TECHNISCHE UNIVERSITÄT DRESDEN

International Center for Computational Logic

KNOWLEDGE GRAPHS

Lecture 3: Modelling in RDF / SPARQL Basics

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 14 Nov 2024

More recent versions of this slide deck might be available.
For the most current version of this course, see
https://iccl.inf.tu-dresden.de/web/Knowledge_Graphs/en

Review: RDF Graphs

The W3C Resource Description Framework considers three types of RDF terms:

- IRIs, representing a resource using a global identifier
- Blank nodes, representing an unspecified resource without giving any identifier
- Literals that represent values of some datatype
  - either typed literals, such as `"2020-11-10"^^<http://www.w3.org/2001/XMLSchema#date>`
  - or language-tagged strings, such as `"Knowledge Graphs"@en`

RDF graphs are sets of triples consisting of

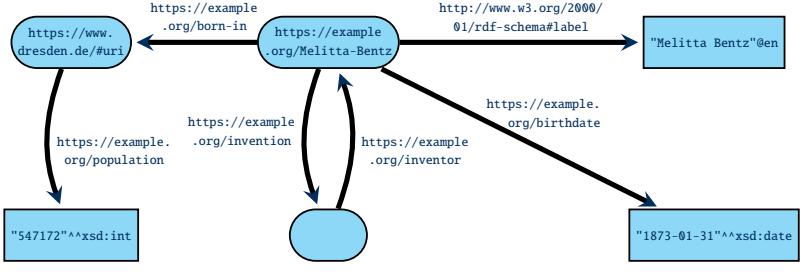
- a subject, which can be an IRI or bnode,
- a predicate, which can be an IRI,
- an object, which might be an IRI, bnode, or literal

$\rightsquigarrow$ naturally viewed as hypergraphs with ternary edges

RDF glossary

- Bnode: blank node
- (Datatype) literal: syntactic representation of a data value, always specifies the datatype
- Data value: actual (semantic) data value, such as a concrete integer number
- Graph: set of RDF triples, also: the (hyper)graph structure it represents
- Property: an IRI that is used in the predicate position of RDF triples
- Qualified name (qname): abbreviated IRI written as `prefix:name`
- Resource: the thing that an IRI refers to; domain element
- Term (RDF term): IRI, literal, or bnode

Review: Turtle



could be encoded as:

```
BASE <https://example.org/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

<Melitta-Bentz>  rdfs:label  "Melitta Bentz"@en ;
                 <birthdate>  "1873-01-31"^^xsd:date ;
                 <invention>  [ <inventor>  <Melitta-Bentz> ] ;
                 <born-in>  <https://www.dresden.de/#uri> .
<https://www.dresden.de/#uri>  <population>  547172 .
```

RDF Datasets

RDF 1.1 also supports datasets that consist of several graphs:

- Useful for organising RDF data, especially within databases
- Several named graphs are identified by IRIs; there is also one default graph without any IRI
- RDF dataset = RDF data that may have more than one graph

RDF Datasets

RDF 1.1 also supports datasets that consist of several graphs:

- Useful for organising RDF data, especially within databases
- Several named graphs are identified by IRIs; there is also one default graph without any IRI
- RDF dataset = RDF data that may have more than one graph

Various syntactic forms can serialise RDF datasets with named graphs:

- N-Quads: Extension of N-Triples with optional fourth component in each line to denote graph.
- TriG: Extension of Turtle with a new feature to declare graphs (group triples of one graph in braces, with the graph IRI written before the opening brace)
- JSON-LD: Can also encode named graphs

RDF Datasets

RDF 1.1 also supports datasets that consist of several graphs:

- Useful for organising RDF data, especially within databases
- Several named graphs are identified by IRIs; there is also one default graph without any IRI
- RDF dataset = RDF data that may have more than one graph

Various syntactic forms can serialise RDF datasets with named graphs:

- N-Quads: Extension of N-Triples with optional fourth component in each line to denote graph.
- TriG: Extension of Turtle with a new feature to declare graphs (group triples of one graph in braces, with the graph IRI written before the opening brace)
- JSON-LD: Can also encode named graphs

How to interpret named graphs?

- W3C standards do not define any official semantics
- Practical approach: interpret RDF datasets as data structures that are sets of quads (similar to the predominant practical interpretation of triples, in spite of their official semantics)

Which IRIs to use?

Where do the IRIs that we use in graphs come from?

Which IRIs to use?

Where do the IRIs that we use in graphs come from?

- They can be newly created for an application
  ⇝ avoid confusion with resources in other graphs
- They can be IRIs that are already in common use
  ⇝ support information integration and re-use across graphs

Which IRIs to use?

Where do the IRIs that we use in graphs come from?

- They can be newly created for an application
  $\rightsquigarrow$ avoid confusion with resources in other graphs
- They can be IRIs that are already in common use
  $\rightsquigarrow$ support information integration and re-use across graphs

Guidelines for creating new IRIs:

1. Check if you could re-use an existing IRI $\rightsquigarrow$ avoid duplication if feasible
2. Use http(s) IRIs $\rightsquigarrow$ useful protocols, registries, resolution mechanisms
3. Create new IRIs based on domains that you own $\rightsquigarrow$ clear ownership; no danger of clashing with other people's IRIs
4. Don't use URLs of existing web pages, unless you want to store data about pages $\rightsquigarrow$ avoid confusion between pages and more abstract resources
5. Make your IRIs return some useful content via http(s) $\rightsquigarrow$ helps others to get information about your resources

Excursion: Resolvable IRIs

We asked for IRIs that are different from URLs of Web pages while at the same time returning some useful content via http(s).

How is this possible?

Excursion: Resolvable IRIs

We asked for IRIs that are different from URLs of Web pages while at the same time returning some useful content via http(s).

How is this possible?

(1) **Use fragments.** IRIs with fragments are different from the IRIs (URLs) without the fragment, but resolving them will return the content of this fragment-less IRI.

> **Example 3.1:** RDF uses some own IRIs for special purposes, e.g., `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` for denoting the relation between resources and their type (class). This resolves to `http://www.w3.org/1999/02/22-rdf-syntax-ns` but is not the identifier of this document.

Excursion: Resolvable IRIs

We asked for IRIs that are different from URLs of Web pages while at the same time returning some useful content via http(s).

How is this possible?

(1) **Use fragments.** IRIs with fragments are different from the IRIs (URLs) without the fragment, but resolving them will return the content of this fragment-less IRI.

(2) **Use HTTP redirects.** Web servers can be configured to transparently redirect one URL to another; for IRIs it is common to use HTTP response codes 302 (temporary redirect) or 303 (see other)

> **Example 3.1:** The Wikidata IRI `http://www.wikidata.org/entity/Q5` redirects permanently (301) to `https://www.wikidata.org/entity/Q5`, which redirects (303) to `https://www.wikidata.org/wiki/Special:EntityData/Q5`, which by default redirects (303) to a JSON document `https://www.wikidata.org/wiki/Special:EntityData/Q5.json`.
>
> Real-world server configurations can be complicated.

Excursion: Content negotiation

The redirect mechanism has another useful application:
Web servers (and the software they run) can dynamically decide where to redirect to.

Content negotiation is the practice of offering several documents under the same URL by redirecting to the version that is most suitable for the request of a client. The HTTP protocol allows clients and servers to exchange extra information with their request that can be used to state preferences.

Example 3.2: The `Accept` header is commonly used by clients to select preferred result formats for a request. For example, the command

```
curl -L -H "Accept: text/turtle" http://www.wikidata.org/entity/Q5
```

makes Wikidata return RDF data in Turtle format rather than JSON.

Modeling data in RDF

Representing data in RDF

Goal: Let's manage more of our data in RDF (for interoperability, data integration, uniform tooling, use RDF software, . . .) – Is this possible?

Representing data in RDF

Goal: Let's manage more of our data in RDF (for interoperability, data integration, uniform tooling, use RDF software, . . .) – Is this possible?

Any information can be encoded in RDF, in particular:

- Relational data, including CSV files
- JSON documents and other "object models"
- XML documents and other "document models"
- Other kinds of graphs and database formats
- (Abstractions of) program code
- . . .

But not all data should be RDF (e.g., media files or executable binaries)

Representing data in RDF

Goal: Let's manage more of our data in RDF (for interoperability, data integration, uniform tooling, use RDF software, . . .) – Is this possible?

Any information can be encoded in RDF, in particular:

- Relational data, including CSV files
- JSON documents and other "object models"
- XML documents and other "document models"
- Other kinds of graphs and database formats
- (Abstractions of) program code
- . . .

But not all data should be RDF (e.g., media files or executable binaries)

Main steps that are needed:

- Define IRIs for relevant resources
- Define RDF graph structure from given data structure

Encoding data as graphs

Challenges when translating data to RDF:

Encoding data as graphs

Challenges when translating data to RDF:

- RDF has only triples – what to do with relations of higher arity (e.g., from a relational database)?
  $\rightsquigarrow$ covered, e.g., by RDB2RDF standard; fairly common

Encoding data as graphs

Challenges when translating data to RDF:

- RDF has only triples – what to do with relations of higher arity (e.g., from a relational database)?
  ⇝ covered, e.g., by RDB2RDF standard; fairly common
- RDF graphs are (unordered) sets of triples – how to represent ordered lists (including, e.g., the ordered children of a node in XML)?
  ⇝ several possible ways around; no standard solution

Encoding data as graphs

Challenges when translating data to RDF:

- RDF has only triples – what to do with relations of higher arity (e.g., from a relational database)?
  ⇝ covered, e.g., by RDB2RDF standard; fairly common
- RDF graphs are (unordered) sets of triples – how to represent ordered lists (including, e.g., the ordered children of a node in XML)?
  ⇝ several possible ways around; no standard solution
- Other data structures might be needed, e.g., maps (associative arrays) or undirected graphs
  ⇝ custom encodings needed; usually no standard solution

Encoding data as graphs

Challenges when translating data to RDF:

- RDF has only triples – what to do with relations of higher arity (e.g., from a relational database)?
  ⇝ covered, e.g., by RDB2RDF standard; fairly common
- RDF graphs are (unordered) sets of triples – how to represent ordered lists (including, e.g., the ordered children of a node in XML)?
  ⇝ several possible ways around; no standard solution
- Other data structures might be needed, e.g., maps (associative arrays) or undirected graphs
  ⇝ custom encodings needed; usually no standard solution
- The datatypes of other formats need to be suitably translated to RDF datatypes
  ⇝ often not hard once it is clear what type the source data really is

Representing n-ary relations

Consider the following relational table:

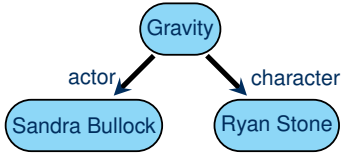
Film	Actor	Character
Arrival	Amy Adams	Louise Banks
Arrival	Jeremy Renner	Ian Donnelly
Gravity	Sandra Bullock	Ryan Stone

Representing n-ary relations

Consider the following relational table:

Film	Actor	Character
Arrival	Amy Adams	Louise Banks
Arrival	Jeremy Renner	Ian Donnelly
Gravity	Sandra Bullock	Ryan Stone

Possible encoding:

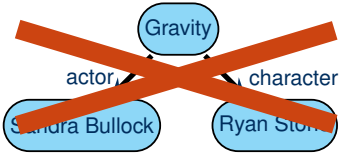


Representing n-ary relations

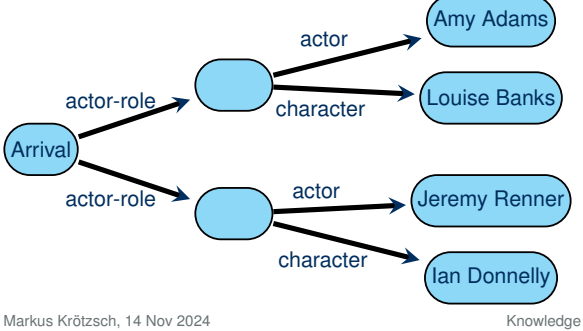
Consider the following relational table:

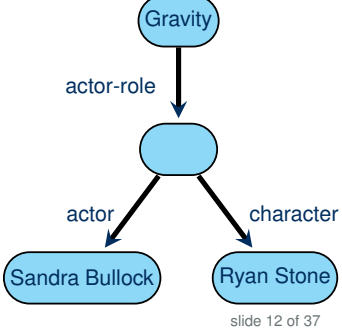
Film	Actor	Character
Arrival	Amy Adams	Louise Banks
Arrival	Jeremy Renner	Ian Donnelly
Gravity	Sandra Bullock	Ryan Stone

Insufficient encoding:



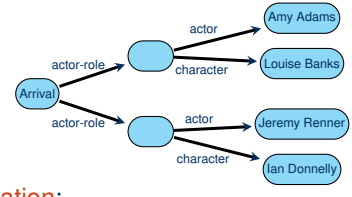
Possible encoding:

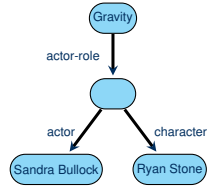




Reification

Film	Actor	Character
Arrival	Amy Adams	Louise Banks
Arrival	Jeremy Renner	Ian Donnelly
Gravity	Sandra Bullock	Ryan Stone





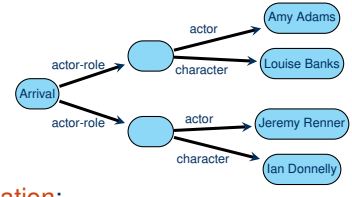
This type of encoding is known as reification:

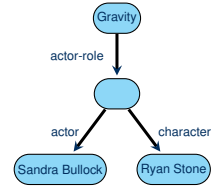
- Introduce auxiliary nodes to represent relationships
- Connect related objects and values with the auxiliary node

⇝ Can be realised in many specific ways (bnodes or not, direction of relations)

Reification

Film	Actor	Character
Arrival	Amy Adams	Louise Banks
Arrival	Jeremy Renner	Ian Donnelly
Gravity	Sandra Bullock	Ryan Stone





This type of encoding is known as reification:

- Introduce auxiliary nodes to represent relationships
- Connect related objects and values with the auxiliary node

⇝ Can be realised in many specific ways (bnodes or not, direction of relations)

RDF Reification: RDF has a dedicated vocabulary to reify RDF triples:

- Reified triples x marked as x `rdf:type rdf:Statement .`
- Triple-pieces connected by properties `rdf:subject`, `rdf:predicate`, and `rdf:object`

Rarely used (reasons: specific to triples, inflexible, incompatible with OWL DL).

RDB2RDF

The W3C's "RDB2RDF" working group has published a direct mapping from relational databases to RDF graphs (`https://www.w3.org/TR/rdb-direct-mapping/`):

- Basic approach is to reify triples as shown
- Special guidelines for creating IRIs for auxiliary nodes (no bnodes!), taking table names into account
- Mapping from SQL table entries to RDF literals
- Additional properties and triples to capture foreign-key relations

More specific mappings can be defined using the R2RML RDB to RDF Mapping Language (`https://www.w3.org/TR/r2rml/`).

CSV: Comma-Separated Values

One of the most common exchange formats for data

- Simple, text-based encoding of tabular data
- No schema, no datatypes
- Many variants (delimiter-separated values, different string escape mechanisms)

CSV: Comma-Separated Values

One of the most common exchange formats for data

- Simple, text-based encoding of tabular data
- No schema, no datatypes
- Many variants (delimiter-separated values, different string escape mechanisms)

How to represent CSV data in RDF:

- Use reification (or RDB2RDF) to represent n-ary tuples
- Heuristics or additional hints needed to map CSV strings to data values
- RDF supports inconsistent datatypes and missing data well

Limitation: order of rows cannot be captured in an easy way

Representing order (1)

Problem: RDF can easily represent sets, but not lists

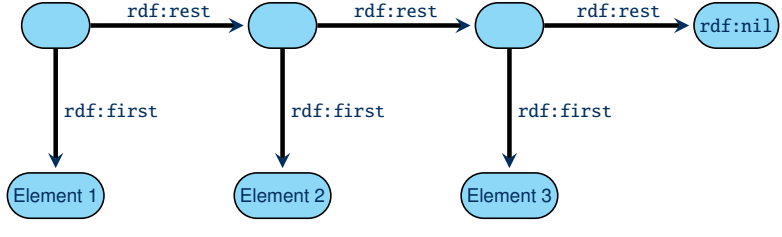
Representing order (1)

Problem: RDF can easily represent sets, but not lists

Possible solution (1): Represent lists as linked lists

- Supported by “RDF Collections” (`rdf:first`, `rdf:rest`, `rdf:nil`)
- **Pro:** Faithful, elegant graph representation; easy insert/delete
- **Con:** Inefficient access (many joins)

Example:



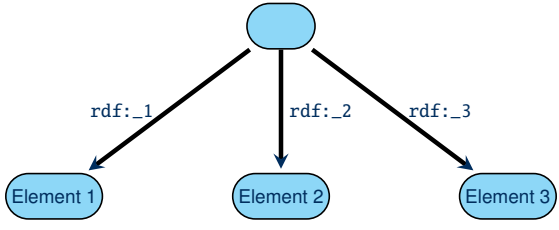
Representing order (2)

Problem: RDF can easily represent sets, but not lists

Possible solution (2): Use order-indicating properties (one property per position)

- Supported by “RDF Containers”
  (`rdf:_1`, `rdf:_2`, ..., `rdfs:ContainerMembershipProperty`)
- **Pro:** direct access, natural for fixed-length lists (e.g., RDBMS rows)
- **Con:** list structure not in graph – requires special knowledge; length only detectable by absence of triples; harder insert/delete

Example:



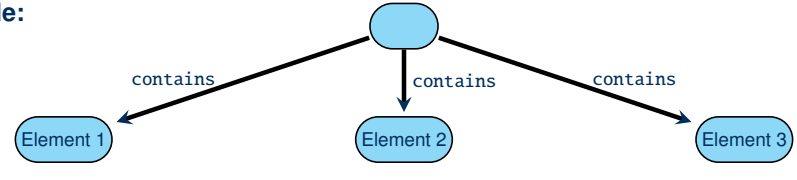
Representing order (3)

Problem: RDF can easily represent sets, but not lists

Possible solution (3): Give up order and represent lists as sets

- Easy to do in RDF
- **Pro:** simple; may suffice for many use cases
- **Con:** no order

Example:



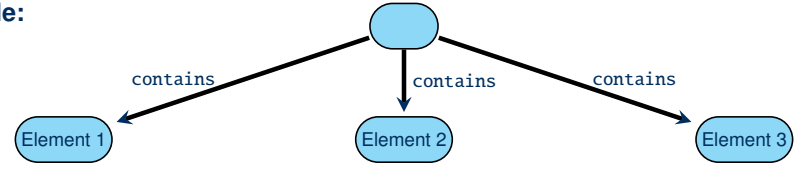
Representing order (3)

Problem: RDF can easily represent sets, but not lists

Possible solution (3): Give up order and represent lists as sets

- Easy to do in RDF
- **Pro:** simple; may suffice for many use cases
- **Con:** no order

Example:



Possible compromise: Combine several encodings within one graph
(increased data volume – easier to read – harder to write)

Representing order (4)

Problem: RDF can easily represent sets, but not lists

Possible solution (4): Use custom list datatype

- Proposals include Amazon's Complex Datatypes for SPARQL (link)
- **Pro:** direct & efficient; lists appear as single "values"
- **Con:** unofficial; not widely supported yet; special functions needed to access/create lists; proposals make further design decisions (SPARQL CDT: inclusion of special "null" value, use of uninterpreted literals in list values, etc.)

Example:

```
"[ <http://example.org/elem1>, '2024-11-07'^^<http://www.w3.org/2001/XMLSchema#date> ]"
                    ^^<http://w3id.org/awslabs/neptune/SPARQL-CDTs/List>
```

Representing order (4)

Problem: RDF can easily represent sets, but not lists

Possible solution (4): Use custom list datatype

- Proposals include Amazon's Complex Datatypes for SPARQL (link)
- **Pro:** direct & efficient; lists appear as single "values"
- **Con:** unofficial; not widely supported yet; special functions needed to access/create lists; proposals make further design decisions (SPARQL CDT: inclusion of special "null" value, use of uninterpreted literals in list values, etc.)

Example:

```
"[ <http://example.org/elem1>, '2024-11-07'^^<http://www.w3.org/2001/XMLSchema#date> ]"
                    ^^<http://w3id.org/awslabs/neptune/SPARQL-CDTs/List>
```

A W3C standard for list data? Lists also appear prominently in the ongoing standardisation activity for N3 (a rule language for RDF). This might lead to further syntactic options in the future.

Representing other data structures

Many further data structures exist, and are not hard to express in RDF.

Maps (associative arrays): key-value mappings

Representing other data structures

Many further data structures exist, and are not hard to express in RDF.

Maps (associative arrays): key-value mappings

- Without order: set of pairs, or keys as RDF properties
- With order: lists of pairs (using any list encoding)

Representing other data structures

Many further data structures exist, and are not hard to express in RDF.

Maps (associative arrays): key-value mappings

- Without order: set of pairs, or keys as RDF properties
- With order: lists of pairs (using any list encoding)

Multisets: sets with repeated elements; “unordered lists”

Representing other data structures

Many further data structures exist, and are not hard to express in RDF.

Maps (associative arrays): key-value mappings

- Without order: set of pairs, or keys as RDF properties
- With order: lists of pairs (using any list encoding)

Multisets: sets with repeated elements; “unordered lists”

- encode as sets where every element has a count ≥ 1

Representing other data structures

Many further data structures exist, and are not hard to express in RDF.

Maps (associative arrays): key-value mappings

- Without order: set of pairs, or keys as RDF properties
- With order: lists of pairs (using any list encoding)

Multisets: sets with repeated elements; “unordered lists”

- encode as sets where every element has a count ≥ 1

Undirected graphs: possibly labeled

Representing other data structures

Many further data structures exist, and are not hard to express in RDF.

Maps (associative arrays): key-value mappings

- Without order: set of pairs, or keys as RDF properties
- With order: lists of pairs (using any list encoding)

Multisets: sets with repeated elements; “unordered lists”

- encode as sets where every element has a count ≥ 1

Undirected graphs: possibly labeled

- standard encoding: specify same edges in both directions

Representing other data structures

Many further data structures exist, and are not hard to express in RDF.

Maps (associative arrays): key-value mappings

- Without order: set of pairs, or keys as RDF properties
- With order: lists of pairs (using any list encoding)

Multisets: sets with repeated elements; “unordered lists”

- encode as sets where every element has a count ≥ 1

Undirected graphs: possibly labeled

- standard encoding: specify same edges in both directions

Pro: usually straightforward combination of previous approaches
Con: data-structure specific constraints usually not enforced by encoding; relevant operations not supported by RDF tools (e.g., compare two structures for equality)

JSON: Javascript Object Notation

One of the most common exchange formats for data on the Web

- Simple, text-based encoding of nested lists and maps
- Few datatypes (standard does not define datatypes, but JavaScript/ECMAScript does)

JSON: Javascript Object Notation

One of the most common exchange formats for data on the Web

- Simple, text-based encoding of nested lists and maps
- Few datatypes (standard does not define datatypes, but JavaScript/ECMAScript does)

How to represent JSON data in RDF:

- Select RDF encodings for arbitrary lists and maps (note: JSON keys are strings, i.e., no IRIs that can be used as properties directly)
- Only three types of data: `xsd:string`, `xsd:double`, `xsd:boolean`
- Special value “null” needs to be encoded by dedicated IRI (not perfect, but workable)

Note: SPARQL CDT also defines an informal mapping from JSON. Proposes Turtle-like interpretation of numbers instead of using double as in JavaScript.

Describing schema information in RDF

Triples about properties can also be used to specify how properties should be used.

Example 3.3: RDF provides several properties for describing properties:

```
<PropertyIRI>  rdf:type  rdfs:Property .      # declare resource as property
<PropertyIRI>  rdfs:label  "some label"@en . # assign label
<PropertyIRI>  rdfs:comment  "Some human-readable comment"@en .
<PropertyIRI>  rdfs:range  xsd:decimal .      # define range datatype
<PropertyIRI>  rdfs:domain  <classIRI> .      # define domain type (class)
```

Describing schema information in RDF

Triples about properties can also be used to specify how properties should be used.

Example 3.3: RDF provides several properties for describing properties:

```
<PropertyIRI>  rdf:type  rdfs:Property .      # declare resource as property
<PropertyIRI>  rdfs:label  "some label"@en . # assign label
<PropertyIRI>  rdfs:comment  "Some human-readable comment"@en .
<PropertyIRI>  rdfs:range  xsd:decimal .      # define range datatype
<PropertyIRI>  rdfs:domain  <classIRI> .      # define domain type (class)
```

- There are many properties beyond those from the RDF standard for such purposes, e.g., `rdfs:label` can be replaced by `<http://schema.org/name>` or `<http://www.w3.org/2004/02/skos/core#prefLabel>`
- RDF defines how its properties should be interpreted semantically (see later lectures)
- There are more elaborate ways of expressing schematic information in RDF
  - The OWL Web Ontology Language extends the semantic features of RDF
  - Constraint languages SHACL and SHEX can restrict graphs syntactically

  (see later lectures)

No schema?

RDF supports properties without any declaration, even with different types of values

Example 3.4: The property `http://purl.org/dc/elements/1.1/creator` from the Dublin Core vocabulary has been used with values that are IRIs (denoting a creator) or strings (names of a creator).

- RDF tools can tolerate such lack of schema . . .
- . . . but it is usually not desirable for applications

The robustness of RDF is useful for merging datasets, but it's easier to build services around more uniform/constrained data

Introduction to SPARQL

An RDF query language and more

SPARQL is short for SPARQL Protocol and RDF Query Language

- W3C standard since 2008
- Updated in 2013 (SPARQL 1.1)
- Supported by many graph databases

An RDF query language and more

SPARQL is short for SPARQL Protocol and RDF Query Language

- W3C standard since 2008
- Updated in 2013 (SPARQL 1.1)
- Supported by many graph databases

The SPARQL specification consists of several major parts:

- A query language
- Result formats in XML, JSON, CSV, and TSV
- An update language
- Protocols for communicating with online SPARQL services
- A vocabulary for describing SPARQL services

Full specifications can be found online:
`https://www.w3.org/TR/sparql11-overview/`

SPARQL queries

The heart of SPARQL is its query language.

Example 3.5: The following simple SPARQL query asks for a list of all resource IRIs together with their labels:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?resource ?label
WHERE {
  ?resource rdfs:label ?label .
}
```

Basic concepts:

- SPARQL uses variables, marked by their initial ?
- The core of a query is the query condition within `WHERE { ... }`
- Conditions can be simple patterns based on triples, similar to Turtle syntax
- `SELECT` specifies how results are produced from query matches

Basic SPARQL by example

Example 3.6: Find up to ten people whose daughter is a professor:

```
PREFIX eg: <http://example.org/>
SELECT ?parent
WHERE {
  ?parent eg:hasDaughter ?child .
  ?child  eg:occupation eg:Professor .
} LIMIT 10
```

Basic SPARQL by example

Example 3.6: Find up to ten people whose daughter is a professor:

```
PREFIX eg: <http://example.org/>
SELECT ?parent
WHERE {
  ?parent eg:hasDaughter ?child .
  ?child  eg:occupation eg:Professor .
} LIMIT 10
```

Example 3.7: Count all triples in the database:

```
SELECT (COUNT(*) AS ?count)
WHERE { ?subject ?predicate ?object . }
```

Basic SPARQL by example

Example 3.6: Find up to ten people whose daughter is a professor:

```
PREFIX eg: <http://example.org/>
SELECT ?parent
WHERE {
  ?parent eg:hasDaughter ?child .
  ?child  eg:occupation eg:Professor .
} LIMIT 10
```

Example 3.7: Count all triples in the database:

```
SELECT (COUNT(*) AS ?count)
WHERE { ?subject ?predicate ?object . }
```

Example 3.8: Count all predicates in the database:

```
SELECT (COUNT(DISTINCT ?predicate) AS ?count)
WHERE { ?subject ?predicate ?object . }
```

Basic SPARQL by example (2)

Example 3.9: Find the person with most friends:

```
SELECT ?person (COUNT(*) AS ?friendCount)
WHERE { ?person <http://example.org/hasFriend> ?friend . }
GROUP BY ?person
ORDER BY DESC(?friendCount) LIMIT 1
```

Basic SPARQL by example (2)

Example 3.9: Find the person with most friends:

```
SELECT ?person (COUNT(*) AS ?friendCount)
WHERE { ?person <http://example.org/hasFriend> ?friend . }
GROUP BY ?person
ORDER BY DESC(?friendCount) LIMIT 1
```

Example 3.10: Find pairs of siblings:

```
SELECT ?child1 ?child2
WHERE {
  ?parent <http://example.org/hasChild> ?child1, ?child2 .
}
```

Basic SPARQL by example (2)

Example 3.9: Find the person with most friends:

```
SELECT ?person (COUNT(*) AS ?friendCount)
WHERE { ?person <http://example.org/hasFriend> ?friend . }
GROUP BY ?person
ORDER BY DESC(?friendCount) LIMIT 1
```

Example 3.10: Find pairs of siblings:

```
SELECT ?child1 ?child2
WHERE {
  ?parent <http://example.org/hasChild> ?child1, ?child2 .
  FILTER(?child1 != ?child2)
}
```

The shape of a SPARQL query

Select queries consist of the following major blocks:

- Prologue: for `PREFIX` and `BASE` declarations (work as in Turtle)
- Select clause: `SELECT` (and possibly other keywords) followed either by a list of variables (e.g., `?person`) and variable assignments (e.g., `(COUNT(*) as ?count)`), or by `*`
- Where clause: `WHERE` followed by a pattern (many possibilities)
- Solution set modifiers: such as `LIMIT` or `ORDER BY`

The shape of a SPARQL query

Select queries consist of the following major blocks:

- Prologue: for `PREFIX` and `BASE` declarations (work as in Turtle)
- Select clause: `SELECT` (and possibly other keywords) followed either by a list of variables (e.g., `?person`) and variable assignments (e.g., `(COUNT(*) as ?count)`), or by `*`
- Where clause: `WHERE` followed by a pattern (many possibilities)
- Solution set modifiers: such as `LIMIT` or `ORDER BY`

SPARQL supports further types of queries, which primarily exchange the Select clause for something else:

- ASK query: to check whether there are results at all (but don't return any)
- CONSTRUCT query: to build an RDF graph from query results
- DESCRIBE query: to get an RDF graph with additional information on each query result (application dependent)

Basic SPARQL syntax

RDF terms are written like in Turtle:

- IRIs may be abbreviated using `qualified:names` (requires `PREFIX` declaration) or `<relativeIris>` (requires `BASE` declaration)
- Literals are written as usual, possibly also with abbreviated datatype IRIs
- Blank nodes are written as usual

Basic SPARQL syntax

RDF terms are written like in Turtle:

- IRIs may be abbreviated using `qualified:names` (requires `PREFIX` declaration) or `<relativeIris>` (requires `BASE` declaration)
- Literals are written as usual, possibly also with abbreviated datatype IRIs
- Blank nodes are written as usual

In addition, SPARQL supports variables:

Definition 3.11: A variable is a string that begins with `?` or `$`, where the string can consist of letters (including many non-Latin letters), numbers, and the symbol `_`. The variable name is the string after `?` or `$`, without this leading symbol.

Basic SPARQL syntax

RDF terms are written like in Turtle:

- IRIs may be abbreviated using `qualified:names` (requires `PREFIX` declaration) or `<relativeIris>` (requires `BASE` declaration)
- Literals are written as usual, possibly also with abbreviated datatype IRIs
- Blank nodes are written as usual

In addition, SPARQL supports variables:

Definition 3.11: A variable is a string that begins with `?` or `$`, where the string can consist of letters (including many non-Latin letters), numbers, and the symbol `_`. The variable name is the string after `?` or `$`, without this leading symbol.

Example 3.12: The variables `?var1` and `$var1` have the same variable name (and same meaning across SPARQL).

Convention: Using `?` is widely preferred these days!

Basic Graph Patterns

We can now define the simplest kinds of patterns:

Definition 3.13: A triple pattern is a triple $\langle \mathtt{s}, \mathtt{p}, \mathtt{o} \rangle$, where $\mathtt{s}$ and $\mathtt{o}$ are arbitrary RDF terms[a] or variables, and $\mathtt{p}$ is an IRI or variable. A basic graph pattern (BGP) is a set of triple patterns.

[a] Curiously, SPARQL allows literals as subjects, although RDF does not.*

Note: These are semantic notions, that are not directly defining query syntax. Triple patterns describe query conditions where we are looking for matching triples. BGPs are interpreted conjunctively, i.e., we are looking for a match that fits all triples at once.

Syntactically, SPARQL supports an extension of Turtle (that allows variables everywhere and literals in subject positions). All Turtle shortcuts are supported.

* This was done for forwards compatibility with future RDF versions, but RDF 1.1 did not add any such extension. Hence such patterns can never match in RDF.

Basic Graph Patterns

We can now define the simplest kinds of patterns:

Definition 3.13: A triple pattern is a triple $\langle \mathtt{s}, \mathtt{p}, \mathtt{o} \rangle$, where $\mathtt{s}$ and $\mathtt{o}$ are arbitrary RDF terms[a] or variables, and $\mathtt{p}$ is an IRI or variable. A basic graph pattern (BGP) is a set of triple patterns.

[a] Curiously, SPARQL allows literals as subjects, although RDF does not.*

Note: These are semantic notions, that are not directly defining query syntax. Triple patterns describe query conditions where we are looking for matching triples. BGPs are interpreted conjunctively, i.e., we are looking for a match that fits all triples at once.

Syntactically, SPARQL supports an extension of Turtle (that allows variables everywhere and literals in subject positions). All Turtle shortcuts are supported.

Convention: We will also use the word triple pattern and basic graph pattern to refer to any (syntactic) Turtle snippet that specifies such (semantic) patterns.

* This was done for forwards compatibility with future RDF versions, but RDF 1.1 did not add any such extension. Hence such patterns can never match in RDF.

Blank nodes

Remember: Bnode ids are syntactic aids to allow us serialising graphs with such nodes. They are not part of the RDF graph.

What is the meaning of blank nodes in query patterns?

Blank nodes

Remember: Bnode ids are syntactic aids to allow us serialising graphs with such nodes. They are not part of the RDF graph.

What is the meaning of blank nodes in query patterns?

- They denote an unspecified resource (in particular: they do not ask for a bnode of a specific node id in the queried graph!)
- In other words: they are like variables, but cannot be used in `SELECT`
- Turtle bnode syntax can be used (`[]` or `_:nodeId`), but any node id can only appear in one part of the query (we will see complex queries with many parts later)

Blank nodes

Remember: Bnode ids are syntactic aids to allow us serialising graphs with such nodes. They are not part of the RDF graph.

What is the meaning of blank nodes in query patterns?

- They denote an unspecified resource (in particular: they do not ask for a bnode of a specific node id in the queried graph!)
- In other words: they are like variables, but cannot be used in `SELECT`
- Turtle bnode syntax can be used (`[]` or `_:nodeId`), but any node id can only appear in one part of the query (we will see complex queries with many parts later)

What is the meaning of blank nodes in query results?

Blank nodes

Remember: Bnode ids are syntactic aids to allow us serialising graphs with such nodes. They are not part of the RDF graph.

What is the meaning of blank nodes in query patterns?

- They denote an unspecified resource (in particular: they do not ask for a bnode of a specific node id in the queried graph!)
- In other words: they are like variables, but cannot be used in `SELECT`
- Turtle bnode syntax can be used (`[]` or `_:nodeId`), but any node id can only appear in one part of the query (we will see complex queries with many parts later)

What is the meaning of blank nodes in query results?

- Such bnodes indicate that a variable was matched to a bnode in the data
- The same node id may occur in multiple rows of the result table, meaning that the same bnode was matched
- However, the node id used in the result is an auxiliary id that might be different from what was used in the data (if an id was used there at all!)

Answers to BGPs

What is the result of a SPARQL query?

Definition 3.14: A solution mapping is a partial function μ from variable names to RDF terms. A solution sequence is a list of solution mappings.

Note: When no specific order is required, the solutions computed for a SPARQL query can be represented by a multiset (= "a set with repeated elements" = "an unordered list").

Answers to BGPs

What is the result of a SPARQL query?

Definition 3.14: A solution mapping is a partial function μ from variable names to RDF terms. A solution sequence is a list of solution mappings.

Note: When no specific order is required, the solutions computed for a SPARQL query can be represented by a multiset (= "a set with repeated elements" = "an unordered list").

Definition 3.15: Given an RDF graph G and a BGP P, a solution mapping μ is a solution to P over G if it is defined exactly on the variable names in P and there is a mapping σ from blank nodes to RDF terms, such that $\mu(\sigma(P)) \subseteq G$.

The cardinality of μ in the multiset of solutions is the number of distinct such mappings σ. The multiset of these solutions is denoted $\text{eval}_G(P)$, where we omit G if clear from the context.

Note: Here, we write $\mu(\sigma(P))$ to denote the graph given by the triples in P after first replacing bnodes according to σ, and then replacing variables according to μ.

Example

We consider a graph based on the earlier film-actor example (but with fewer bnodes!):

```
eg:Arrival eg:actorRole eg:aux1, eg:aux2 .
eg:aux1 eg:actor eg:Adams  ; eg:character "Louise Banks" .
eg:aux2 eg:actor eg:Renner ; eg:character "Ian Donnelly" .
eg:Gravity eg:actorRole [ eg:actor eg:Bullock;
                          eg:character "Ryan Stone" ] .
```

The BGP (and triple pattern) `?film eg:actorRole []` has the solution multiset:

film	cardinality
`eg:Arrival`	2
`eg:Gravity`	1

The cardinality of the first solution mapping is 2 since the bnode can be mapped to two resources, `eg:aux1` and `eg:aux2`, to find a subgraph.

Example (2)

We consider a graph based on the earlier film-actor example (but with fewer bnodes!):

```
eg:Arrival eg:actorRole eg:aux1, eg:aux2 .
eg:aux1 eg:actor eg:Adams  ; eg:character "Louise Banks" .
eg:aux2 eg:actor eg:Renner ; eg:character "Ian Donnelly" .
eg:Gravity eg:actorRole [ eg:actor eg:Bullock;
                          eg:character "Ryan Stone" ] .
```

The BGP `?film eg:actorRole [ eg:actor ?person ]` has the solution multiset:

film	**person**	cardinality
eg:Arrival	eg:Adams	1
eg:Arrival	eg:Renner	1
eg:Gravity	eg:Bullock	1

Boolean queries

Definition 3.15: Given an RDF graph G and a BGP P, a solution mapping μ is a solution to P over G if it is defined exactly on the variable names in P and there is a mapping σ from blank nodes to RDF terms, such that $\mu(\sigma(P)) \subseteq G$.

The cardinality of μ in the multiset of solutions is the number of distinct such mappings σ. The multiset of these solutions is denoted $\text{eval}_G(P)$, where we omit G if clear from the context.

Q: What is $\text{eval}_G(\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o})$ over the empty graph $G = \emptyset$?

Boolean queries

Definition 3.15: Given an RDF graph G and a BGP P, a solution mapping μ is a solution to P over G if it is defined exactly on the variable names in P and there is a mapping σ from blank nodes to RDF terms, such that $\mu(\sigma(P)) \subseteq G$.

The cardinality of μ in the multiset of solutions is the number of distinct such mappings σ. The multiset of these solutions is denoted $\text{eval}_G(P)$, where we omit G if clear from the context.

Q: What is $\text{eval}_G(\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o})$ over the empty graph $G = \emptyset$?

A: The empty multiset $\emptyset$ of solutions!

Boolean queries

Definition 3.15: Given an RDF graph G and a BGP P, a solution mapping μ is a solution to P over G if it is defined exactly on the variable names in P and there is a mapping σ from blank nodes to RDF terms, such that $\mu(\sigma(P)) \subseteq G$.

The cardinality of μ in the multiset of solutions is the number of distinct such mappings σ. The multiset of these solutions is denoted $\text{eval}_G(P)$, where we omit G if clear from the context.

Q: What is $\text{eval}_G(\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o})$ over the empty graph $G = \emptyset$?

A: The empty multiset $\emptyset$ of solutions!

Q: What is $\text{eval}_G(\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o})$ over the graph $G = \{\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o}\}$?

Boolean queries

Definition 3.15: Given an RDF graph G and a BGP P, a solution mapping μ is a solution to P over G if it is defined exactly on the variable names in P and there is a mapping σ from blank nodes to RDF terms, such that $\mu(\sigma(P)) \subseteq G$.

The cardinality of μ in the multiset of solutions is the number of distinct such mappings σ. The multiset of these solutions is denoted $\text{eval}_G(P)$, where we omit G if clear from the context.

Q: What is $\text{eval}_G(\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o})$ over the empty graph $G = \emptyset$?

A: The empty multiset $\emptyset$ of solutions!

Q: What is $\text{eval}_G(\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o})$ over the graph $G = \{\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o}\}$?

A: The multiset $\{\mu_0\}$ that contains the unique solution mapping with domain $\emptyset$ (the maximally partial function).

Boolean queries

Definition 3.15: Given an RDF graph G and a BGP P, a solution mapping μ is a solution to P over G if it is defined exactly on the variable names in P and there is a mapping σ from blank nodes to RDF terms, such that $\mu(\sigma(P)) \subseteq G$.

The cardinality of μ in the multiset of solutions is the number of distinct such mappings σ. The multiset of these solutions is denoted $\text{eval}_G(P)$, where we omit G if clear from the context.

Q: What is $\text{eval}_G(\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o})$ over the empty graph $G = \emptyset$?

A: The empty multiset $\emptyset$ of solutions!

Q: What is $\text{eval}_G(\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o})$ over the graph $G = \{\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o}\}$?

A: The multiset $\{\mu_0\}$ that contains the unique solution mapping with domain $\emptyset$ (the maximally partial function).

Terminology: Queries that cannot yield bindings for any variable are called Boolean queries, since they admit only two solutions: $\{\}$ (“false”) and $\{\mu_0\}$ (“true”).

Summary

RDF can express n-ary relations and (with some effort) lists

SPARQL, the main query language for RDF, is based on matching graph patterns

What's next?

- Wikidata as a working example to try out our knowledge
- More SPARQL query features
- Further background on SPARQL complexity and semantics