TECHNISCHE UNIVERSITÄT DRESDEN

KNOWLEDGE GRAPHS

Lecture 4: Introduction to SPARQL

Markus Krötzsch

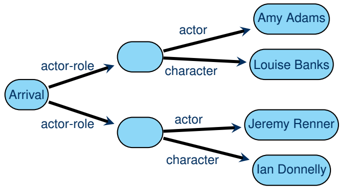
Knowledge-Based Systems

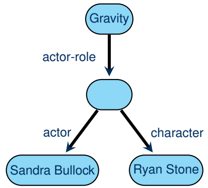
TU Dresden, 6th Nov 2018

Review

We can use reification to encode complex structures in RDF graphs:

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





We can also encode lists:

- Linked lists (e.g., RDF collections)
- Array-style representations with one property per position (e.g., RDF containers)
- as sets (give up order)

But not all data structures are a natural fit for RDF graphs (and not for other knowledge graph either)

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 4.1: 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 4.2: 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 4.2: 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 4.3: Count all triples in the database:

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

Basic SPARQL by example

Example 4.2: 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 4.3: Count all triples in the database:

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

Example 4.4: Count all predicates in the database:

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

Basic SPARQL by example (2)

Example 4.5: 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 4.5: 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 4.6: Find pairs of siblings:

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

Basic SPARQL by example (2)

Example 4.5: 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 4.6: 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 4.7: 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 4.7: 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 4.8: 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 4.9: A triple pattern is a triple $\langle \mathrm{s}, \mathrm{p}, \mathrm{o} \rangle$, 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.

[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 4.9: A triple pattern is a triple $\langle \mathrm{s}, \mathrm{p}, \mathrm{o} \rangle$, 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.

[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 4.10: 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 4.10: 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 4.11: 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.

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 (and triple pattern) `?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 4.11: 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 $\text{eval}_G($`eg:s   eg:p   eg:o`$)$ over the empty graph $G = \emptyset$?

Boolean queries

Definition 4.11: 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 $\text{eval}_G(\texttt{eg:s eg:p eg:o})$ over the empty graph $G = \emptyset$?

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

Boolean queries

Definition 4.11: 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 $\text{eval}_G(\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o})$ 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 4.11: 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 $\text{eval}_G(\texttt{eg:s}\quad\texttt{eg:p}\quad\texttt{eg:o})$ 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 4.11: 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 $\text{eval}_G(\texttt{eg:s eg:p eg:o})$ over the empty graph $G = \emptyset$?

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

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

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”).

Finding BGP solutions

How hard is it to compute solutions to BGPs?

Observation: It is easy to check if a given mapping of bnodes and variables produces a solution:

- Simply verify that the mapped triples are contained in the given graph
- Can be done in quadratic time (# triples in pattern $\times$ # edges in graph)

Finding BGP solutions

How hard is it to compute solutions to BGPs?

Observation: It is easy to check if a given mapping of bnodes and variables produces a solution:

- Simply verify that the mapped triples are contained in the given graph
- Can be done in quadratic time (# triples in pattern $\times$ # edges in graph)

In other words: the problem (as a decision problem) is in NP. It turns out, this is the best we can do:

Theorem 4.12: Determining if a BGP has solution mappings over a graph is NP-complete.

Finding BGP solutions

How hard is it to compute solutions to BGPs?

Observation: It is easy to check if a given mapping of bnodes and variables produces a solution:

- Simply verify that the mapped triples are contained in the given graph
- Can be done in quadratic time (# triples in pattern $\times$ # edges in graph)

In other words: the problem (as a decision problem) is in NP. It turns out, this is the best we can do:

Theorem 4.12: Determining if a BGP has solution mappings over a graph is NP-complete.

Proof: Inclusion: guess mapping for bnodes and variables; check if guess was correct.

Hardness: by reduction from 3-colourability of graphs

From 3-colourability to BGP matching

The problem of graph 3-colourability (**3Col**) is defined as follows:
Given: An undirected graph G
Question: Can the vertices of G be assigned colours red, green and blue so that no two adjacent vertices have the same colour?

It is known that this problem is NP-complete (and in particular NP-hard).

From 3-colourability to BGP matching

The problem of graph 3-colourability (**3Col**) is defined as follows:
Given: An undirected graph G
Question: Can the vertices of G be assigned colours red, green and blue so that no two adjacent vertices have the same colour?

It is known that this problem is NP-complete (and in particular NP-hard).

We can find a polynomial many-one reduction from **3Col** to BGP matching:

- A given graph G is mapped to a BGP P_G by introducing, for each undirected edge e−f in G, two triples `?e <edge> ?f` and `?f <edge> ?e`.
- We consider the RDF graph C given by

```
<red> <edge> <green>, <blue> .
<green> <edge> <red>, <blue> .
<blue> <edge> <green>, <red> .
```

Then P_G has a solution mapping over C if and only if G is 3-colourable. □

Finding BGP solutions using joins

Real graph databases do not find solutions by guessing mappings until they found one that works: they retrieve solutions for triple patterns and combine them with joins.

Finding BGP solutions using joins

Real graph databases do not find solutions by guessing mappings until they found one that works: they retrieve solutions for triple patterns and combine them with joins.

Definition 4.13: Two solution mappings μ_1 and μ_2 are compatible if $\mu_1(x) = \mu_2(x)$ for all variable names $x \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$, where dom is the domain on which a (partial) function is defined. In this case, $\mu_1 \uplus \mu_2$ is the mapping defined as

$$\mu_1 \uplus \mu_2(x) = \begin{cases} \mu_1(x) & \text{if } x \in \mathrm{dom}(\mu_1) \\ \mu_2(x) & \text{if } x \in \mathrm{dom}(\mu_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Finding BGP solutions using joins

Real graph databases do not find solutions by guessing mappings until they found one that works: they retrieve solutions for triple patterns and combine them with joins.

Definition 4.13: Two solution mappings μ_1 and μ_2 are compatible if $\mu_1(x) = \mu_2(x)$ for all variable names $x \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$, where dom is the domain on which a (partial) function is defined. In this case, $\mu_1 \uplus \mu_2$ is the mapping defined as

$$\mu_1 \uplus \mu_2(x) = \begin{cases} \mu_1(x) & \text{if } x \in \mathrm{dom}(\mu_1) \\ \mu_2(x) & \text{if } x \in \mathrm{dom}(\mu_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 4.14: The join of two multisets Ω_1 and Ω_2 of solution mappings is the multiset $\mathrm{Join}(\Omega_1, \Omega_2) = \{\mu_1 \uplus \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \text{ and } \mu_1 \text{ and } \mu_2 \text{ are compatible}\}$.

The multiplicity $\mathrm{card}_\Omega(\mu)$ of each solution $\mu \in \Omega = \mathrm{Join}(\Omega_1, \Omega_2)$ is given as $\mathrm{card}_\Omega(\mu) = \sum_{\mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \uplus \mu_2 = \mu} \mathrm{card}_{\Omega_1}(\mu_1) \times \mathrm{card}_{\Omega_2}(\mu_2)$.

Finding BGP solutions using joins

Theorem 4.15: Let G be an RDF graph, and let $P = P_1 \cup P_2$ be a bnode-free BGP that is a disjoint union of two BGPs P_1 and P_2. Then $\text{eval}_G(P) = \text{Join}(\text{eval}_G(P_1), \text{eval}_G(P_2))$. Therefore, $\text{eval}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.

Finding BGP solutions using joins

Theorem 4.15: Let G be an RDF graph, and let $P = P_1 \cup P_2$ be a bnode-free BGP that is a disjoint union of two BGPs P_1 and P_2. Then $\text{eval}_G(P) = \text{Join}(\text{eval}_G(P_1), \text{eval}_G(P_2))$. Therefore, $\text{eval}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.

Proof: Since P contains no bnodes, solutions are defined without considering mappings "σ" and the multiplicity of any solution will therefore be 1.

Finding BGP solutions using joins

Theorem 4.15: Let G be an RDF graph, and let $P = P_1 \cup P_2$ be a bnode-free BGP that is a disjoint union of two BGPs P_1 and P_2. Then $\text{eval}_G(P) = \text{Join}(\text{eval}_G(P_1), \text{eval}_G(P_2))$. Therefore, $\text{eval}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.

Proof: Since P contains no bnodes, solutions are defined without considering mappings "σ" and the multiplicity of any solution will therefore be 1.

"$\subseteq$" Consider $\mu \in \text{eval}_G(P)$.

- Let μ_i be the restriction of μ to variables in P_i $(i = 1, 2)$
- Then $\mu_i \in \text{eval}_G(P_i)$ and μ_1 and μ_2 are compatible
- Therefore $\mu_1 \uplus \mu_2 = \mu \in \text{Join}(\text{eval}_G(P_1), \text{eval}_G(P_2))$

Finding BGP solutions using joins

Theorem 4.15: Let G be an RDF graph, and let $P = P_1 \cup P_2$ be a bnode-free BGP that is a disjoint union of two BGPs P_1 and P_2. Then $\text{eval}_G(P) = \text{Join}(\text{eval}_G(P_1), \text{eval}_G(P_2))$. Therefore, $\text{eval}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.

Proof: Since P contains no bnodes, solutions are defined without considering mappings "σ" and the multiplicity of any solution will therefore be 1.

"$\subseteq$" Consider $\mu \in \text{eval}_G(P)$.

- Let μ_i be the restriction of μ to variables in P_i $(i = 1, 2)$
- Then $\mu_i \in \text{eval}_G(P_i)$ and μ_1 and μ_2 are compatible
- Therefore $\mu_1 \uplus \mu_2 = \mu \in \text{Join}(\text{eval}_G(P_1), \text{eval}_G(P_2))$

"$\supseteq$" Consider $\mu \in \text{Join}(\text{eval}_G(P_1), \text{eval}_G(P_2))$.

- Then there are compatible $\mu_i \in \text{eval}_G(P_i)$ with $\mu_1 \uplus \mu_2 = \mu$
- By construction, $\mu_1(P_1) = \mu(P_1) \subseteq G$ and $\mu_2(P_2) = \mu(P_2) \subseteq G$
- Hence $\mu_1(P_1) \cup \mu_2(P_2) = \mu(P_1) \cup \mu(P_2) = \mu(P_1 \cup P_2) \subseteq G$, as claimed

Finding BGP solutions using joins

Theorem 4.15: Let G be an RDF graph, and let $P = P_1 \cup P_2$ be a bnode-free BGP that is a disjoint union of two BGPs P_1 and P_2. Then $\text{eval}_G(P) = \text{Join}(\text{eval}_G(P_1), \text{eval}_G(P_2))$. Therefore, $\text{eval}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.

Proof: Since P contains no bnodes, solutions are defined without considering mappings "σ" and the multiplicity of any solution will therefore be 1.

"$\subseteq$" Consider $\mu \in \text{eval}_G(P)$.

- Let μ_i be the restriction of μ to variables in P_i $(i = 1, 2)$
- Then $\mu_i \in \text{eval}_G(P_i)$ and μ_1 and μ_2 are compatible
- Therefore $\mu_1 \uplus \mu_2 = \mu \in \text{Join}(\text{eval}_G(P_1), \text{eval}_G(P_2))$

"$\supseteq$" Consider $\mu \in \text{Join}(\text{eval}_G(P_1), \text{eval}_G(P_2))$.

- Then there are compatible $\mu_i \in \text{eval}_G(P_i)$ with $\mu_1 \uplus \mu_2 = \mu$
- By construction, $\mu_1(P_1) = \mu(P_1) \subseteq G$ and $\mu_2(P_2) = \mu(P_2) \subseteq G$
- Hence $\mu_1(P_1) \cup \mu_2(P_2) = \mu(P_1) \cup \mu(P_2) = \mu(P_1 \cup P_2) \subseteq G$, as claimed $\square$

Finding BGP solutions . . . in practice

Theorem 4.15 does not work if the patterns contains blank nodes! (see exercise)

In practice, we can treat bnodes like variables that are projected away later on (leading to increased multiplicities).

Finding BGP solutions . . . in practice

Theorem 4.15 does not work if the patterns contains blank nodes! (see exercise)

In practice, we can treat bnodes like variables that are projected away later on (leading to increased multiplicities).

Real graph databases compute joins in highly optimised ways:

- Efficient data structures for finding compatible solutions to triple patterns (e.g., hash maps, tries, ordered lists, . . .)
- Query planners for optimising order of joins (goal: small intermediate results)
- Streaming joins: returning first results before join is complete
- Sometimes: multi-way joins (joining more than two triple patterns at once)

. . . but they still compute BGP solutions by joining partial solutions and hoping for an overall match

In the worst case, any known algorithm needs exponential time.

Outline of plan

So far we have:

- seen some examples of simple SPARQL queries,
- introduced the syntax for basic graph patterns (BGPs),
- defined the semantics of BGPs and the complexity of computing it

SPARQL includes many further features . . . (too many to discuss all in detail).

Here is the plan:

1. Property paths: making connections through graphs
2. Filters: expressing further conditions on terms and triples
3. Selection and solution set modifiers: changing the output
4. Everything else: union, optional, bind, values, subqueries, aggregates, . . .

Summary

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

Basic Graph Pattern matching is already rather complex (NP-complete), but can be implemented using joins

What's next?

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