

KNOWLEDGE GRAPHS

Lecture 3: Modelling in RDF / SPARQL Basics

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 29th Oct 2019

Review: RDF Graphs

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

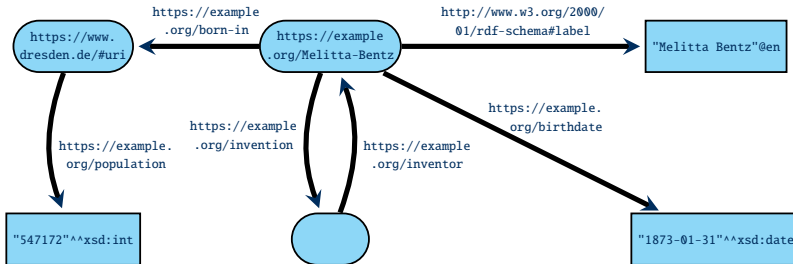
- **IRIs**, representing a resource using a global identifier
- **Blank nodes**, representing an unspecified resource without giving any identifier
- **Literals** that represent values of some datatype
 - either typed literals, such as
"2019-10-29"^^<http://www.w3.org/2001/XMLSchema#date>
 - or language-tagged 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

↪ naturally viewed as hypergraphs with ternary edges

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 .
```

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
- XML documents
- Other kinds of graphs
- Other kinds of 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
- XML documents
- Other kinds of graphs
- Other kinds of 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)?
~> 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

Representing n-ary relations

Consider the following relational table:

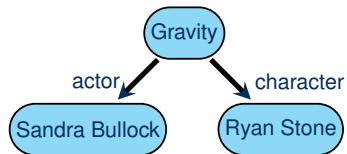
Film	Actor	Character
Arrival	Amy Adams	Louise Banks
Arrival	Jeremy Renner	Ian Donnelly
Gravity	Sandra Bullock	Ryan Stone

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:

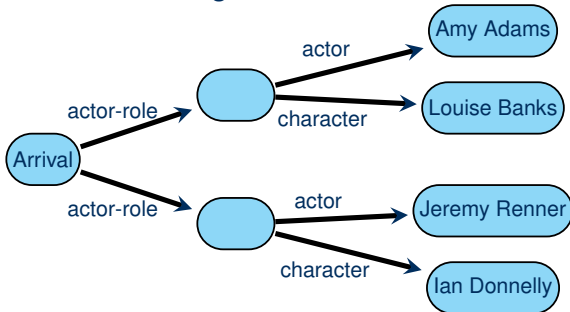


Representing n-ary relations

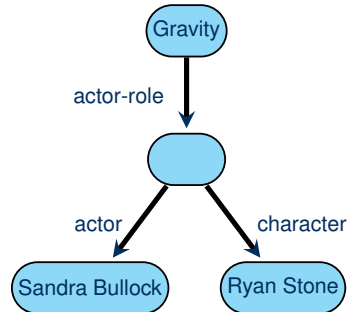
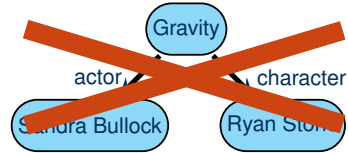
Consider the following relational table:

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

Possible encoding:

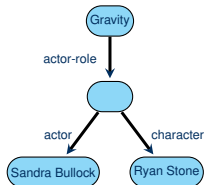
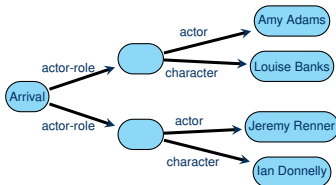


Insufficient encoding:



Reification

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



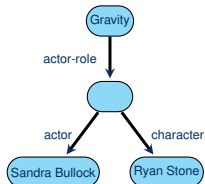
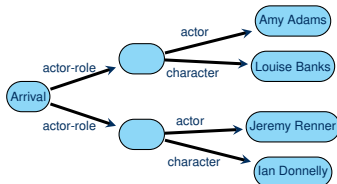
This type of encoding is known as **reification**:

- 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 even 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`

However, this is not frequently used (reasons: specific to triples, inflexible, incompatible with OWL DL).

RDB2RDF

The W3C's “RDB2RDF” working group has published a [direct mapping](https://www.w3.org/TR/rdb-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](https://www.w3.org/TR/r2rml/) RDB to RDF Mapping Language (<https://www.w3.org/TR/r2rml/>).

Representing order

Problem: RDF can easily represent sets, but not lists

Representing order

Problem: RDF can easily represent sets, but not lists

Possible solutions:

- 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)

Representing order

Problem: RDF can easily represent sets, but not lists

Possible solutions:

- 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)

- Use **order-indicating properties** (one property per position)

Supported by “RDF Containers” (`rdf:_1`, `rdf:_2`, ..., `rdfs:ContainerMembershipProperty`)

Pro: direct access; length only detectable by absence of triples

Con: list structure requires external knowledge; harder insert/delete

Representing order

Problem: RDF can easily represent sets, but not lists

Possible solutions:

- 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)
- Use **order-indicating properties** (one property per position)
Supported by “RDF Containers” (`rdf:_1`, `rdf:_2`, ..., `rdfs:ContainerMembershipProperty`)
Pro: direct access; length only detectable by absence of triples
Con: list structure requires external knowledge; harder insert/delete
- Give up order and **represent lists as sets**
Easy to do in RDF
Pro: simple; may suffice for many use cases
Con: no order
- Other encodings are conceivable ...

Representing order

Problem: RDF can easily represent sets, but not lists

Possible solutions:

- 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)
- Use **order-indicating properties** (one property per position)
Supported by “RDF Containers” (`rdf:_1`, `rdf:_2`, ..., `rdfs:ContainerMembershipProperty`)
Pro: direct access; length only detectable by absence of triples
Con: list structure requires external knowledge; harder insert/delete
- Give up order and **represent lists as sets**
Easy to do in RDF
Pro: simple; may suffice for many use cases
Con: no order
- Other encodings are conceivable ...

Possible compromise: Combine several encodings within one graph

(increased data volume – easier to read – harder to write)

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 3.1: 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 3.2: 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 3.2: 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 3.3: Count all triples in the database:

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

Basic SPARQL by example

Example 3.2: 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 3.3: Count all triples in the database:

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

Example 3.4: Count all predicates in the database:

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

Basic SPARQL by example (2)

Example 3.5: 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 3.5: 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 3.6: Find pairs of siblings:

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

Basic SPARQL by example (2)

Example 3.5: 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 3.6: 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 3.7: 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 3.7: 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 3.8: 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 3.9: A **triple pattern** is a triple $\langle s, p, o \rangle$, where s and o are arbitrary RDF terms^a or variables, and p is an IRI or variable. A **basic graph pattern (BGP)** is a set of triple patterns.

^aCuriously, 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 3.9: A **triple pattern** is a triple $\langle s, p, o \rangle$, where s and o are arbitrary RDF terms^a or variables, and p is an IRI or variable. A **basic graph pattern** (BGP) is a set of triple patterns.

^aCuriously, 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 3.10: 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 3.10: 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 3.11: 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 **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 nodes!):

```
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 nodes!):

```
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 3.11: 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 **eval $_G(P)$** , where we omit G if clear from the context.

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

Boolean queries

Definition 3.11: 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 **eval $_G(P)$** , where we omit G if clear from the context.

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

A: The empty multiset \emptyset of solutions!

Boolean queries

Definition 3.11: 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 $\text{eval}_G(\text{eg:s} \quad \text{eg:p} \quad \text{eg:o})$ over the empty graph $G = \emptyset$?

A: The empty multiset \emptyset of solutions!

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

Boolean queries

Definition 3.11: 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 **eval $_G(P)$** , where we omit G if clear from the context.

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

A: The empty multiset \emptyset of solutions!

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

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

Boolean queries

Definition 3.11: 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 $\text{eval}_G(\text{eg:s } \text{eg:p } \text{eg:o})$ over the empty graph $G = \emptyset$?

A: The empty multiset \emptyset of solutions!

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

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