads	nquads	N-Triples-like format for RDF quads

See `https://ontola.io/blog/rdf-serialization-formats` for a description of the different formats.

Creating and removing triples in rdflib

In RDF the *atoms* are URIs, blank nodes or literals:

```
from rdflib import URIRef, BNode, Literal
from rdflib.namespace import FOAF, RDF

bob = URIRef("http://example.org/people/Bob")
linda = BNode()  # a GUID is generated

name = Literal("Bob")  # passing a string
age = Literal(24)  # passing a python int
height = Literal(76.5)  # passing a python float

g.add((bob, RDF.type, FOAF.Person))
g.add((bob, FOAF.name, name))
g.add((bob, FOAF.age, age))
g.add((bob, FOAF.knows, linda))
g.add((linda, RDF.type, FOAF.Person))
g.add((linda, FOAF.name, Literal("Linda")))

print(g.serialize())
```

Triples can also be removed using `g.remove(..)`

The Graph Object

In rdflib, the Graph object is iterable, and you can check membership

```
for s, p, o in someGraph:
  if not (s, p, o) in someGraph:
  raise Exception("Iterator / Container Protocols are Broken!!")
```

and basic set operations are implemented:

operation	effect
G1 + G2	return new graph with union (triples on both)
G1 += G2	in place union / addition
G1 - G2	return new graph with difference (triples in G1, not in G2)
G1 -= G2	in place difference / subtraction
G1 & G2	intersection (triples in both graphs)
G1 ^ G2	xor (triples in either G1 or G2, but not in both)

There are also methods for extracting all triples, subjects, predicates etc.

SPARQL queries in rdflib

SPARQL (SPARQL Protocol and RDF Query Language) is the query language we use to shape and return linked data from a triplestore.
SPARQL queries contain triple patterns, much like the data itself, which utilise the relationships to quickly navigate any linked data. This language is common for all linked data so queries can traverse across multiple RDF databases at once.

```python
from rdflib import Graph

# Create a Graph, add in some test data
g = Graph()
g.parse(
    data="""
        <x:> a <c:> .
        <y:> a <c:> .
    """,
    format="turtle"
)

# Select all the things (s) that are of type (rdf:type) c:
qres = g.query("""SELECT ?s WHERE { ?s a <c:> }""")

for row in qres:
    print(f"{row.s}")
# prints:
# x:
# y:

# Add in a new triple using SPARQL UPDATE
g.update("""INSERT DATA { <z:> a <c:> }""")

# Select all the things (s) that are of type (rdf:type) c:
qres = g.query("""SELECT ?s WHERE { ?s a <c:> }""")

print("After update:")
for row in qres:
    print(f"{row.s}")
# prints:
# x:
# y:
# z:

# Change type of <y:> from <c:> to <d:>
g.update("""
         DELETE { <y:> a <c:> }
         INSERT { <y:> a <d:> }
         WHERE { <y:> a <c:> }
         """)
print("After second update:")
qres = g.query("""SELECT ?s ?o WHERE { ?s a ?o }""")
for row in qres:
    print(f"{row.s} a {row.o}")
# prints:
# x: a c:
# z: a c:
# y: a d:
```

SPARQL Syntax

SPARQL syntax is similar to SQL:
`SELECT` *vars* `WHERE` *pattern* `FILTER` *condition* `GROUP/SORT`
The main difference is that the pattern uses subgraph matching:

```
PREFIX walls: <http://wallscope.co.uk/ontology/olympics/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?name (COUNT(?name) As ?noOfMedals)
WHERE {
  ?instance walls:athlete ?athlete ;
            walls:medal   ?medal .

  ?athlete  rdfs:label    ?name .
}
GROUP BY ?name
ORDER BY DESC(?noOfMedals)
```

Find all athletes with at least one medal.
Note that the subgraph pattern is presented as a set of branching nodes, with common variables.

SPARQL Examples

Aggregation:

```
PREFIX walls: <http://wallscope.co.uk/ontology/olympics/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?countryCode
       (AVG(?height) As ?avgHeight)
       (AVG(?weight) As ?avgWeight)
WHERE {
 ?noc dbo:ground ?team ;
      rdfs:label ?countryCode .

 ?athlete rdf:type   foaf:Person ;
          dbo:team   ?team ;
          dbo:height ?height ;
          dbo:weight ?weight .
}
GROUP BY ?countryCode
ORDER BY DESC(?avgHeight)
```

find the average height and weight of each country's atheletes

Filtration:

```
PREFIX walls: <http://wallscope.co.uk/ontology/olympics/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/property/>

SELECT DISTINCT ?name ?cityName ?seasonName
WHERE {
 ?instance walls:games   ?games ;
           walls:athlete ?athlete .

 ?games dbp:location ?city ;
        walls:season ?season .

 ?city rdfs:label ?cityName .

 ?season rdfs:label ?seasonName .

 ?athlete rdfs:label ?name .

 Filter (REGEX(lcase(?name),"louis.*"))
}
```

find every athletes with "louis" in their name

SPARQL Examples

Nested Queries:

```
PREFIX walls: <http://wallscope.co.uk/ontology/olympics/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX noc: <http://wallscope.co.uk/resource/olympics/NOC/>

SELECT ?genderName (COUNT(?athlete) AS ?count)
WHERE {
  ?instance walls:games   ?games ;
            walls:athlete ?athlete .

  ?games dbp:location ?city .

  ?athlete foaf:gender ?gender .

  ?gender rdfs:label ?genderName .

  {
    SELECT DISTINCT ?city
    WHERE {
    ?instance walls:games   ?games ;
              walls:athlete ?athlete .

    ?athlete dbo:team ?team .

    noc:SCG dbo:ground ?team .

    ?games dbp:location ?city .
    }
  }
}
GROUP BY ?genderName
```

count the number of atheletes by gender, for a given set of cities

Services:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

SELECT ?sportName ?teamSize
WHERE {
 ?sport rdf:type   dbo:Sport ;
        rdfs:label ?sportName .

 SERVICE <http://dbpedia.org/sparql> {
 ?dbsport rdfs:label   ?sportName ;
          dbo:teamSize ?teamSize .
 }
}
```

find the team sizes from dbpedia

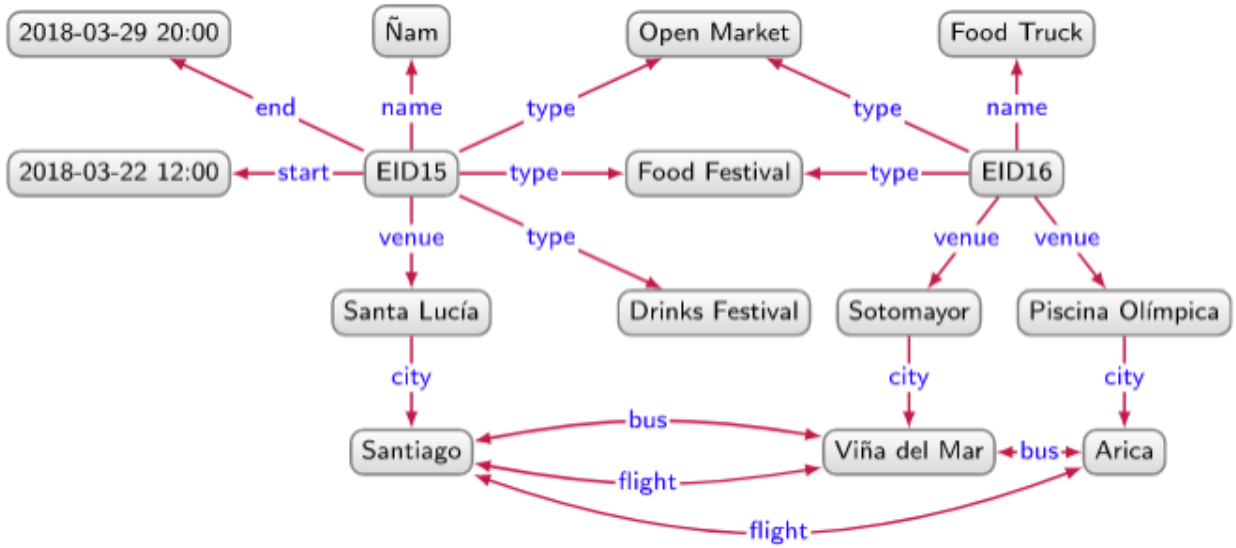
Schemas

One of the benefits of modelling data as graphs is the option to postpone the definition of a schema.
Schemata can be used to prescribe a high-level structure and/or semantics that the graph follows or should follow.
This provides critical context for the data in the graph, and this context provides meaning (semantics) for the data.
We discuss two types of graph schemata:

- *Semantic schema* provide context of classes for entities and properties, allowing new relations (, *knowledge*) to be deduced through a process of entailment. This is realised through an extension to RDF called RDFS.
- *Validating schema* provide constraints for what should be considered valid data. They often take the form of integrity constraints (e.g every licensed car should have a registered owner). and are specified via *shapes*. This is realised using the Shape Constraint Language, ShaCL.

Semantic Schema

A semantic schema allows for defining the meaning of high-level terms which facilitates reasoning using those terms.
In the figure we can see some natural groupings of nodes based on the types of entities to which they refer: Event, City, etc.



Aside from classes, we may also wish to define the semantics of edge labels, aka properties. We may consider that the properties city and venue are sub-properties of a more general property location. We may also consider, for example, that bus and flight are both sub-properties of a more general property connects to.

RDF Schema

A prominent standard for defining a semantic schema for (RDF) graphs is the RDF Schema (RDFS) standard, which allows for defining sub-classes, sub-properties, domains, and ranges amongst the classes and properties used in an RDF graph.

Table 3.1: Definitions for sub-class, sub-property, domain and range

Feature	Definition	Condition	Example
SUB-CLASS	c —subc. of→ d	x —type→ c implies x —type→ d	City —subc. of→ Place
SUB-PROPERTY	p —subp. of→ q	x —p→ y implies x —q→ y	venue —subp. of→ location
DOMAIN	p —domain→ c	x —p→ y implies x —type→ c	venue —domain→ Event
RANGE	p —range→ c	x —p→ y implies y —type→ c	venue —range→ Venue

The definitions in RDFS come with the RDF notion of type to let types and properties be *inferred* from other relations. This allows us to define concepts in terms of other concepts.
RDFS defines

- predefined triples (in RDF, compatible with rdflib).
- *rules* for how some triples cab *entail* additional triples.

RDFS Syntax

`rdfs:Class` represents a type of similar resources, which are the individuals in the class.

- class membership is expressed by the `rdf:type` property: `<RDF individual> rdf:type <RDFS class>` or `<RDF individual> a <RDFS class>`
- an individual can belong to several classes
- e.g., dbpedia:Person, schema:Person, foaf:Person.

`rdfs:subClassOf` is the *property* that states membership of one class necessitates membership of another class:

- allows you to describe properties at different levels of abstraction, and promotes reuse.
- this allows improved completeness, and can help managing the merging of different vocabularies.
- `dbpedia:Politician rdfs:subClassOf dbpedia:Person`

Entailment and Syllogism

These simple properties can represent reasoning!

From *all men are mortal* and *Socrates is a man* we can deduce *Socrates is mortal*.

This is one of the 16 rules built into RDFS:

From

`ex:Man rdfs:SubclassOf ex:Mortal`

and

`ex:socrates rdf:type ex:Man`,

Infer

`ex:socrates rdf:type ex:Mortal`.

ext1	uuu rdfs:domain vvv . vvv rdfs:subClassOf zzz .	uuu rdfs:domain zzz .
ext2	uuu rdfs:range vvv . vvv rdfs:subClassOf zzz .	uuu rdfs:range zzz .
ext3	uuu rdfs:domain vvv . www rdfs:subPropertyOf uuu .	www rdfs:domain vvv .
ext4	uuu rdfs:range vvv . www rdfs:subPropertyOf uuu .	www rdfs:range vvv .
ext5	rdf:type rdfs:subPropertyOf www . www rdfs:domain vvv .	rdfs:Resource rdfs:subClassOf vvv .
ext6	rdfs:subClassOf rdfs:subPropertyOf www . www rdfs:domain vvv .	rdfs:Class rdfs:subClassOf vvv .
ext7	rdfs:subPropertyOf rdfs:subPropertyOf www . www rdfs:domain vvv .	rdf:Property rdfs:subClassOf vvv .
ext8	rdfs:subClassOf rdfs:subPropertyOf www . www rdfs:range vvv .	rdfs:Class rdfs:subClassOf vvv .
ext9	rdfs:subPropertyOf rdfs:subPropertyOf www . www rdfs:range vvv .	rdf:Property rdfs:subClassOf vvv .

Example: Transitivity

RDFS subclass and subproperties should be *transitive*. A subclass of a subclass of A should, itself, be a subclass of A.
That is, the triples:
`?c1 rdfs:subClassOf ?c2.` and
`?c2 rdfs:subClassOf ?c3.`
entail `?c1 rdfs:subClassOf ?c3.`
This *entailment* could be realised using the SPARQL query:

```
...prefixes....
INSERT {?c1 rdfs:subClassOf ?c3. }
WHERE { ?c1 rdfs:subClassOf ?c2.  ?c2 rdfs:subClassOf ?c3.}
```

However, we would like an entailment engine to perform this inference automatically.

Domain and Range of properties

The subjects and objects that occur in triples along with some property belong to certain classes.
For example, given the triple `<subject> ex:presidentOf <object> .`
we may know that:

- the `<subject>` has `rdf:type dbpedia:President`
- the `<object>` has `rdf:type dbpedia:Country`

This is part of the semantics of *ex:president* in the context of the `ex:vocabulary` (note in another context, say, the president of a club, this would not be valid.)
We can express this property as follows:

```
ex:presidentOf rdfs:domain dbpedia:President .
ex:presidentOf rdfs:range dbpedia:Country .
```

Utilities and limitations of RDFS

Utility Properties in RDFS

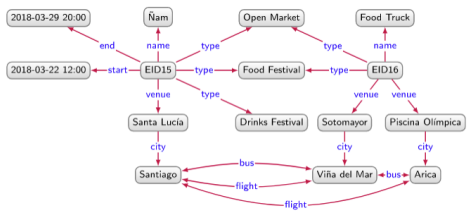
- `rdfs:label` a human-readable label
- `rdfs:comment` a human-readable comment
- `rdfs:seeAlso` reference to further information
- `rdfs:isDefinedBy` a human-readable definition (is a `rdfs:subPropertyOf` `rdfs:seeAlso`)

Limitations: RDFS cannot express:

- my ancestors' ancestors are also my ancestors
- a Person has a unique birth number
- a Person has exactly one father
- a SoccerTeam has 11 players, but a BasketballTeam has 5
- classes with different URIs actually represent the same class
- properties with different URIs are actually the same
- two individuals with different URIs are actually different
- a class is a combination (union or intersection) of

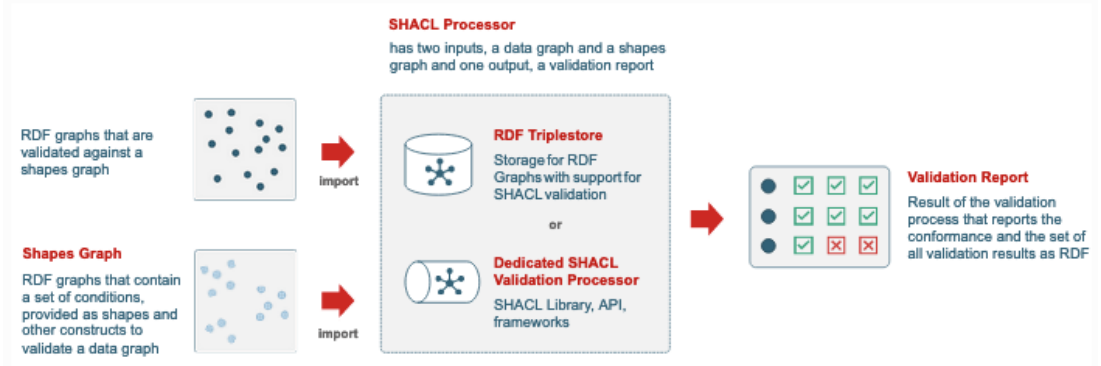
Open World Assumption

From the graph of Figure 2.1, we cannot assume that there is no flight between Vina del Mar and Arica.
In contrast, if the *Closed World Assumption* (CWA) were adopted (e.g. database systems) it would be assumed that the data graph is a complete description of the world, thus allowing to assert with certainty that no flight exists between the two cities.
Systems that do not adopt the CWA are said to adopt the *Open World Assumption* (OWA). RDFS is designed to operate with the OWA. It specifies logical closures of entailments, but never asserts that something is not known.
However, there are many cases where we would like the knowledge graph to be a (relatively) complete description of the modelled world.



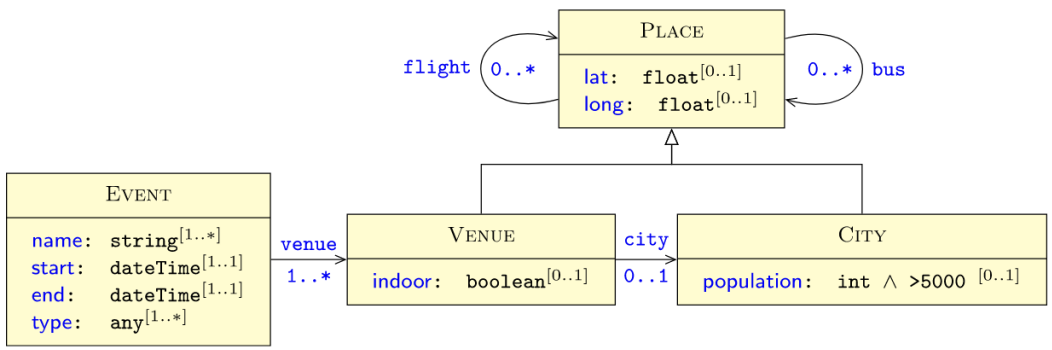
Validating Schema

While RDFS allows us to take some minimal description of a world and extend it through entailment, *Validating Schema* specify the minimum requirements that a *valid* knowledge graph should achieve.
For example, we may wish to ensure that all events have at least a name, a venue, a start date, and an end date. Or we may wish to ensure that the city of an event is stated to be a city (rather than just inferring that it is a city).
We can define such constraints in a validating schema and validate the data graph with respect to the resulting schema, listing constraint violations (if any).



Shapes

The standard approach to representing schema constraints is to use *shapes*.
Shapes are similar to the graph patterns used by SPARQL: shapes target a set of nodes in a data graph and specifies constraints on those nodes. The shape's target can be defined in many ways, such as targeting all instances of a class, the domain or range of a property, the result of a query, etc. Constraints can then be defined on the targeted nodes, such as to restrict the number or types of values taken on a given property, the shapes that such values must satisfy, etc.



A shapes graph is formed from a set of interrelated shapes. Shapes graphs can be depicted as UML-like class diagrams, with constraints on the properties indicating necessary membership.
(Note we can also specify upper bounds!)

Checking Shapes

Given a shape and a targeted node, we can check if the node conforms to that shape or not, which may require checking conformance of other nodes.

Conformance dependencies may also be recursive, where the conformance of Santiago to City requires that it conforms to Place, which requires that Viña del Mar and Arica conform to Place, and so on.

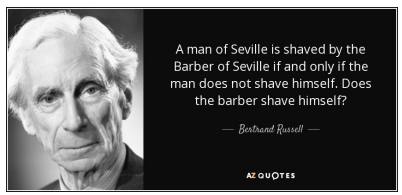
Conversely, EID16 does not conform to Event, as it does not have the start and end properties required by the example shapes graph.

When declaring shapes, the data modeller may not know in advance the entire set of properties that some nodes can have (now or in the future). An open shape allows the node to have additional properties not specified by the shape, while a closed shape does not.

Shape paradoxes

Practical languages for shapes often support additional Boolean features, such as conjunction (and), disjunction (or), and negation (not) of shapes. However, shapes languages that freely combine recursion and negation may lead to semantic problems, depending on how their semantics are defined.

To illustrate, consider the following case inspired by the Barber paradox (or Russel's Paradox), involving a shape `Barber` whose conforming nodes shave at least one node conforming to `Person` and (not `Barber`).
Now, given (only) `Bob shave Bob` with `Bob` conforming to `Person`, does `Bob` conform to `Barber`?
If `Bob` conforms to `Barber`, then `Bob` violates the constraint by not shaving at least one node conforming to `Person` and (not `Barber`).
If `Bob` does not conform to `Barber` – then `Bob` satisfies the `Barber` constraint by shaving such a node.



Shape Constraint Language (SHACL)

SHACL constrains the structure of RDF (and RDFS, OWL...) graphs. It is a W3C standard (from around 2018) used to validate:

- open and other KGs we want to reuse
- graphs resulting from user input
- the KGs we make ourselves

SHACL constraints are, themselves, written in RDF and a *shapes graph* is used to validate a *data graph*
SHACL validator for rdf implemented as PySHACL.

Syntax: Node Shapes

Node shapes specify constraints on focus nodes:
`<node_shape_URI> a sh:NodeShape`

- the focus nodes are often specified as
  `sh:targetClass <class_URI>`
- the node constraints apply to all instances of the target class
- alternatives:
  `sh:targetNode, sh targetSubjectsOf, sh:targetObjectsOf`
- constraints on each focus node itself:
  `sh:class <class_URI>` or
  `sh:datatype <datatype_URI>` or
  `sh:in ( ...list of URIs/values... )` or
  `sh:hasValue ...URI/value...` or
  `sh:nodeKind sh:IRI/sh:Literal/sh:BlankNode` or
  `sh:pattern <regular_expression>`.
- constraints on properties from the focus node either:
  by shape URI:
  `sh:property <property_shape_URI>`, or by anonymous node:
  `sh:property [a sh:PropertyShape ; sh:path <property_URI> ...]`

Syntax: Property Shapes

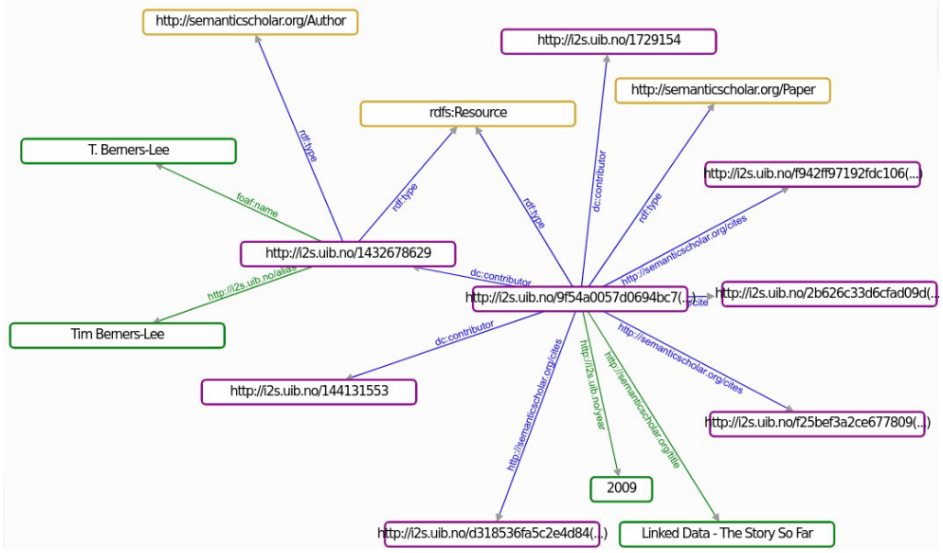
Property shapes specify constraints about the values that can be reached from a focus node by some path:

`<property_shape_URI> a sh:PropertyShape`

- the property is often specified as `sh:path <property_URI>`
- alternatively, SPARQL like property paths can be used
- the property constraints apply to all uses of the property path from the focus nodes, and all values reached by the property path from the focus nodes
- property constraints: `sh:minCount`, `sh:maxCount`, ...
- node constraints about the property value (the object resource or literal): `sh:class`, `sh:datatype`, `sh:nodeKind`, `sh:pattern`, ...

SPARQL path...	***...corresponds to SHACL path***
schema:name	schema:name
^schema:knows	[sh:inversePath schema:knows]
schema:knows / schema:name	(schema:knows schema:name)
schema:knows | schema:follows	[sh:alternativePath (schema:knows schema:follows)]
schema:knows?	[sh:zeroOrOnePath schema:knows]
schema:knows+	[sh:oneOrMorePath schema:knows]
schema:knows / schema:name*	([sh:zeroOrMorePath schema:knows] schema:name)

Example



Every paper is the subject of exactly 1 year property:

```
kg:MainPaperShape a sh:NodeShape ;
  sh:targetClass kg:MainPaper ;
  sh:property [
    sh:path kg:year ;
    sh:minCount 1 ;
    sh:maxCount 1
  ].
```

SHACL constraint structure

SHACL constrains *node shapes* and *property shapes*.
The node shapes are mostly collections of property shapes pertaining to the same class.

```
kg:MainPaperShape
  a sh:NodeShape ;
  sh:targetClass kg:MainPaper ;
  sh:property kg:SubjectShape .

kg:SubjectShape
  a sh:PropertyShape ;
  sh:path dcterm:subject ;
  sh:minCount 1 ;
  sh:or (
    [sh:class th:Theme]
    [sh:class ss:Topic] ) ;
  sh:nodeKind sh:IRI .
```

Validation

Reports the results applying a SHACL shapes graph to a data graph a `sh:ValidationReport` has three components:

- `sh:conforms` (either true or false)
- a `results_text` (from pySHACL)
- zero or more `sh:ValidationResults`

Result property	Explanation
sh:focusNode	The focus node that was being validated when the error happened.
sh:resultPath	The path from the focus node. This property is optional usually corresponds to the sh:path declaration of property shapes.
sh:value	The value that violated the constraint, when available .
sh:sourceShape	The shape that the focus node was validated against when the constraint was violated.
sh:source ConstraintComponent	The IRI that identifies the component that caused the violation.
sh:detail	May point to further details about the cause of the error. This property can be used for reporting errors in nested nested shapes.
sh:resultMessage	Textual details about the error. This message can be affected by the sh:message property.
sh:resultSeverity	A value which is equal to the sh:severity value of the shape that caused the violation error, if present. Otherwise the default value will be sh:Violation.

Validation with pySHACL

Programming pySHACL

```
# pip install pyshacl
from pyshacl import validate
from rdflib import Graph

data_graph = Graph()
data_graph.parse('...')
shacl_str = \"" ... \""
shacl_graph = Graph()
shacl_graph.parse(data=shacl_str, format='ttl')

results = validate(
  data_graph,
  shacl_graph=shacl_graph,
  inference='both'
)

(conforms, results_graph, results_text) = results
print(results_text)
```

Formal Definition

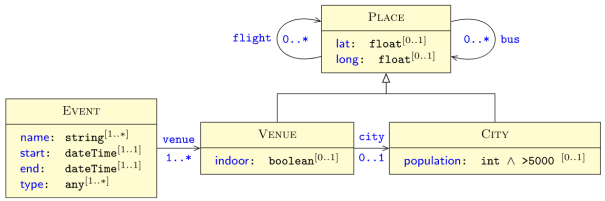
Definition

Shape A *shape* ϕ is defined as:

$$
\begin{array}{rcll}
\phi & ::= & \top & \text{true} \\
 & | & \Delta_N & \text{node belongs to set N} \\
 & | & \Psi_C & \text{node satisfies condition } C \\
 & | & \phi \wedge \phi & \text{conjunction} \\
 & | & \neg\phi & \text{negation} \\
 & | & @S & \text{shape with label } S \\
 & | & \widehat{p}\phi\{\min, \max\} & \text{range of } p \text{ edges } \textit{satsifying} \phi
\end{array}
$$

Definition

Shapes schema A *shapes schema* is defined as a tuple $\Sigma = (\Phi, S, \lambda)$ where Φ is a set of shapes, S is a set of shape labels, and $\lambda : S \longrightarrow \Phi$ is a total function from labels to shapes.



$$
\begin{aligned}
&\textsc{Event} \mapsto \xrightarrow{\text{name}} \Delta_{\text{string}}\{1,*\} \wedge \xrightarrow{\text{start}} \Delta_{\text{dateTime}}\{1,1\} \wedge \xrightarrow{\text{end}} \Delta_{\text{dateTime}}\{1,1\} \\
&\qquad \wedge \xrightarrow{\text{type}} \top\{1,*\} \wedge \xrightarrow{\textit{venue}} \textsc{Venue}\{1,*\} \\
&\textsc{Venue} \mapsto \textsc{Place} \wedge \xrightarrow{\text{indoor}} \Delta_{\text{boolean}}\{0,1\} \wedge \xrightarrow{\textit{city}} \textsc{City}\{0,1\} \\
&\textsc{City} \mapsto \textsc{Place} \wedge \xrightarrow{\text{population}} (\Delta_{\text{int}} \wedge \Psi_{>5000})\{0,1\} \\
&\textsc{Place} \mapsto \xrightarrow{\text{lat}} \Delta_{\text{float}}\{0,1\} \wedge \xrightarrow{\text{long}} \Delta_{\text{float}}\{0,1\} \\
&\qquad \wedge \xrightarrow{\textit{flight}} \textsc{Place}\{0,*\} \wedge \xrightarrow{\textit{bus}} \textsc{Place}\{0,*\}
\end{aligned}
$$

Formal Evaluation

Definition

Shapes map Given a directed edge-labelled graph $G = (V, E, L)$ and a shapes schema $\Sigma = (\Phi, S, \lambda)$, a shapes map is a (partial) mapping $\sigma : V \times S \longrightarrow \{0, 1\}$.

Definition

Shape Evaluation Given a shapes schema, $\Sigma = (\Phi, S, \lambda)$, a directed edge-labeled graph $G = (V, E, L)$, a node $v \in V$ and a total shapes map σ, the *shape evaluation function* $\phi^{G,v,\sigma}$ is defined recursively:

$$
\begin{aligned}
\top^{G,v,\sigma} &= 1 \\
\Delta_N^{G,v,\sigma} &= 1 \text{iff} v \in N \\
\Psi C^{G,v,\sigma} &= 1 \text{iff} C(v) \\
\phi 1 \wedge \phi_2^{G,v,\sigma} &= \min \phi_1^{G,v,\sigma}, \phi_2^{G,v,\sigma} \\
\neg \phi^{G,v,\sigma} &= 1 - [\phi]^{G,v,\sigma} \\
@S^{G,v,\sigma} &= 1 \text{iff} \sigma(v, s) = 1 \\
\widehat{p}\phi\{\ell, u\}^{G,v,\sigma} &= 1 \text{iff } \ell \leq |\{(v, p, u) | \phi^(G, u, \sigma) = 1\}| \leq u
\end{aligned}
$$

Shapes Target

Typically a shapes target is defined that requires certain nodes to satisfy certain shapes.

Definition

Shapes target Given a directed edge-labelled graph $G = (V, E, L)$ and a shapes schema $\Sigma = (\Phi, S, \lambda)$, a *shapes target* $T \subseteq V \times S$ is a set of pairs of nodes and shape labels from G and Σ.

Lastly, we define the notion of a valid graph under a given shapes schema and target based on the existence of a shapes map satisfying certain conditions.

Definition

Valid graph Given a shapes schema $\Sigma = (\Phi, S, \lambda)$, a directed edge-labelled graph $G = (V, E, L)$, and a shapes target T, we say that G is valid under Σ and T if and only if there exists a shapes map σ such that, for all $s \in S$ and $v \in V$ it holds that $\sigma(v, s) = \lambda(s)^{G,v,\sigma}$ and $(v, s) \in T$ implies $\sigma(v, s) = 1$.

Summary

We have considered pragmatic aspects of knowledge graphs where common constraints and inferences can be included in the schema of the graph.
However, it is not clear if this enough. Can we describe all inference and knowledge in this format?
Next we will move in this direction, by considering the study of everything! Ontologies! (... and the web ontology language, OWL).