Federation and Navigation in SPARQL 1.1

Jorge Pérez

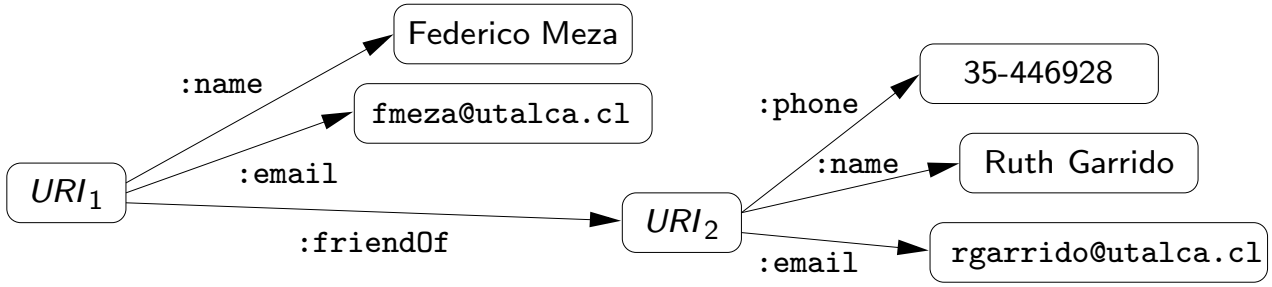
Assistant Professor
Department of Computer Science
Universidad de Chile

Outline

SPARQL query language for RDF

SPARQL query language for RDF

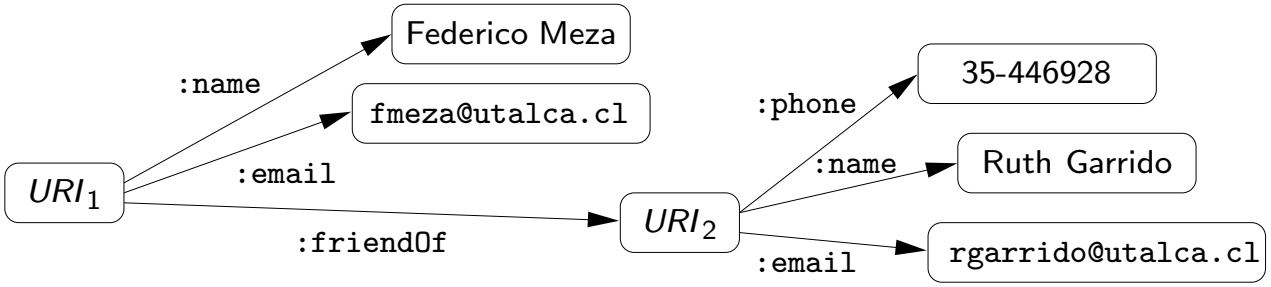
RDF Graph:



RDF-triples: (URI_2, `:email`, `rgarrido@utalca.cl`)

SPARQL query language for RDF

RDF Graph:



RDF-triples: (URI_2, `:email`, `rgarrido@utalca.cl`)

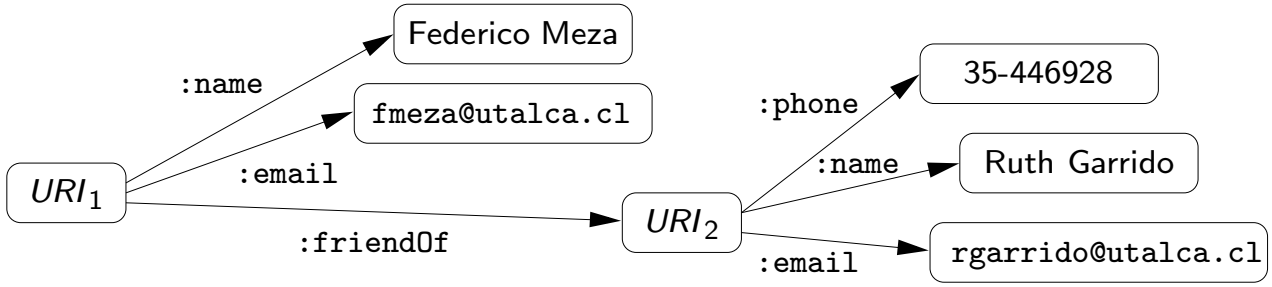
SPARQL Query:

```
SELECT ?N
WHERE
{
  ?X :name ?N .


}
```

SPARQL query language for RDF

RDF Graph:



RDF-triples: (URI_2, `:email`, `rgarrido@utalca.cl`)

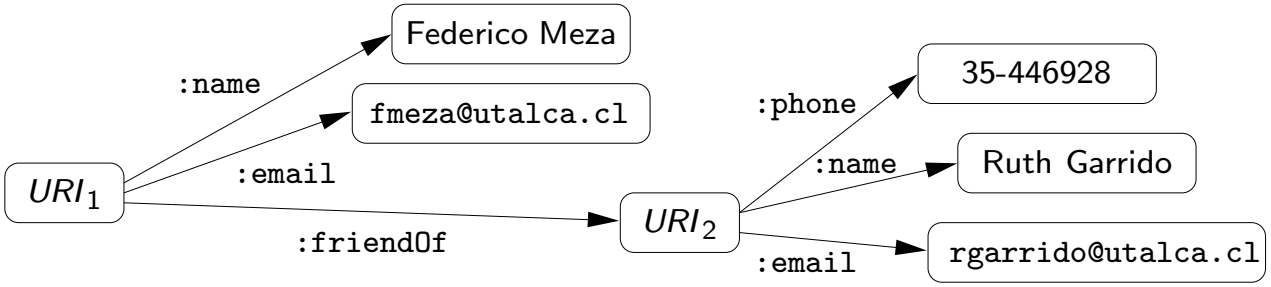
SPARQL Query:

```
SELECT ?N ?E
WHERE
{
  ?X :name ?N .
  ?X :email ?E .

}
```

SPARQL query language for RDF

RDF Graph:



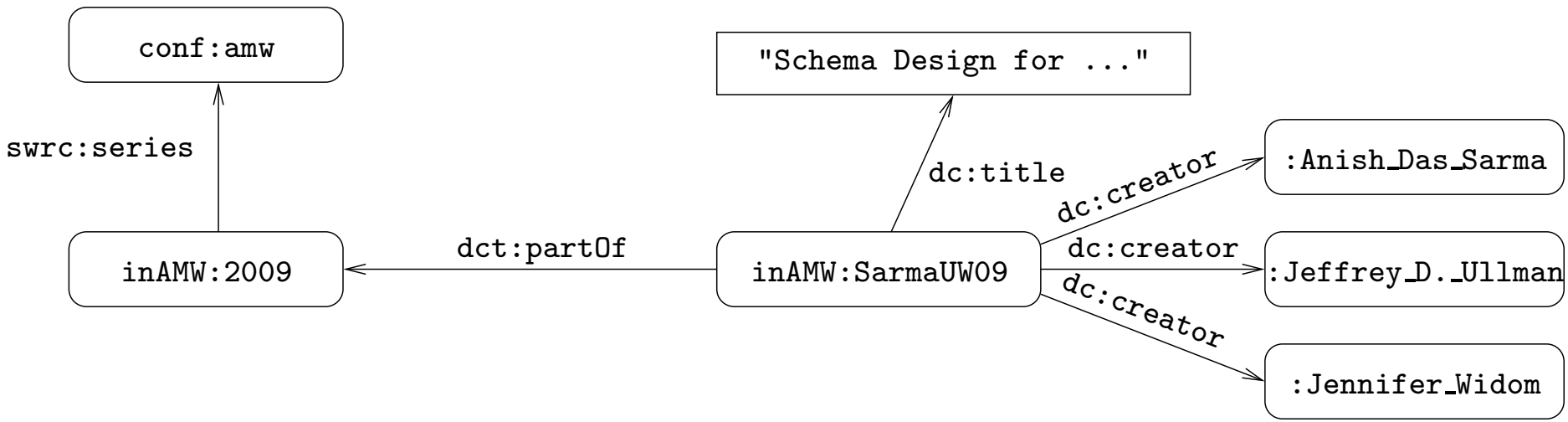
RDF-triples: (URI_2, `:email`, `rgarrido@utalca.cl`)

SPARQL Query:

```
SELECT ?N ?E
WHERE
{
  ?X :name ?N .
  ?X :email ?E .
  ?X :friendOf ?Y . ?Y :name "Ruth Garrido"
}
```

An example of an RDF graph to query: DBLP

```
     : <http://dblp.l3s.de/d2r/resource/authors/>
  conf: <http://dblp.l3s.de/d2r/resource/conferences/>
 inAMW: <http://dblp.l3s.de/d2r/resource/publications/conf/amw/>
  swrc: <http://swrc.ontoware.org/ontology#>
    dc: <http://purl.org/dc/elements/1.1/>
   dct: <http://purl.org/dc/terms/>
```



SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
```

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{



}
```

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{
  ?Paper      dc:creator     ?Author .


}
```

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .

}
```

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .
  ?Conf       swrc:series    conf:iswc .
}
```

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .
  ?Conf       swrc:series    conf:iswc .
}
```

A SPARQL query consists of a:

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .
  ?Conf       swrc:series    conf:iswc .
}
```

A SPARQL query consists of a:

Head: Processing of the variables

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC

```
SELECT ?Author
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .
  ?Conf       swrc:series    conf:iswc .
}
```

A SPARQL query consists of a:

- Head: Processing of the variables
- Body: Pattern matching expression

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC, and their Web pages if this information is available:

```
SELECT ?Author ?WebPage
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .
  ?Conf       swrc:series    conf:iswc .

  OPTIONAL {
      ?Author  foaf:homePage  ?WebPage . }
}
```

SPARQL: A Simple RDF Query Language

Example: Authors that have published in ISWC, and their Web pages if this information is available:

```
SELECT ?Author ?WebPage
WHERE
{
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:partOf     ?Conf .
  ?Conf       swrc:series    conf:iswc .

  OPTIONAL {
      ?Author  foaf:homePage  ?WebPage . }
}
```

But things can become more complex...

Interesting features of pattern matching on graphs

```
SELECT ?X1 ?X2 ...
  { P1 .
    P2 }
```

But things can become more complex...

Interesting features of pattern matching on graphs

- Grouping

```
SELECT ?X1 ?X2 ...
 {{ P1 .
    P2 }

  { P3 .
    P4 }

 }
```

But things can become more complex...

Interesting features of pattern matching on graphs

- Grouping
- Optional parts

```
SELECT ?X1 ?X2 ...
 {{ P1 .
    P2
    OPTIONAL { P5 }  }

  { P3 .
    P4
    OPTIONAL { P7 }  }

 }
```

But things can become more complex...

Interesting features of pattern matching on graphs

- Grouping
- Optional parts
- Nesting

```
SELECT ?X1 ?X2 ...
 {{ P1 .
    P2
    OPTIONAL { P5 }  }

  { P3 .
    P4
    OPTIONAL { P7
      OPTIONAL { P8 }  }  }
 }
```

But things can become more complex...

Interesting features of pattern matching on graphs

- Grouping
- Optional parts
- Nesting
- Union of patterns

```
SELECT ?X1 ?X2 ...
{{{ P1 .
    P2
    OPTIONAL { P5 }  }

  { P3 .
    P4
    OPTIONAL { P7
      OPTIONAL { P8 }  }  }
 }
 UNION
 { P9 }}
```

But things can become more complex...

Interesting features of pattern matching on graphs

- Grouping
- Optional parts
- Nesting
- Union of patterns
- Filtering
- ...
- + several new features in the upcoming version: federation, navigation

```
SELECT ?X1 ?X2 ...
{{{ P1 .
    P2
    OPTIONAL { P5 }  }

  { P3 .
    P4
    OPTIONAL { P7
      OPTIONAL { P8 }  }  }
 }
 UNION
 { P9
  FILTER ( R ) }}
```

But things can become more complex...

Interesting features of pattern matching on graphs

- Grouping
- Optional parts
- Nesting
- Union of patterns
- Filtering
- ...
- + several new features in the upcoming version: federation, navigation

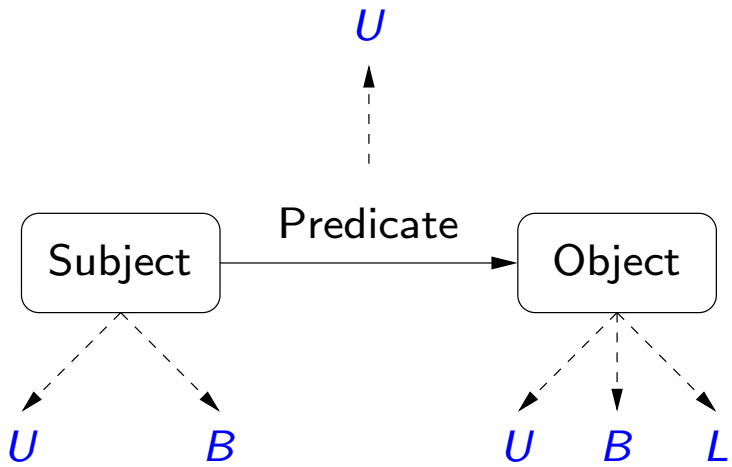
```
SELECT ?X1 ?X2 ...
{{{ P1 .
    P2
    OPTIONAL { P5 }  }

  { P3 .
    P4
    OPTIONAL { P7
      OPTIONAL { P8 }  }  }
 }
 UNION
 { P9
  FILTER ( R ) }}
```

What is the (formal) *meaning* of a general SPARQL query?

Outline

RDF triples and graphs

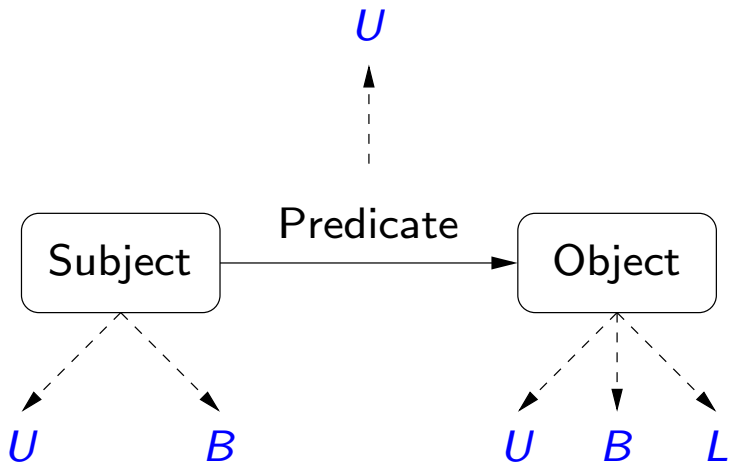


U : set of URIs

B : set of blank nodes

L : set of literals

RDF triples and graphs



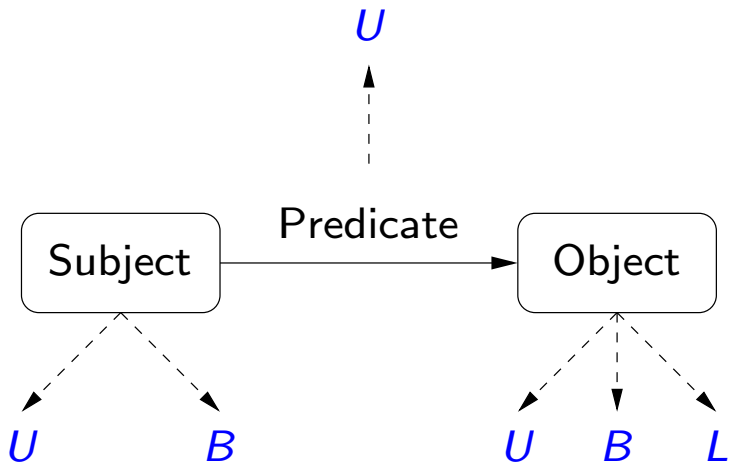
U : set of URIs

B : set of blank nodes

L : set of literals

$(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ is called an RDF triple

RDF triples and graphs



U : set of URIs

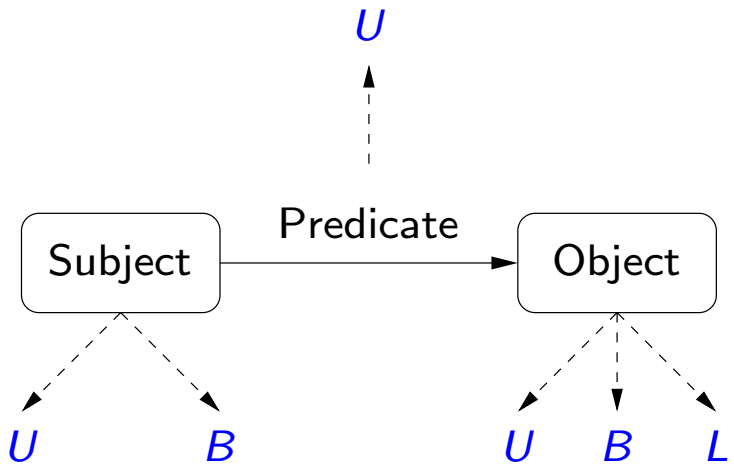
B : set of blank nodes

L : set of literals

$(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ is called an RDF triple

A set of RDF triples is called an RDF graph

RDF triples and graphs



U : set of URIs

B : set of blank nodes

L : set of literals

$(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ is called an RDF triple

A set of RDF triples is called an RDF graph

In this talk, we do not consider blank nodes

- $(s, p, o) \in U \times U \times (U \cup L)$ is called an RDF triple

A standard algebraic syntax

- Triple patterns: just RDF triples + variables (from a set V)

```
?X :name "john"
```

$(?X, \text{name}, \text{john})$

A standard algebraic syntax

- Triple patterns: just RDF triples + variables (from a set V)

```
?X :name "john"
```

$(?X, \text{name}, \text{john})$

- Graph patterns: full parenthesized algebra

original SPARQL syntax

```
{  P1 .  P2  }
```

algebraic syntax

$(\ P_1 \text{ AND } P_2\ )$

A standard algebraic syntax

- Triple patterns: just RDF triples + variables (from a set V)

```
?X :name "john"
```

$(?X, \text{name}, \text{john})$

- Graph patterns: full parenthesized algebra

original SPARQL syntax

```
{  P1 .  P2  }
```

$(\ P_1 \text{ AND } P_2\ )$

```
{ P1 OPTIONAL { P2 }}
```

$(\ P_1 \text{ OPT } P_2\ )$

A standard algebraic syntax

- Triple patterns: just RDF triples + variables (from a set V)

`?X :name "john"`	$(?X, \text{name}, \text{john})$

- Graph patterns: full parenthesized algebra

original SPARQL syntax	algebraic syntax
`{ P1 . P2 }`	$(P_1 \text{ AND } P_2)$
`{ P1 OPTIONAL { P2 }}`	$(P_1 \text{ OPT } P_2)$
`{ P1 } UNION { P2 }`	$(P_1 \text{ UNION } P_2)$

A standard algebraic syntax

- Triple patterns: just RDF triples + variables (from a set V)

`?X :name "john"`	$(?X, \text{name}, \text{john})$

- Graph patterns: full parenthesized algebra

original SPARQL syntax	algebraic syntax
`{ P1 . P2 }`	$(P_1 \text{ AND } P_2)$
`{ P1 OPTIONAL { P2 }}`	$(P_1 \text{ OPT } P_2)$
`{ P1 } UNION { P2 }`	$(P_1 \text{ UNION } P_2)$
`{ P1 FILTER ( R ) }`	$(P_1 \text{ FILTER } R)$

A standard algebraic syntax (cont.)

- Explicit precedence/association

Example

```
{ t1
  t2
  OPTIONAL { t3 }
  OPTIONAL { t4 }
  t5
}
```

$$((((t_1 \text{ AND } t_2) \text{ OPT } t_3) \text{ OPT } t_4) \text{ AND } t_5)$$

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \quad V \rightarrow U \cup L$$

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \quad V \rightarrow U \cup L$$

Given a mapping μ and a triple pattern t:

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \quad V \rightarrow U \cup L$$

Given a mapping μ and a triple pattern t:

- $\mu(t)$: triple obtained from t replacing variables according to μ

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \quad V \rightarrow U \cup L$$

Given a mapping μ and a triple pattern t:

- $\mu(t)$: triple obtained from t replacing variables according to μ

Example

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \quad V \rightarrow U \cup L$$

Given a mapping μ and a triple pattern t:

- $\mu(t)$: triple obtained from t replacing variables according to μ

Example

$$\mu = \{?X \rightarrow R_1, ?Y \rightarrow R_2, ?Name \rightarrow \text{john}\}$$

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \quad V \rightarrow U \cup L$$

Given a mapping μ and a triple pattern t:

- $\mu(t)$: triple obtained from t replacing variables according to μ

Example

$$\mu = \{?X \rightarrow R_1, ?Y \rightarrow R_2, ?Name \rightarrow \text{john}\}$$

$$t = (?X, \text{name}, ?Name)$$

Mappings: building block for the semantics

Definition

A mapping is a partial function from variables to RDF terms

$$\mu : \quad V \rightarrow U \cup L$$

Given a mapping μ and a triple pattern t:

- $\mu(t)$: triple obtained from t replacing variables according to μ

Example

$$\mu = \{?X \rightarrow R_1, ?Y \rightarrow R_2, ?Name \rightarrow \text{john}\}$$

$$t = (?X, \text{name}, ?Name)$$

$$\mu(t) = (R_1, \text{name}, \text{john})$$

The semantics of triple patterns

Definition

The evaluation of triple patter t over a graph G, denoted by $[\![t]\!]_G$, is the set of all mappings μ such that:

The semantics of triple patterns

Definition

The evaluation of triple patter t over a graph G, denoted by $[\![t]\!]_G$, is the set of all mappings μ such that:

- $\text{dom}(\mu)$ is exactly the set of variables occurring in t

The semantics of triple patterns

Definition

The evaluation of triple patter t over a graph G, denoted by $[\![t]\!]_G$, is the set of all mappings μ such that:

- $\text{dom}(\mu)$ is exactly the set of variables occurring in t
- $\mu(t) \in G$

Example

$$G$$

$$(R_1, \text{name}, \text{john})$$
$$(R_1, \text{email}, \text{J@ed.ex})$$
$$(R_2, \text{name}, \text{paul})$$

$$[\![(?X, \text{name}, ?N)]\!]_G$$

Example

$$G$$

$$(R_1, \text{name}, \text{john})$$
$$(R_1, \text{email}, \text{J@ed.ex})$$
$$(R_2, \text{name}, \text{paul})$$

$$[\![(?X, \text{name}, ?N)]\!]_G$$

$$\left\{ \begin{array}{l} \mu_1 = \{?X \to R_1, ?N \to \text{john}\} \\ \mu_2 = \{?X \to R_2, ?N \to \text{paul}\} \end{array} \right\}$$

Example

$$G$$

$$(R_1, \text{name}, \text{john})$$
$$(R_1, \text{email}, \text{J@ed.ex})$$
$$(R_2, \text{name}, \text{paul})$$

$$[\![(?X, \text{name}, ?N)]\!]_G$$

$$\left\{ \begin{array}{l} \mu_1 = \{?X \to R_1, ?N \to \text{john}\} \\ \mu_2 = \{?X \to R_2, ?N \to \text{paul}\} \end{array} \right\}$$

$$[\![(?X, \text{email}, ?E)]\!]_G$$

Example

$$G$$

$$(R_1, \text{name}, \text{john})$$
$$(R_1, \text{email}, \text{J@ed.ex})$$
$$(R_2, \text{name}, \text{paul})$$

$$[\![(?X, \text{name}, ?N)]\!]_G$$

$$\left\{ \begin{array}{l} \mu_1 = \{?X \to R_1, ?N \to \text{john}\} \\ \mu_2 = \{?X \to R_2, ?N \to \text{paul}\} \end{array} \right\}$$

$$[\![(?X, \text{email}, ?E)]\!]_G$$

$$\left\{ \mu = \{?X \to R_1, ?E \to \text{J@ed.ex}\} \right\}$$

Example

G

$(R_1, \text{name}, \text{john})$
$(R_1, \text{email}, \text{J@ed.ex})$
$(R_2, \text{name}, \text{paul})$

$[\![(?X, \text{name}, ?N)]\!]_G$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul

$[\![(?X, \text{email}, ?E)]\!]_G$

	$?X$	$?E$
μ	R_1	J@ed.ex

Example

$$G$$
$$(R_1, \text{name}, \text{john})$$
$$(R_1, \text{email}, \text{J@ed.ex})$$
$$(R_2, \text{name}, \text{paul})$$

$[\![(R_1, \text{webPage}, ?W)]\!]_G$

$[\![(R_3, \text{name}, \text{ringo})]\!]_G$

$[\![(R_2, \text{name}, \text{paul})]\!]_G$

Example

$$G$$
$$(R_1,\ \text{name},\ \text{john})$$
$$(R_1,\ \text{email},\ \text{J@ed.ex})$$
$$(R_2,\ \text{name},\ \text{paul})$$

$$[\![(R_1, \text{webPage}, ?W)]\!]_G$$
$$\{\quad\}$$
$$[\![(R_2, \text{name}, \text{paul})]\!]_G$$

$$[\![(R_3, \text{name}, \text{ringo})]\!]_G$$

Example

$$G$$
$$(R_1,\ \text{name},\ \text{john})$$
$$(R_1,\ \text{email},\ \text{J@ed.ex})$$
$$(R_2,\ \text{name},\ \text{paul})$$

$$[\![(R_1, \text{webPage}, ?W)]\!]_G$$
$$\{\quad\}$$

$$[\![(R_2, \text{name}, \text{paul})]\!]_G$$

$$[\![(R_3, \text{name}, \text{ringo})]\!]_G$$
$$\{\quad\}$$

Example

$$G$$
$$(R_1,\ \text{name},\ \text{john})$$
$$(R_1,\ \text{email},\ \text{J@ed.ex})$$
$$(R_2,\ \text{name},\ \text{paul})$$

$$[\![(R_1, \text{webPage}, ?W)]\!]_G$$
$$\{\quad\}$$

$$[\![(R_2, \text{name}, \text{paul})]\!]_G$$
$$\{\ \mu_\emptyset = \{\ \}\ \}$$

$$[\![(R_3, \text{name}, \text{ringo})]\!]_G$$
$$\{\quad\}$$

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are *compatibles* iff they agree in their *shared variables*:

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are *compatibles* iff they agree in their *shared variables*:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are *compatibles* iff they agree in their *shared variables*:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are *compatibles* iff they agree in their *shared variables*:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are *compatibles* iff they agree in their *shared variables*:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are *compatibles* iff they agree in their *shared variables*:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2
$\mu_1 \cup \mu_2$	R_1	*john*	*J@edu.ex*	

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are *compatibles* iff they agree in their *shared variables*:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2
$\mu_1 \cup \mu_2$	R_1	*john*	*J@edu.ex*	

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are *compatibles* iff they agree in their *shared variables*:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2
$\mu_1 \cup \mu_2$	R_1	*john*	*J@edu.ex*	
$\mu_1 \cup \mu_3$	R_1	*john*	*P@edu.ex*	R_2

Compatible mappings: mappings that can be merged.

Definition

The mappings μ_1, μ_2 are *compatibles* iff they agree in their *shared variables*:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2
$\mu_1 \cup \mu_2$	R_1	*john*	*J@edu.ex*	
$\mu_1 \cup \mu_3$	R_1	*john*	*P@edu.ex*	R_2

$\mu_\emptyset = \{\ \}$ is compatible with every mapping.

Sets of mappings and operations

Let M_1 and M_2 be sets of mappings:

Definition

Join: $M_1 \bowtie M_2$

- $\{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1, \mu_2 \in M_2,$ and μ_1, μ_2 are compatibles$\}$
- extending mappings in M_1 with compatible mappings in M_2

will be used to define AND

Sets of mappings and operations

Let M_1 and M_2 be sets of mappings:

Definition

Join: $M_1 \bowtie M_2$

- $\{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1, \mu_2 \in M_2,$ and μ_1, μ_2 are compatibles$\}$
- extending mappings in M_1 with compatible mappings in M_2

will be used to define AND

Definition

Union: $M_1 \cup M_2$

- $\{\mu \mid \mu \in M_1$ or $\mu \in M_2\}$
- mappings in M_1 plus mappings in M_2 (the usual set union)

will be used to define UNION

Sets of mappings and operations

Definition

Difference: $M_1 \setminus M_2$

- $\{\mu \in M_1 \mid$ for all $\mu' \in M_2$, μ and μ' are not compatibles$\}$
- mappings in M_1 that cannot be extended with mappings in M_2

Sets of mappings and operations

Definition

Difference: $M_1 \setminus M_2$

- $\{\mu \in M_1 \mid$ for all $\mu' \in M_2$, μ and μ' are not compatibles$\}$
- mappings in M_1 that cannot be extended with mappings in M_2

Definition

Left outer join: $M_1 \mathbin{⟕} M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

- extension of mappings in M_1 with compatible mappings in M_2
- plus the mappings in M_1 that cannot be extended.

will be used to define OPT

Semantics of general graph patterns

Definition

Given a graph G the evaluation of a pattern is recursively defined

the base case is the evaluation of a triple pattern.

Semantics of general graph patterns

Definition

Given a graph G the evaluation of a pattern is recursively defined

- $[\![(P_1 \text{ AND } P_2)]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$

the base case is the evaluation of a triple pattern.

Semantics of general graph patterns

Definition

Given a graph G the evaluation of a pattern is recursively defined

- $[\![(P_1 \text{ AND } P_2)]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$
- $[\![(P_1 \text{ UNION } P_2)]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$

the base case is the evaluation of a triple pattern.

Semantics of general graph patterns

Definition

Given a graph G the evaluation of a pattern is recursively defined

- $[\![(P_1 \text{ AND } P_2)]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$
- $[\![(P_1 \text{ UNION } P_2)]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$
- $[\![(P_1 \text{ OPT } P_2)]\!]_G = [\![P_1]\!]_G \mathbin{⟕} [\![P_2]\!]_G$

the base case is the evaluation of a triple pattern.

Example (AND)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E))]\!]_G$$

Example (AND)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \bowtie [\![(?X, \text{email}, ?E)]\!]_G$$

Example (AND)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \bowtie [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

Example (AND)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \bowtie [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

	$?X$	$?E$
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

Example (AND)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \bowtie [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

$\bowtie$

	$?X$	$?E$
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

Example (AND)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \bowtie [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

$\bowtie$

	$?X$	$?E$
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

	$?X$	$?N$	$?E$
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex

Example (OPT)

G :

$(R_1, \text{name}, \text{john})$	$(R_2, \text{name}, \text{paul})$	$(R_3, \text{name}, \text{ringo})$
$(R_1, \text{email}, \text{J@ed.ex})$		$(R_3, \text{email}, \text{R@ed.ex})$
		$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E))]\!]_G$$

Example (OPT)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \text{ ⟕} [\![(?X, \text{email}, ?E)]\!]_G$$

Example (OPT)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \text{ ⟕} [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

Example (OPT)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \text{ ⟕ } [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

	$?X$	$?E$
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

Example (OPT)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \text{ ⟕ } [\![(?X, \text{email}, ?E)]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

⟕

	$?X$	$?E$
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

Example (OPT)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \text{ ⟕ } [\![(?X, \text{email}, ?E)]\!]_G$$

	?X	?N
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

⟕

	?X	?E
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

	?X	?N	?E
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

Example (OPT)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E))]\!]_G$$

$$[\![(?X, \text{name}, ?N)]\!]_G \bowtie [\![(?X, \text{email}, ?E)]\!]_G$$

	?X	?N
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

$\bowtie$

	?X	?E
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

	?X	?N	?E
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

Example (UNION)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{email}, ?\mathit{Info}) \text{ UNION } (?X, \text{webPage}, ?\mathit{Info}))]\!]_G$$

Example (UNION)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{email}, ?\mathit{Info}) \text{ UNION } (?X, \text{webPage}, ?\mathit{Info}))]\!]_G$$

$$[\![(?X, \text{email}, ?\mathit{Info})]\!]_G \cup [\![(?X, \text{webPage}, ?\mathit{Info})]\!]_G$$

Example (UNION)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{email}, ?\mathit{Info}) \text{ UNION } (?X, \text{webPage}, ?\mathit{Info}))]\!]_G$$

$$[\![(?X, \text{email}, ?\mathit{Info})]\!]_G \cup [\![(?X, \text{webPage}, ?\mathit{Info})]\!]_G$$

	$?X$	$?\mathit{Info}$
μ_1	R_1	J@ed.ex
μ_2	R_3	R@ed.ex

Example (UNION)

G : (R_1, name, john) (R_1, email, J@ed.ex) (R_2, name, paul) (R_3, name, ringo) (R_3, email, R@ed.ex) (R_3, webPage, www.ringo.com)

$$[\![((?X, \text{email}, ?Info) \text{ UNION } (?X, \text{webPage}, ?Info))]\!]_G$$

$$[\![(?X, \text{email}, ?Info)]\!]_G \cup [\![(?X, \text{webPage}, ?Info)]\!]_G$$

	$?X$	$?Info$
μ_1	R_1	J@ed.ex
μ_2	R_3	R@ed.ex

	$?X$	$?Info$
μ_3	R_3	www.ringo.com

Example (UNION)

G : (R_1, name, john) (R_1, email, J@ed.ex) (R_2, name, paul) (R_3, name, ringo) (R_3, email, R@ed.ex) (R_3, webPage, www.ringo.com)

$$[\![((?X, \text{email}, ?Info) \text{ UNION } (?X, \text{webPage}, ?Info))]\!]_G$$

$$[\![(?X, \text{email}, ?Info)]\!]_G \cup [\![(?X, \text{webPage}, ?Info)]\!]_G$$

	$?X$	$?Info$
μ_1	R_1	J@ed.ex
μ_2	R_3	R@ed.ex

$\cup$

	$?X$	$?Info$
μ_3	R_3	www.ringo.com

Example (UNION)

G : (R_1, name, john) (R_1, email, J@ed.ex) (R_2, name, paul) (R_3, name, ringo) (R_3, email, R@ed.ex) (R_3, webPage, www.ringo.com)

$$[\![((?X, \text{email}, ?Info) \text{ UNION } (?X, \text{webPage}, ?Info))]\!]_G$$

$$[\![(?X, \text{email}, ?Info)]\!]_G \cup [\![(?X, \text{webPage}, ?Info)]\!]_G$$

	$?X$	$?Info$
μ_1	R_1	J@ed.ex
μ_2	R_3	R@ed.ex

$\cup$

	$?X$	$?Info$
μ_3	R_3	www.ringo.com

	$?X$	$?Info$
μ_1	R_1	J@ed.ex
μ_2	R_3	R@ed.ex
μ_3	R_3	www.ringo.com

Boolean filter expressions (value constraints)

In filter expressions we consider

- the equality $=$ among variables and RDF terms
- a unary predicate bound
- boolean combinations ($\wedge$, $\vee$, $\neg$)

A mapping μ satisfies

- $?X = c$ if $\mu(?X) = c$
- $?X = ?Y$ if $\mu(?X) = \mu(?Y)$
- bound$(?X)$ if μ is defined in $?X$, i.e. $?X \in \text{dom}(\mu)$

Satisfaction of value constraints

- If P is a graph pattern and R is a value constraint then (P FILTER R) is also a graph pattern.

Satisfaction of value constraints

- If P is a graph pattern and R is a value constraint then $(P \text{ FILTER } R)$ is also a graph pattern.

Definition

Given a graph G

- $[\![(P \text{ FILTER } R)]\!]_G = \{\mu \in [\![P]\!]_G \mid \mu \text{ satisfies } R\}$
  i.e. mappings in the evaluation of P that satisfy R.

Example (FILTER)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ FILTER } (?N = \text{ringo} \vee ?N = \text{paul}))]\!]_G$$

Example (FILTER)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ FILTER } (?N = \text{ringo} \vee ?N = \text{paul}))]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

Example (FILTER)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ FILTER } (?N = \text{ringo} \vee ?N = \text{paul}))]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

$?N = \text{ringo} \vee ?N = \text{paul}$

Example (FILTER)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![((?X, \text{name}, ?N) \text{ FILTER } (?N = \text{ringo} \vee ?N = \text{paul}))]\!]_G$$

	$?X$	$?N$
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

$$?N = \text{ringo} \vee ?N = \text{paul}$$

	$?X$	$?N$
μ_2	R_2	paul
μ_3	R_3	ringo

Example (FILTER)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![(((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E)) \text{ FILTER } \neg \text{bound}(?E))]\!]_G$$

Example (FILTER)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![(((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E)) \text{ FILTER } \neg \text{bound}(?E))]\!]_G$$

	$?X$	$?N$	$?E$
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

Example (FILTER)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![(((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E)) \text{ FILTER } \neg \text{bound}(?E))]\!]_G$$

	$?X$	$?N$	$?E$
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

$\neg \text{bound}(?E)$

Example (FILTER)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![(((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E)) \text{ FILTER } \neg \text{bound}(?E))]\!]_G$$

	$?X$	$?N$	$?E$
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

$\neg \text{bound}(?E)$

	$?X$	$?N$
μ_2	R_2	paul

Example (FILTER)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![(((?X, \text{name}, ?N) \text{ OPT } (?X, \text{email}, ?E)) \text{ FILTER } \neg \text{bound}(?E))]\!]_G$$

	$?X$	$?N$	$?E$
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

$\neg \text{bound}(?E)$

	$?X$	$?N$
μ_2	R_2	paul

(a *non-monotonic* query)

Why do we need/want to formalize SPARQL

A formalization is beneficial

- clarifying corner cases
- helping in the implementation process
- providing solid foundations (we can actually prove properties!)

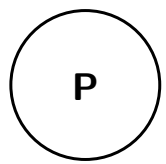
The evaluation decision problem

Evaluation problem for SPARQL patterns

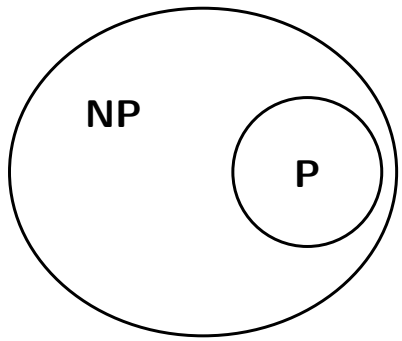
Input: A mapping μ, an RDF graph G
a graph pattern P

Output: Is the mapping μ in the evaluation of pattern P
over the RDF graph G

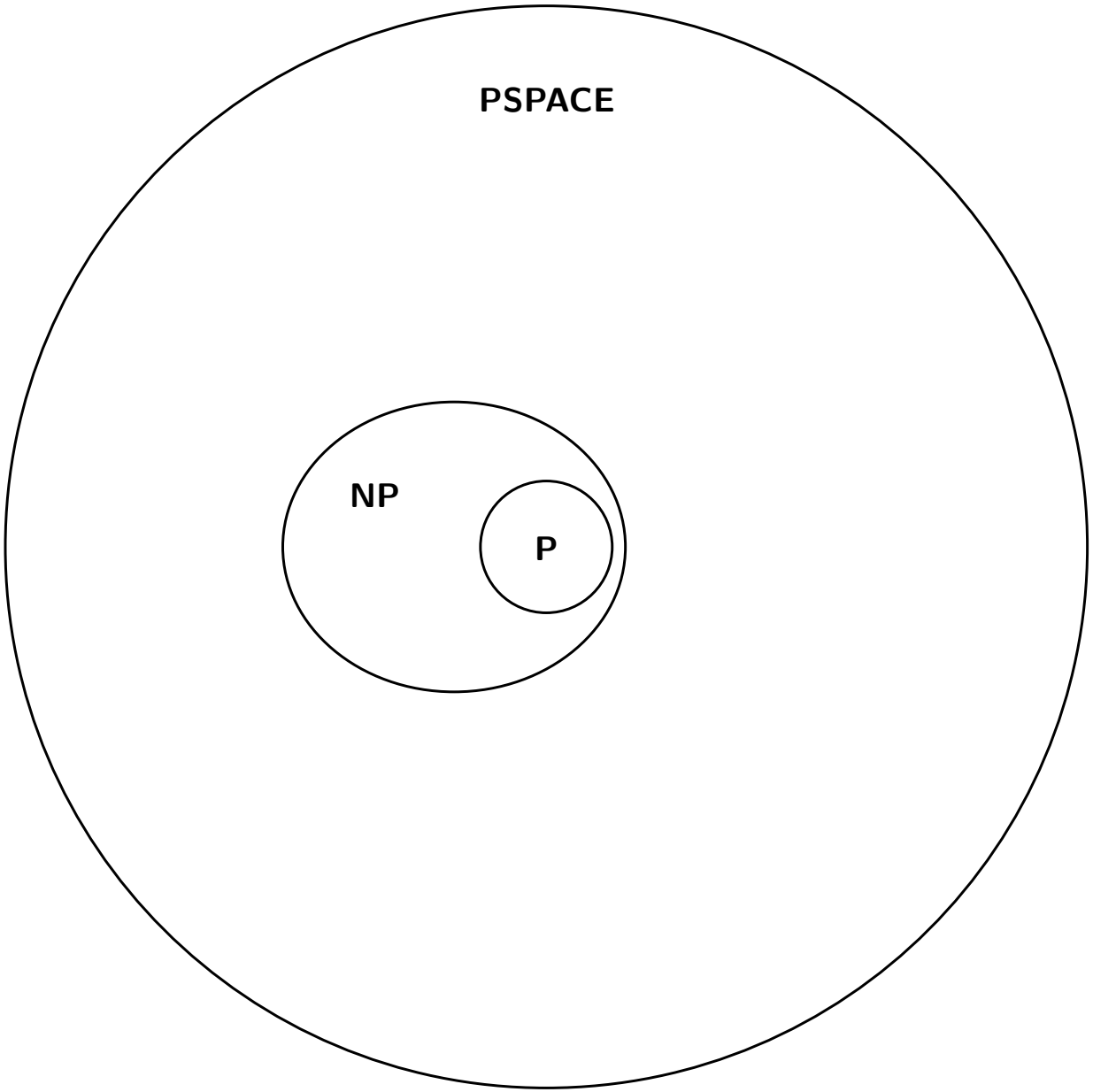
Brief complexity-theory reminder



Brief complexity-theory reminder



Brief complexity-theory reminder



Complexity of the evaluation problem

Theorem (PAG09)

For patterns using only the AND *operator,*

the evaluation problem is in **P**

Complexity of the evaluation problem

Theorem (PAG09)

For patterns using only the AND *operator,*

the evaluation problem is in **P**

Theorem (SML10)

For patterns using AND *and* UNION *operators,*

the evaluation problem is **NP***-complete.*

Complexity of the evaluation problem

Theorem (PAG09)

For patterns using only the AND *operator,*

the evaluation problem is in **P**

Theorem (SML10)

For patterns using AND *and* UNION *operators,*

the evaluation problem is **NP***-complete.*

Theorem (PAG09,SML10)

For general patterns that include OPT *operator,*

the evaluation problem is **PSPACE***-complete.*

Complexity of the evaluation problem

Theorem (PAG09)

For patterns using only the AND *operator,*

the evaluation problem is in **P**

Theorem (SML10)

For patterns using AND *and* UNION *operators,*

the evaluation problem is **NP***-complete.*

Theorem (PAG09,SML10)

For general patterns that include OPT *operator,*

the evaluation problem is **PSPACE***-complete.*

Good news: evaluation in **P** if the query is fixed (*data complexity*)

Well–designed patterns

Can we find a natural fragment with better complexity?

Definition

An AND-OPT pattern is *well–designed* iff for every OPT in the pattern

$$(\ \cdots\cdots\cdots\ \ (\ \ A\ \ \text{OPT}\ \ B\ \ )\ \ \cdots\cdots\cdots\ )$$

if a variable occurs

Well–designed patterns

Can we find a natural fragment with better complexity?

Definition

An AND-OPT pattern is *well–designed* iff for every OPT in the pattern

$$(\cdots\cdots\cdots\cdots \quad (\quad A \quad \text{OPT} \quad \underset{\uparrow}{B} \quad) \quad \cdots\cdots\cdots\cdots)$$

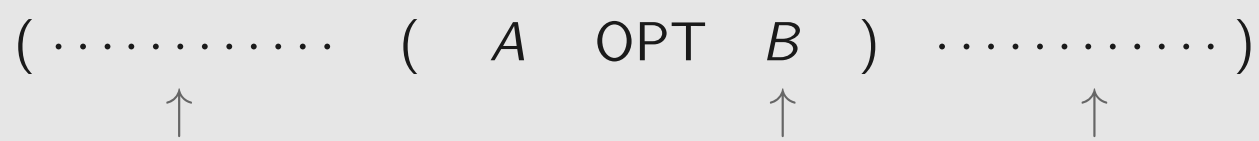
if a variable occurs *inside* B

Well–designed patterns

Can we find a natural fragment with better complexity?

Definition

An AND-OPT pattern is *well–designed* iff for every OPT in the pattern



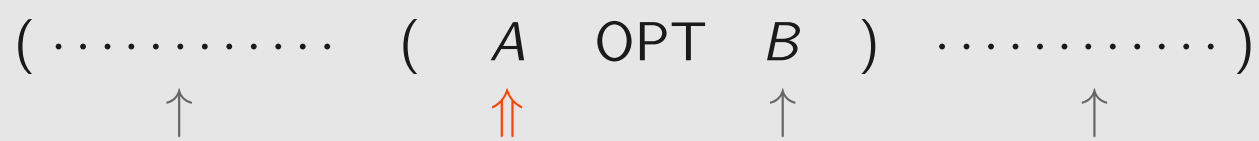
if a variable occurs *inside* B and *anywhere outside the* OPT,

Well–designed patterns

Can we find a natural fragment with better complexity?

Definition

An AND-OPT pattern is *well–designed* iff for every OPT in the pattern



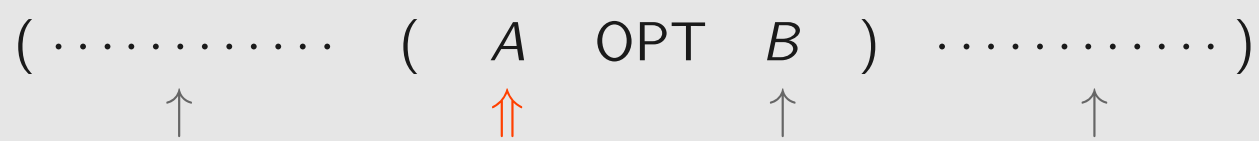
if a variable occurs *inside B* and *anywhere outside the* OPT, then the variable *must also occur inside A*.

Well–designed patterns

Can we find a natural fragment with better complexity?

Definition

An AND-OPT pattern is *well–designed* iff for every OPT in the pattern



if a variable occurs *inside B* and *anywhere outside the* OPT, then the variable *must also occur inside A*.

Example

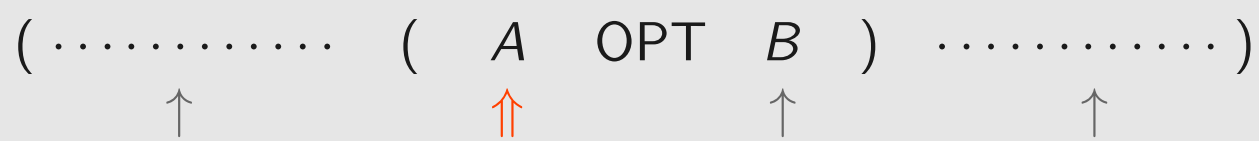
$$\big[\,\big[\,(?Y, \text{name}, \text{paul})\ \text{OPT}\ (?X, \text{email}, ?Z)\,\big]\ \ \text{AND}\ \ (?X, \text{name}, \text{john})\,\big]$$

Well–designed patterns

Can we find a natural fragment with better complexity?

Definition

An AND-OPT pattern is *well–designed* iff for every OPT in the pattern



if a variable occurs *inside B* and *anywhere outside the* OPT, then the variable *must also occur inside A*.

Example

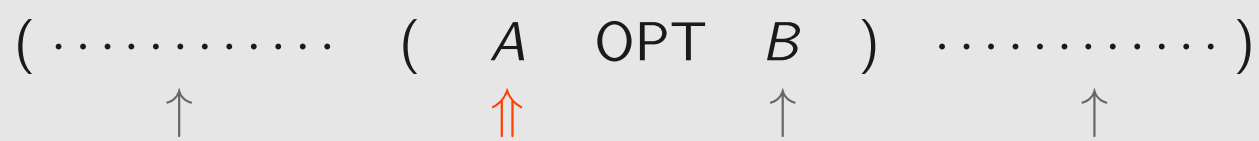
$$\big[\,\big[\,(?Y,\ \text{name},\ \text{paul})\ \text{OPT}\ (?X,\ \text{email},\ ?Z)\,\big]\ \ \text{AND}\ \ (?X,\ \text{name},\ \text{john})\,\big]$$

Well–designed patterns

Can we find a natural fragment with better complexity?

Definition

An AND-OPT pattern is *well–designed* iff for every OPT in the pattern



if a variable occurs *inside B* and *anywhere outside the* OPT, then the variable *must also occur inside A*.

Example

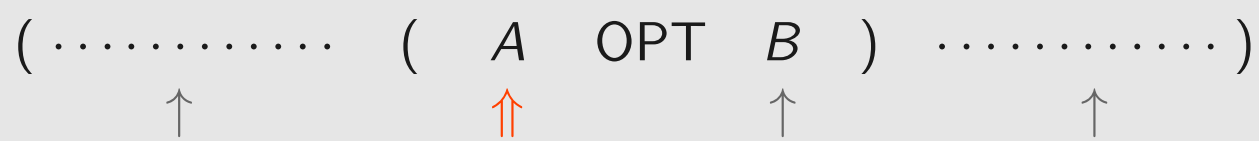
$$[\,[\,(?Y, \text{name}, \text{paul}) \text{ OPT } (?X, \text{email}, ?Z)\,]\ \text{AND}\ (?X, \text{name}, \text{john})\,]$$

Well–designed patterns

Can we find a natural fragment with better complexity?

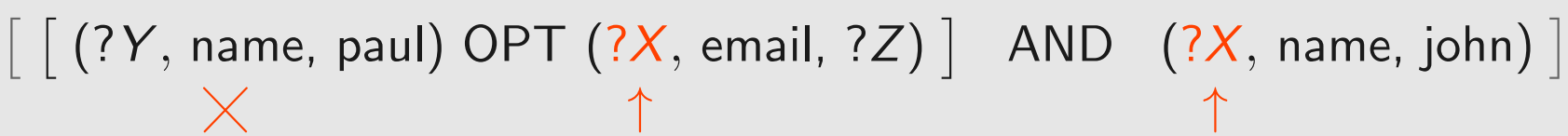
Definition

An AND-OPT pattern is *well–designed* iff for every OPT in the pattern

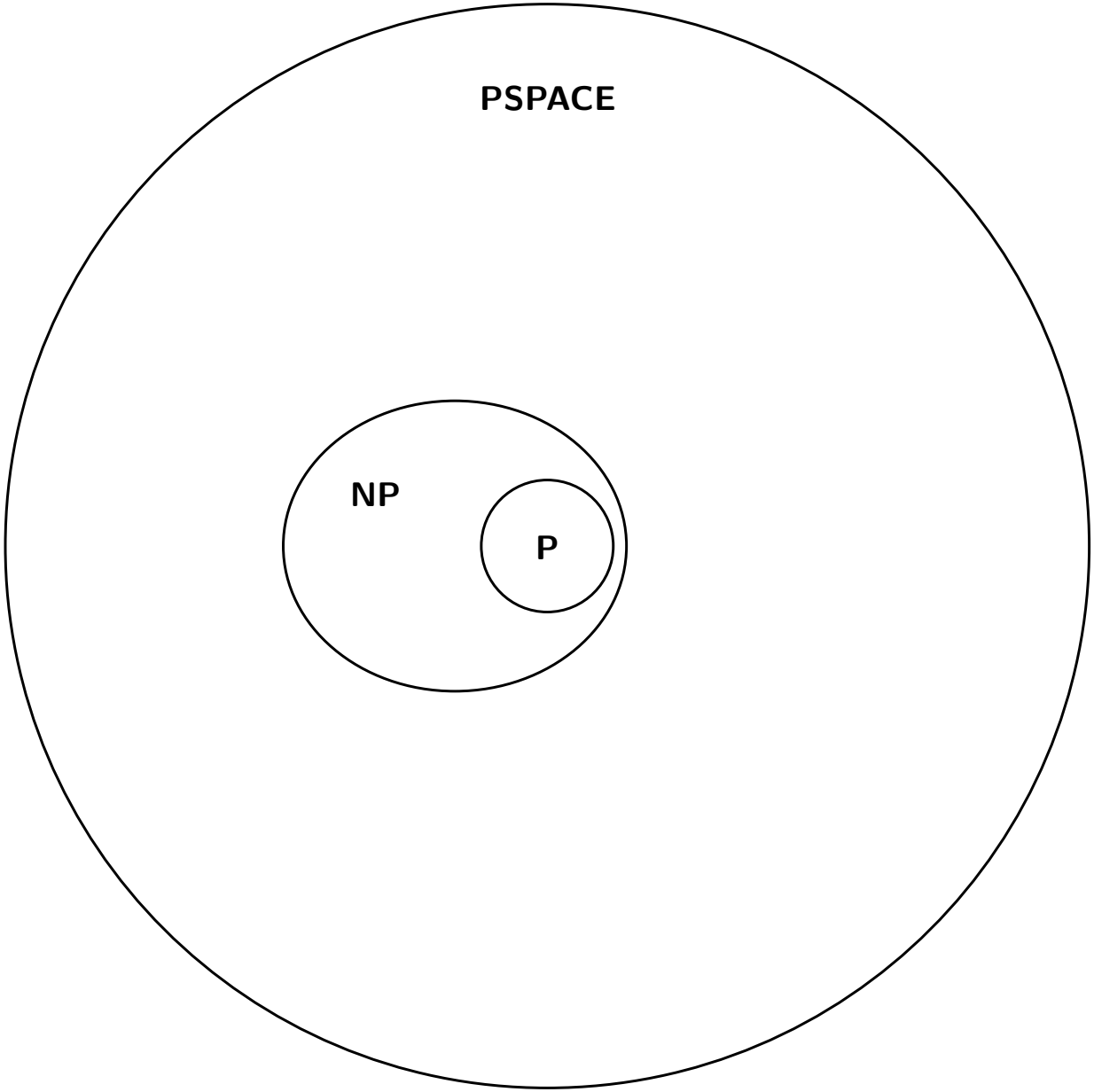


if a variable occurs *inside B* and *anywhere outside the* OPT, then the variable *must also occur inside A*.

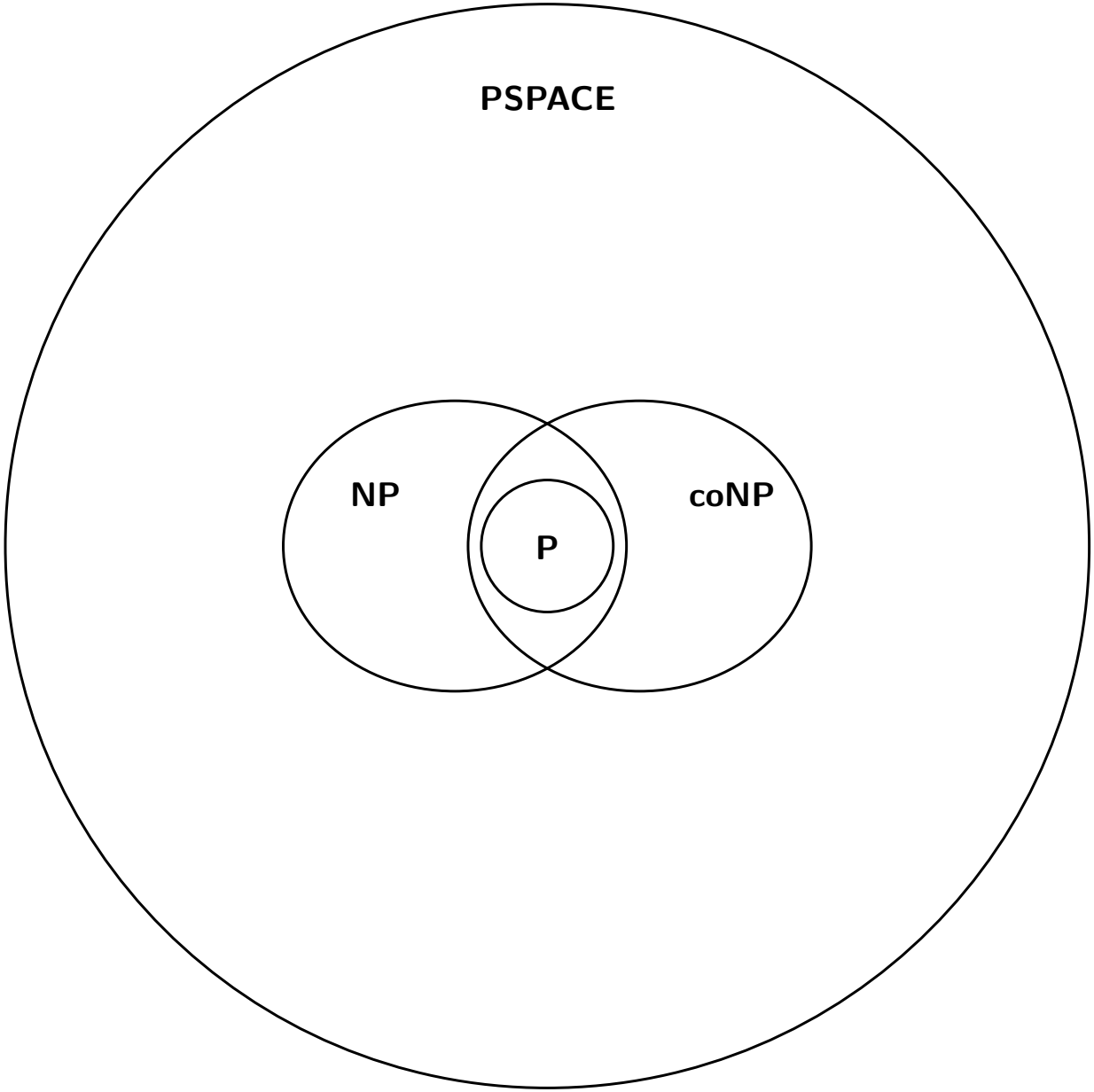
Example



A bit more on complexity...



A bit more on complexity...



Evaluation of well-designed patterns is in coNP-complete

Theorem (PAG09)

For AND-OPT *well–designed graph patterns*

the evaluation problem is **coNP***-complete*

Evaluation of well-designed patterns is in coNP-complete

Theorem (PAG09)

For AND-OPT *well–designed graph patterns*

the evaluation problem is **coNP***-complete*

Well-designed patterns also allow to study static analysis:

Theorem (LPPS12)

Equivalence of well-designed SPARQL patterns is in **NP**

SELECT (a.k.a. projection)

Besides graph patterns, SPARQL 1.0 allow *result forms*
the most simple is SELECT

Definition

A SELECT query is an expression

$$(\text{SELECT } W \; P)$$

where P is a graph pattern and W is a set of variables, or $*$

SELECT (a.k.a. projection)

Besides graph patterns, SPARQL 1.0 allow *result forms*
the most simple is SELECT

Definition

A SELECT query is an expression

$$(\text{SELECT } W\ P)$$

where P is a graph pattern and W is a set of variables, or $*$

The evaluation of a SELECT query against G is

- $[\![(\text{SELECT } W\ P)]\!]_G = \{\mu_{|W} \mid \mu \in [\![P]\!]_G\}$
  where $\mu_{|W}$ is the restriction of μ to domain W.
- $[\![(\text{SELECT } *\ P)]\!]_G = [\![P]\!]_G$

Example (SELECT)

G :

$(R_1, \text{name}, \text{john})$	$(R_2, \text{name}, \text{paul})$	$(R_3, \text{name}, \text{ringo})$
$(R_1, \text{email}, \text{J@ed.ex})$		$(R_3, \text{email}, \text{R@ed.ex})$
		$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![(\text{SELECT } \{?N, ?E\} ((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E)))]\!]_G$$

Example (SELECT)

G :

$(R_1, \text{name}, \text{john})$	$(R_2, \text{name}, \text{paul})$	$(R_3, \text{name}, \text{ringo})$
$(R_1, \text{email}, \text{J@ed.ex})$		$(R_3, \text{email}, \text{R@ed.ex})$
		$(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![(\text{SELECT } \{?N, ?E\} ((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E)))]\!]_G$$

$\text{SELECT}\{?N, ?E\}$

	$?X$	$?N$	$?E$
μ_1	R_1	john	J@ed.ex
μ_2	R_3	ringo	R@ed.ex

Example (SELECT)

G :
(R_1, name, john)
(R_1, email, J@ed.ex)
(R_2, name, paul)
(R_3, name, ringo)
(R_3, email, R@ed.ex)
(R_3, webPage, www.ringo.com)

$$[\![(\text{SELECT } \{?N, ?E\} \ ((?X, \text{name}, ?N) \text{ AND } (?X, \text{email}, ?E)))]\!]_G$$

$\text{SELECT}_{\{?N,?E\}}$

	$?X$	$?N$	$?E$
μ_1	R_1	john	J@ed.ex
μ_2	R_3	ringo	R@ed.ex

	$?N$	$?E$
$\mu_1\|_{\{?N,?E\}}$	john	J@ed.ex
$\mu_2\|_{\{?N,?E\}}$	ringo	R@ed.ex

SPARQL 1.1 introduces several new features

In SPARQL 1.1:

- (SELECT W P) can be used as any other graph pattern
  (ASK P) can be used as a constraint in FILTER
  $\Rightarrow$ sub-queries
- Aggregations via ORDER-BY plus COUNT, SUM, etc.
- More important for us: Federation and Navigation

Outline

SPARQL endpoints and the SERVICE operator

- *SPARQL endpoints* are services that accepts HTTP requests asking for SPARQL queries
  - `http://www.w3.org/wiki/SparqlEndpoints` lists some
- SPARQL 1.1 allows to *mix* local and remote queries to endpoints via the SERVICE operator

SPARQL endpoints and the SERVICE operator

- *SPARQL endpoints* are services that accepts HTTP requests asking for SPARQL queries
  - `http://www.w3.org/wiki/SparqlEndpoints` lists some
- SPARQL 1.1 allows to *mix* local and remote queries to endpoints via the SERVICE operator

```
?SELECT ?Author
WHERE
{
   ?Paper dc:creator ?Author .
   ?Paper dct:partOf ?Conf .
   ?Conf  swrc:series conf:iswc .



}
```

SPARQL endpoints and the SERVICE operator

- *SPARQL endpoints* are services that accepts HTTP requests asking for SPARQL queries
  - `http://www.w3.org/wiki/SparqlEndpoints` lists some
- SPARQL 1.1 allows to *mix* local and remote queries to endpoints via the SERVICE operator

```
?SELECT ?Author ?Place
WHERE
{
   ?Paper dc:creator ?Author .
   ?Paper dct:partOf ?Conf .
   ?Conf  swrc:series conf:iswc .

   SERVICE dbpedia:service
         { ?Author dbpedia:birthPlace ?Place . }
}
```

Syntax of SERVICE

Syntax

If $c \in U$, $?X$ is a variable, and P is a graph pattern then

Syntax of SERVICE

Syntax

If $c \in U$, $?X$ is a variable, and P is a graph pattern then

- (SERVICE c P)
- (SERVICE $?X$ P)

are SERVICE graph patterns.

Syntax of SERVICE

Syntax

If $c \in U$, $?X$ is a variable, and P is a graph pattern then

- (SERVICE c P)
- (SERVICE $?X$ P)

are SERVICE graph patterns.

- SERVICE graph pattern are included recursively in the algebra of graph patterns
- Variables are included in the SERVICE syntax to allow *dynamic* choosing of endpoints

Semantics of SERVICE

We assume the existence of a partial function

$$\text{ep} : U \rightarrow \text{RDF graphs}$$

intuitively, ep(c) is the (default) graph associated to the SPARQL endpoint defined by c

Semantics of SERVICE

We assume the existence of a partial function

$$\text{ep} : U \rightarrow \text{RDF graphs}$$

intuitively, ep(c) is the (default) graph associated to the SPARQL endpoint defined by c

Definition

Given an RDF graph G, an element $c \in U$, and a graph pattern P

$$[\![(\text{SERVICE } c \ P)]\!]_G = \left\{ \right.$$

Semantics of SERVICE

We assume the existence of a partial function

$$\text{ep} : U \rightarrow \text{RDF graphs}$$

intuitively, ep(c) is the (default) graph associated to the SPARQL endpoint defined by c

Definition

Given an RDF graph G, an element $c \in U$, and a graph pattern P

$$[\![(\text{SERVICE } c \ P)]\!]_G = \begin{cases} [\![P]\!]_{\text{ep}(c)} & \text{if } c \in \text{dom(ep)} \\ \end{cases}$$

Semantics of SERVICE

We assume the existence of a partial function

$$\text{ep} : U \rightarrow \text{RDF graphs}$$

intuitively, $\text{ep}(c)$ is the (default) graph associated to the SPARQL endpoint defined by c

Definition

Given an RDF graph G, an element $c \in U$, and a graph pattern P

$$[\![(\text{SERVICE } c \ P)]\!]_G = \begin{cases} [\![P]\!]_{\text{ep}(c)} & \text{if } c \in \text{dom}(\text{ep}) \\ \{\mu_\emptyset\} & \text{otherwise} \end{cases}$$

Semantics of SERVICE

We assume the existence of a partial function

$$\text{ep} : U \rightarrow \text{RDF graphs}$$

intuitively, ep(c) is the (default) graph associated to the SPARQL endpoint defined by c

Definition

Given an RDF graph G, an element $c \in U$, and a graph pattern P

$$[\![(\text{SERVICE } c \ P)]\!]_G = \begin{cases} [\![P]\!]_{\text{ep}(c)} & \text{if } c \in \text{dom(ep)} \\ \{\mu_\emptyset\} & \text{otherwise} \end{cases}$$

but the interesting case is when the endpoint is a variable...

Semantics of SERVICE

Definition

Given an RDF graph G, a variable $?X$, and a graph pattern P

$$[\![(\text{SERVICE } ?X \ P)]\!]_G =$$

Semantics of SERVICE

Definition

Given an RDF graph G, a variable $?X$, and a graph pattern P

$$[\![(\text{SERVICE } ?X \ P)]\!]_G = \qquad [\![P]\!]_{\text{ep}(c)}$$

Semantics of SERVICE

Definition

Given an RDF graph G, a variable $?X$, and a graph pattern P

$$[\![(\text{SERVICE } ?X \ P)]\!]_G = [\![P]\!]_{\text{ep}(c)} \bowtie \{\{?X \to c\}\}$$

Semantics of SERVICE

Definition

Given an RDF graph G, a variable $?X$, and a graph pattern P

$$[\![(\text{SERVICE } ?X \ P)]\!]_G = \bigcup_{c \in \text{dom}(\text{ep})} \left([\![P]\!]_{\text{ep}(c)} \bowtie \{\{?X \to c\}\} \right)$$

Semantics of SERVICE

Definition

Given an RDF graph G, a variable $?X$, and a graph pattern P

$$[\![(\text{SERVICE } ?X \ P)]\!]_G = \bigcup_{c \in \text{dom(ep)}} \left([\![P]\!]_{\text{ep}(c)} \bowtie \{\{?X \to c\}\} \right)$$

Semantics of SERVICE

Definition

Given an RDF graph G, a variable $?X$, and a graph pattern P

$$[\![(\text{SERVICE } ?X \ P)]\!]_G = \bigcup_{c \in \text{dom(ep)}} \left([\![P]\!]_{\text{ep}(c)} \bowtie \{\{?X \to c\}\} \right)$$

can we effectively evaluate a SERVICE query?

SERVICE example

Some queries/patterns can be safely evaluated:

SERVICE example

Some queries/patterns can be safely evaluated:

- $((?X, \text{service\_address}, ?Y)$ AND $(\text{SERVICE } ?Y\ (?N, \text{email}, ?E)))$

SERVICE example

Some queries/patterns can be safely evaluated:

- $((?X, \text{service\_address}, ?Y)$ AND $(\text{SERVICE } ?Y\ (?N, \text{email}, ?E)))$
- $((\text{SERVICE } ?Y\ (?N, \text{email}, ?E))$ AND $(?X, \text{service\_address}, ?Y))$

SERVICE example

Some queries/patterns can be safely evaluated:

- $((?X, \text{service\_address}, ?Y)$ AND $(\text{SERVICE } ?Y\ (?N, \text{email}, ?E)))$
- $((\text{SERVICE } ?Y\ (?N, \text{email}, ?E))$ AND $(?X, \text{service\_address}, ?Y))$

In both cases, the SERVICE variable is *controlled* by the data in the initial graph

SERVICE example

Some queries/patterns can be safely evaluated:

- $((?X, \text{service\_address}, ?Y)$ AND $(\text{SERVICE } ?Y\ (?N, \text{email}, ?E)))$
- $((\text{SERVICE } ?Y\ (?N, \text{email}, ?E))$ AND $(?X, \text{service\_address}, ?Y))$

In both cases, the SERVICE variable is *controlled* by the data in the initial graph

There is natural way of evaluating the query:

SERVICE example

Some queries/patterns can be safely evaluated:

- $((?X, \text{service\_address}, ?Y)$ AND $(\text{SERVICE } ?Y\ (?N, \text{email}, ?E)))$
- $((\text{SERVICE } ?Y\ (?N, \text{email}, ?E))$ AND $(?X, \text{service\_address}, ?Y))$

In both cases, the SERVICE variable is *controlled* by the data in the initial graph

There is natural way of evaluating the query:

- Evaluate first $(?X, \text{service\_address}, ?Y)$

SERVICE example

Some queries/patterns can be safely evaluated:

- $((?X, \text{service\_address}, ?Y)$ AND $(\text{SERVICE } ?Y\ (?N, \text{email}, ?E)))$
- $((\text{SERVICE } ?Y\ (?N, \text{email}, ?E))$ AND $(?X, \text{service\_address}, ?Y))$

In both cases, the SERVICE variable is *controlled* by the data in the initial graph

There is natural way of evaluating the query:

- Evaluate first $(?X, \text{service\_address}, ?Y)$
- only for the obtained mappings evaluate the SERVICE on endpoint $\mu(?Y)$

Unbounded SERVICE queries

What about this pattern?

$$\Big[((?X, \text{service\_description}, ?Z) \text{ UNION } (?X, \text{service\_address}, ?Y)) \text{ AND } (\text{SERVICE } ?Y \ (?N, \text{email}, ?E))\Big]$$

Unbounded SERVICE queries

What about this pattern?

$$\Big[((?X, \text{service\_description}, ?Z) \text{ UNION } (?X, \text{service\_address}, ?Y)) \text{ AND } (\text{SERVICE } ?Y \ (?N, \text{email}, ?E))\Big]$$

:-(

Unbounded SERVICE queries

What about this pattern?

$$\Big[((?X, \text{service\_description}, ?Z) \text{ UNION } (?X, \text{service\_address}, ?Y)) \text{ AND } (\text{SERVICE } ?Y\ (?N, \text{email}, ?E))\Big]$$

:-(

Idea: force the SERVICE variable to be *bound* in every solution.

Boundedness

Definition

A variable $?X$ is *bound* in graph pattern P if
for every graph G and every $\mu \in [\![P]\!]_G$ it holds that:

- $?X \in \mathrm{dom}(\mu)$, and
- $\mu(?X)$ is a value in G.

Boundedness

Definition

A variable $?X$ is *bound* in graph pattern P if
for every graph G and every $\mu \in [\![P]\!]_G$ it holds that:

- $?X \in \mathrm{dom}(\mu)$, and
- $\mu(?X)$ is a value in G.

We only need a procedure to ensure that every variable mentioned in SERVICE is bounded!

Boundedness

Definition

A variable $?X$ is *bound* in graph pattern P if
for every graph G and every $\mu \in [\![P]\!]_G$ it holds that:

- $?X \in \mathrm{dom}(\mu)$, and
- $\mu(?X)$ is a value in G.

We only need a procedure to ensure that every variable mentioned in SERVICE is bounded! Oh wait...

Boundedness

Definition

A variable $?X$ is *bound* in graph pattern P if
for every graph G and every $\mu \in [\![P]\!]_G$ it holds that:

- $?X \in \mathrm{dom}(\mu)$, and
- $\mu(?X)$ is a value in G.

We only need a procedure to ensure that every variable mentioned in SERVICE is bounded! Oh wait...

Theorem (BAC11)

The problem of checking if a variable is bound in a graph pattern is undecidable.

Boundedness

Definition

A variable $?X$ is *bound* in graph pattern P if
for every graph G and every $\mu \in [\![P]\!]_G$ it holds that:

- $?X \in \mathrm{dom}(\mu)$, and
- $\mu(?X)$ is a value in G.

We only need a procedure to ensure that every variable mentioned in SERVICE is bounded! Oh wait...

Theorem (BAC11)

The problem of checking if a variable is bound in a graph pattern is undecidable.

:-(

Undecidability of boundedness

Proof idea

- From [AG08]: it is undecidable to check if a SPARQL pattern P is satisfiable (if $[\![P]\!]_G \neq \emptyset$ for some G).

Undecidability of boundedness

Proof idea

- From [AG08]: it is undecidable to check if a SPARQL pattern P is satisfiable (if $[\![P]\!]_G \neq \emptyset$ for some G).
- Assume P does not mention $?X$, and let $Q = ((?X, ?Y, ?Z)$ UNION $P)$:

Undecidability of boundedness

Proof idea

- From [AG08]: it is undecidable to check if a SPARQL pattern P is satisfiable (if $[\![P]\!]_G \neq \emptyset$ for some G).
- Assume P does not mention $?X$, and let $Q = ((?X, ?Y, ?Z)$ UNION $P)$:

  $?X$ is bound in Q iff P is not satisfiable.

Undecidability of boundedness

Proof idea

- From [AG08]: it is undecidable to check if a SPARQL pattern P is satisfiable (if $[\![P]\!]_G \neq \emptyset$ for some G).
- Assume P does not mention $?X$, and let $Q = ((?X, ?Y, ?Z)$ UNION $P)$:

  $?X$ is bound in Q iff P is not satisfiable.

Undecidable: the SPARQL engine cannot check for boundedness...

A sufficient condition for boundedness

Definition [BAC11]

The set of *strongly bounded* variables in a pattern P, denoted by $\mathrm{SB}(P)$ is defined recursively as follows.

- if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$
- if $P = (P_1 \text{ AND } P_2)$, then

A sufficient condition for boundedness

Definition [BAC11]

The set of *strongly bounded* variables in a pattern P, denoted by $\mathrm{SB}(P)$ is defined recursively as follows.

- if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$
- if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$

A sufficient condition for boundedness

Definition [BAC11]

The set of *strongly bounded* variables in a pattern P, denoted by $\mathrm{SB}(P)$ is defined recursively as follows.

- if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$
- if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$
- if $P = (P_1 \text{ OPT } P_2)$, then

A sufficient condition for boundedness

Definition [BAC11]

The set of *strongly bounded* variables in a pattern P, denoted by $\mathrm{SB}(P)$ is defined recursively as follows.

- if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$
- if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$
- if $P = (P_1 \text{ OPT } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1)$

A sufficient condition for boundedness

Definition [BAC11]

The set of *strongly bounded* variables in a pattern P, denoted by $\mathrm{SB}(P)$ is defined recursively as follows.

- if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$
- if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$
- if $P = (P_1 \text{ OPT } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1)$
- if $P = (P_1 \text{ UNION } P_2)$, then

A sufficient condition for boundedness

Definition [BAC11]

The set of *strongly bounded* variables in a pattern P, denoted by $\mathrm{SB}(P)$ is defined recursively as follows.

- if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$
- if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$
- if $P = (P_1 \text{ OPT } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1)$
- if $P = (P_1 \text{ UNION } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cap \mathrm{SB}(P_2)$

A sufficient condition for boundedness

Definition [BAC11]

The set of *strongly bounded* variables in a pattern P, denoted by $\mathrm{SB}(P)$ is defined recursively as follows.

- if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$
- if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$
- if $P = (P_1 \text{ OPT } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1)$
- if $P = (P_1 \text{ UNION } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cap \mathrm{SB}(P_2)$
- if $P = (\text{SELECT } W\, P_1)$, then

A sufficient condition for boundedness

Definition [BAC11]

The set of *strongly bounded* variables in a pattern P, denoted by $\mathrm{SB}(P)$ is defined recursively as follows.

- if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$
- if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$
- if $P = (P_1 \text{ OPT } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1)$
- if $P = (P_1 \text{ UNION } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cap \mathrm{SB}(P_2)$
- if $P = (\text{SELECT } W\, P_1)$, then $\mathrm{SB}(P) = W \cap \mathrm{SB}(P_1)$

A sufficient condition for boundedness

Definition [BAC11]

The set of *strongly bounded* variables in a pattern P, denoted by $\mathrm{SB}(P)$ is defined recursively as follows.

- if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$
- if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$
- if $P = (P_1 \text{ OPT } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1)$
- if $P = (P_1 \text{ UNION } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cap \mathrm{SB}(P_2)$
- if $P = (\text{SELECT } W\, P_1)$, then $\mathrm{SB}(P) = W \cap \mathrm{SB}(P_1)$
- if P is a SERVICE pattern, then

A sufficient condition for boundedness

Definition [BAC11]

The set of *strongly bounded* variables in a pattern P, denoted by $\mathrm{SB}(P)$ is defined recursively as follows.

- if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$
- if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$
- if $P = (P_1 \text{ OPT } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1)$
- if $P = (P_1 \text{ UNION } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cap \mathrm{SB}(P_2)$
- if $P = (\text{SELECT } W\, P_1)$, then $\mathrm{SB}(P) = W \cap \mathrm{SB}(P_1)$
- if P is a SERVICE pattern, then $\mathrm{SB}(P) = \emptyset$

A sufficient condition for boundedness

Definition [BAC11]

The set of *strongly bounded* variables in a pattern P, denoted by $\mathrm{SB}(P)$ is defined recursively as follows.

- if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$
- if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$
- if $P = (P_1 \text{ OPT } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1)$
- if $P = (P_1 \text{ UNION } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cap \mathrm{SB}(P_2)$
- if $P = (\text{SELECT } W\, P_1)$, then $\mathrm{SB}(P) = W \cap \mathrm{SB}(P_1)$
- if P is a SERVICE pattern, then $\mathrm{SB}(P) = \emptyset$

Proposition (BAC11)

If $?X \in \mathrm{SB}(P)$ *then* $?X$ *is bound in* P.

A sufficient condition for boundedness

Definition [BAC11]

The set of *strongly bounded* variables in a pattern P, denoted by $\mathrm{SB}(P)$ is defined recursively as follows.

- if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$
- if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$
- if $P = (P_1 \text{ OPT } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1)$
- if $P = (P_1 \text{ UNION } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cap \mathrm{SB}(P_2)$
- if $P = (\text{SELECT } W\, P_1)$, then $\mathrm{SB}(P) = W \cap \mathrm{SB}(P_1)$
- if P is a SERVICE pattern, then $\mathrm{SB}(P) = \emptyset$

Proposition (BAC11)

If $?X \in \mathrm{SB}(P)$ *then* $?X$ *is bound in* P.

($\mathrm{SB}(P)$ can be efficiently computed)

(Strongly) boundedness is not enough

Are we happy now?

$$P_1 = \Big[(?X, \text{service\_description}, ?Z) \text{ UNION } \Big((?X, \text{service\_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y \ (?N, \text{email}, ?E)) \Big) \Big].$$

(Strongly) boundedness is not enough

Are we happy now?

$$P_1 = \Big[(?X, \text{service\_description}, ?Z) \text{ UNION } \Big((?X, \text{service\_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y \ (?N, \text{email}, ?E)) \Big) \Big].$$

- $?Y$ is not bound in P_1 (nor strongly bound)

(Strongly) boundedness is not enough

Are we happy now?

$$P_1 = \Big[(?X, \text{service\_description}, ?Z) \text{ UNION } \Big((?X, \text{service\_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y \ (?N, \text{email}, ?E)) \Big) \Big].$$

- $?Y$ is not bound in P_1 (nor strongly bound)
- nevertheless there is a natural evaluation of this pattern

(Strongly) boundedness is not enough

Are we happy now?

$$P_1 = \Big[(?X, \text{service\_description}, ?Z) \text{ UNION } \Big((?X, \text{service\_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y \ (?N, \text{email}, ?E)) \Big) \Big].$$

- $?Y$ is not bound in P_1 (nor strongly bound)
- nevertheless there is a natural evaluation of this pattern
  ($?Y$ is bounded in the *important* part!)

(Strongly) boundedness is not enough

Are we happy now?

(Strongly) boundedness is not enough

Are we happy now?

$$P_2 = \Big[(?U_1, \text{related\_with}, ?U_2) \ \text{AND} \ \Big(\text{SERVICE } ?U_1 \ \Big((?N, \text{email}, ?E) \ \text{OPT} \ (\text{SERVICE } ?U_2 \ (?N, \text{phone}, ?F)) \Big) \Big) \Big].$$

(Strongly) boundedness is not enough

Are we happy now?

$$P_2 = \Big[(?U_1, \text{related\_with}, ?U_2) \text{ AND } \Big(\text{SERVICE } ?U_1 \Big((?N, \text{email}, ?E) \text{ OPT } (\text{SERVICE } ?U_2 \ (?N, \text{phone}, ?F)) \Big) \Big) \Big].$$

- $?U_1$ and $?U_2$ are strongly bounded, but

(Strongly) boundedness is not enough

Are we happy now?

$$P_2 = \Big[(?U_1, \text{related\_with}, ?U_2) \text{ AND } \Big(\text{SERVICE } ?U_1 \Big((?N, \text{email}, ?E) \text{ OPT } (\text{SERVICE } ?U_2 \ (?N, \text{phone}, ?F)) \Big) \Big) \Big].$$

- $?U_1$ and $?U_2$ are strongly bounded, but
- can we effectively evaluate this query?
  ($?U_2$ is unbounded in the *important* part!)

(Strongly) boundedness is not enough

Are we happy now?

$$P_2 = \Big[(?U_1, \text{related\_with}, ?U_2) \text{ AND } \Big(\text{SERVICE } ?U_1 \Big((?N, \text{email}, ?E) \text{ OPT } (\text{SERVICE } ?U_2 \ (?N, \text{phone}, ?F)) \Big) \Big) \Big].$$

- $?U_1$ and $?U_2$ are strongly bounded, but
- can we effectively evaluate this query?
  ($?U_2$ is unbounded in the *important* part!)

we need to define what the *important* part is...

Parse tree of a pattern

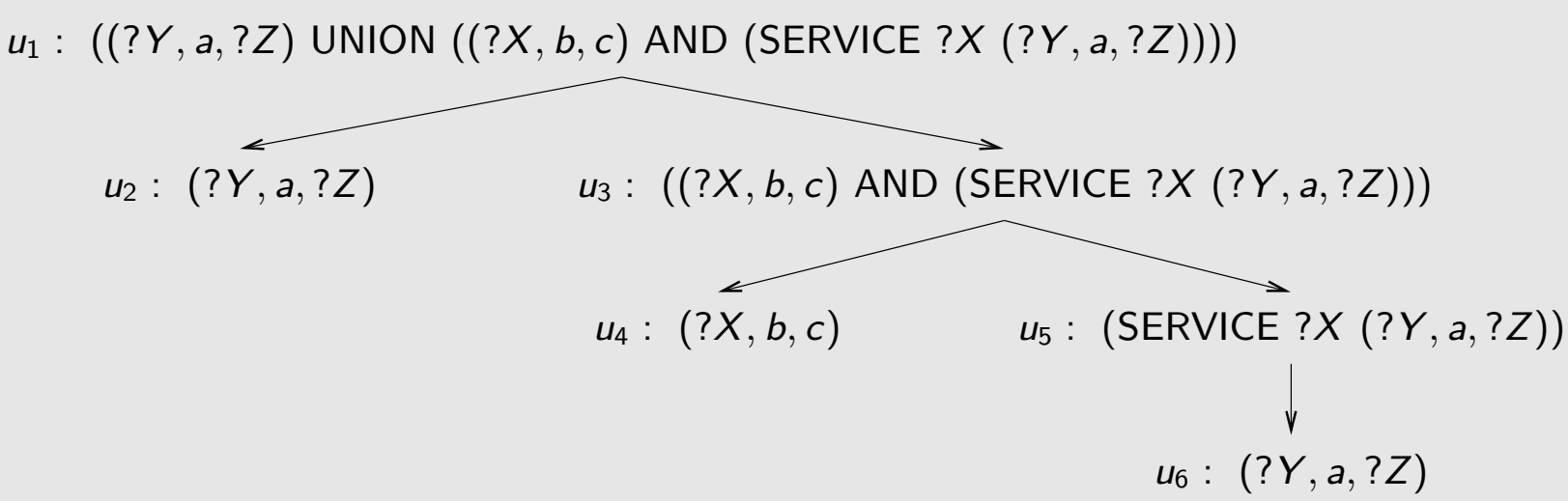
We need first to formalize the tree of subexpressions of a pattern

$$\Big((?Y, a, ?Z) \text{ UNION } ((?X, b, c) \text{ AND } (\text{SERVICE } ?X \ (?Y, a, ?Z)))\Big)$$

Parse tree of a pattern

We need first to formalize the tree of subexpressions of a pattern

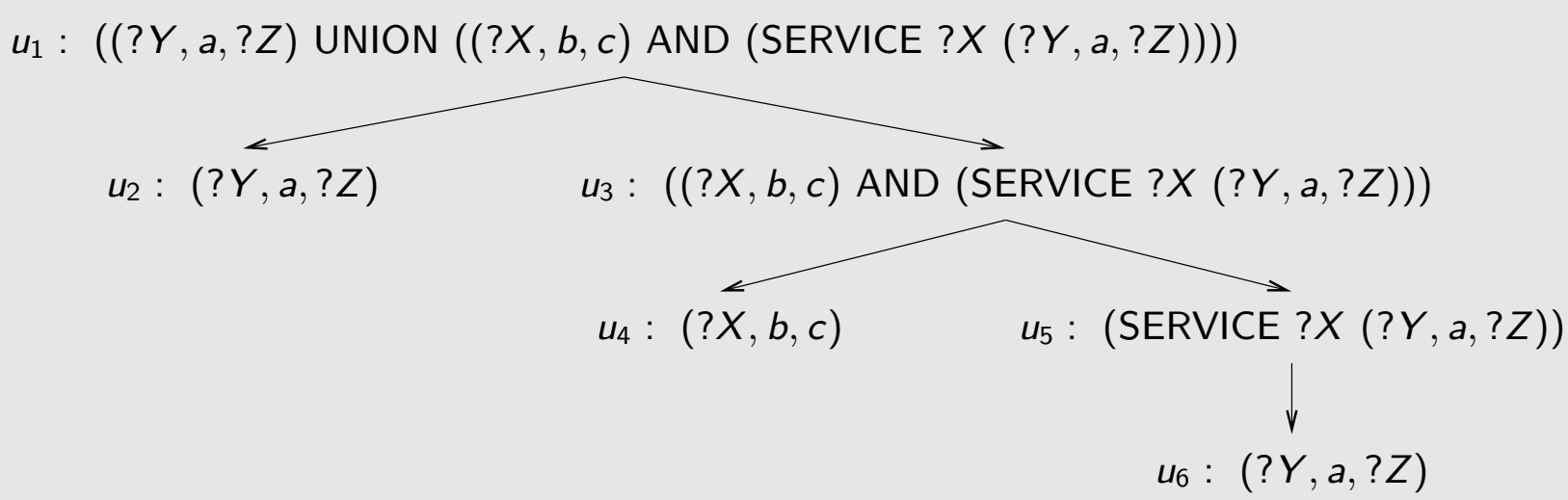
$$\Big((?Y, a, ?Z) \text{ UNION } ((?X, b, c) \text{ AND } (\text{SERVICE } ?X\ (?Y, a, ?Z)))\Big)$$



Parse tree of a pattern

We need first to formalize the tree of subexpressions of a pattern

$$\Big((?Y, a, ?Z) \text{ UNION } ((?X, b, c) \text{ AND } (\text{SERVICE } ?X\ (?Y, a, ?Z)))\Big)$$



We denote by $\mathcal{T}(P)$ the tree of subexpressions of P.

Service-boundedness

Notion of boundedness considering the important part

Definition

A pattern P is *service-bound* if for every node u in $\mathcal{T}(P)$ with label (SERVICE $?X$ P_1) it holds that:

Service-boundedness

Notion of boundedness considering the important part

Definition

A pattern P is *service-bound* if for every node u in $\mathcal{T}(P)$ with label (SERVICE $?X$ P_1) it holds that:

1. There exists an ancestor of u in $\mathcal{T}(P)$ with label P_2 s.t. $?X$ is bound in P_2, and

Service-boundedness

Notion of boundedness considering the important part

Definition

A pattern P is *service-bound* if for every node u in $\mathcal{T}(P)$ with label (SERVICE $?X$ P_1) it holds that:

1. There exists an ancestor of u in $\mathcal{T}(P)$ with label P_2 s.t. $?X$ is bound in P_2, and
2. P_1 is service-bound.

Service-boundedness

Notion of boundedness considering the important part

Definition

A pattern P is *service-bound* if for every node u in $\mathcal{T}(P)$ with label (SERVICE $?X$ P_1) it holds that:

1. There exists an ancestor of u in $\mathcal{T}(P)$ with label P_2 s.t. $?X$ is bound in P_2, and
2. P_1 is service-bound.

Unfortunately... (and not so surprisingly anymore)

Service-boundedness

Notion of boundedness considering the important part

Definition

A pattern P is *service-bound* if for every node u in $\mathcal{T}(P)$ with label (SERVICE $?X$ P_1) it holds that:

1. There exists an ancestor of u in $\mathcal{T}(P)$ with label P_2 s.t. $?X$ is bound in P_2, and
2. P_1 is service-bound.

Unfortunately... (and not so surprisingly anymore)

Theorem (BAC11)

Checking if a pattern is service-bound is undecidable.

Service-boundedness

Notion of boundedness considering the important part

Definition

A pattern P is *service-bound* if for every node u in $\mathcal{T}(P)$ with label (SERVICE $?X$ P_1) it holds that:

1. There exists an ancestor of u in $\mathcal{T}(P)$ with label P_2 s.t. $?X$ is bound in P_2, and
2. P_1 is service-bound.

Unfortunately... (and not so surprisingly anymore)

Theorem (BAC11)

Checking if a pattern is service-bound is undecidable.

Exercise: prove the theorem.

Service-safeness

We need a decidable sufficient condition.

Idea: replace *bound* by SB(·) in the previous notion.

Service-safeness

We need a decidable sufficient condition.

Idea: replace *bound* by SB($\cdot$) in the previous notion.

Definition

A pattern P is *service-safe* if of every node u in $\mathcal{T}(P)$ with label (SERVICE $?X$ P_1) it holds that:

Service-safeness

We need a decidable sufficient condition.

Idea: replace *bound* by $\mathrm{SB}(\cdot)$ in the previous notion.

Definition

A pattern P is *service-safe* if of every node u in $\mathcal{T}(P)$ with label (SERVICE $?X$ P_1) it holds that:

1. There exists an ancestor of u in $\mathcal{T}(P)$ with label P_2 s.t. $?X \in \mathrm{SB}(P_2)$, and

Service-safeness

We need a decidable sufficient condition.

Idea: replace *bound* by $\mathrm{SB}(\cdot)$ in the previous notion.

Definition

A pattern P is *service-safe* if of every node u in $\mathcal{T}(P)$ with label (SERVICE $?X$ P_1) it holds that:

1. There exists an ancestor of u in $\mathcal{T}(P)$ with label P_2 s.t. $?X \in \mathrm{SB}(P_2)$, and
2. P_1 is service-safe.

Service-safeness

We need a decidable sufficient condition.

Idea: replace *bound* by $\mathrm{SB}(\cdot)$ in the previous notion.

Definition

A pattern P is *service-safe* if of every node u in $\mathcal{T}(P)$ with label (SERVICE $?X$ P_1) it holds that:

1. There exists an ancestor of u in $\mathcal{T}(P)$ with label P_2 s.t. $?X \in \mathrm{SB}(P_2)$, and
2. P_1 is service-safe.

Proposition (BAC11)

If P is service-safe the it is service-bound.

Service-safeness

We need a decidable sufficient condition.

Idea: replace *bound* by SB($\cdot$) in the previous notion.

Definition

A pattern P is *service-safe* if of every node u in $\mathcal{T}(P)$ with label (SERVICE $?X$ P_1) it holds that:

1. There exists an ancestor of u in $\mathcal{T}(P)$ with label P_2 s.t. $?X \in \mathrm{SB}(P_2)$, and
2. P_1 is service-safe.

Proposition (BAC11)

If P is service-safe the it is service-bound.

We finally have our desired decidable condition.

Applying service-safeness

$$P_1 = \Big[(?X, \text{service\_description}, ?Z) \text{ UNION } \Big((?X, \text{service\_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y \ (?N, \text{email}, ?E)) \Big) \Big]$$

Applying service-safeness

$$P_1 = \Big[(?X, \text{service\_description}, ?Z) \text{ UNION } \Big((?X, \text{service\_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y \ (?N, \text{email}, ?E)) \Big) \Big]$$

is service-safe.

Applying service-safeness

$$P_1 = \Big[(?X, \text{service\_description}, ?Z) \text{ UNION } \Big((?X, \text{service\_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y \ (?N, \text{email}, ?E)) \Big) \Big]$$

is service-safe.

$$P_2 = \Big[(?U_1, \text{related\_with}, ?U_2) \text{ AND } \Big(\text{SERVICE } ?U_1 \ \Big((?N, \text{email}, ?E) \text{ OPT } (\text{SERVICE } ?U_2 \ (?N, \text{phone}, ?F)) \Big) \Big) \Big]$$

Applying service-safeness

$$P_1 = \Big[(?X, \text{service\_description}, ?Z) \text{ UNION } \Big((?X, \text{service\_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y \ (?N, \text{email}, ?E)) \Big) \Big]$$

is service-safe.

$$P_2 = \Big[(?U_1, \text{related\_with}, ?U_2) \text{ AND } \Big(\text{SERVICE } ?U_1 \ \Big((?N, \text{email}, ?E) \text{ OPT } (\text{SERVICE } ?U_2 \ (?N, \text{phone}, ?F)) \Big) \Big) \Big]$$

is not service-safe.

Closing words on SERVICE

SERVICE: several interesting research question

- Optimization (rewriting, reordering, containment?)
- Cost analysis: in practice we cannot query all that we may want
- Different endpoints provide different *completeness* certificates (regarding the data they return)
- Several implementations challenges

Outline

SPARQL 1.0 has very limited navigational capabilities

Assume a graph with cities and connections with RDF triples like:

$$(C_1, \text{connected}, C_2)$$

SPARQL 1.0 has very limited navigational capabilities

Assume a graph with cities and connections with RDF triples like:

$$(C_1, \text{connected}, C_2)$$

query: is city B *reachable* from A by a *sequence* of connections?

SPARQL 1.0 has very limited navigational capabilities

Assume a graph with cities and connections with RDF triples like:

$$(C_1, \text{connected}, C_2)$$

query: is city B *reachable* from A by a *sequence* of connections?

- Known fact: SPARQL 1.0 cannot express this query!
- Follows easily from locality of FO-logic

SPARQL 1.0 has very limited navigational capabilities

Assume a graph with cities and connections with RDF triples like:

$$(C_1, \text{connected}, C_2)$$

query: is city B *reachable* from A by a *sequence* of connections?

- Known fact: SPARQL 1.0 cannot express this query!
- Follows easily from locality of FO-logic

You (should) already know that Datalog can express this query.

SPARQL 1.0 has very limited navigational capabilities

Assume a graph with cities and connections with RDF triples like:

$$(C_1, \text{connected}, C_2)$$

query: is city B *reachable* from A by a *sequence* of connections?

- Known fact: SPARQL 1.0 cannot express this query!
- Follows easily from locality of FO-logic

You (should) already know that Datalog can express this query.

We can consider a new predicate *reach* and the program

SPARQL 1.0 has very limited navigational capabilities

Assume a graph with cities and connections with RDF triples like:

$$(C_1, \text{connected}, C_2)$$

query: is city B *reachable* from A by a *sequence* of connections?

- Known fact: SPARQL 1.0 cannot express this query!
- Follows easily from locality of FO-logic

You (should) already know that Datalog can express this query.

We can consider a new predicate *reach* and the program

$$(?X, \mathit{reach}, ?Y) \quad \leftarrow \quad (?X, \text{connected}, ?Y)$$

SPARQL 1.0 has very limited navigational capabilities

Assume a graph with cities and connections with RDF triples like:

$$(C_1, \text{connected}, C_2)$$

query: is city B *reachable* from A by a *sequence* of connections?

- Known fact: SPARQL 1.0 cannot express this query!
- Follows easily from locality of FO-logic

You (should) already know that Datalog can express this query.

We can consider a new predicate *reach* and the program

$$\begin{array}{lcl} (?X, \mathit{reach}, ?Y) & \leftarrow & (?X, \text{connected}, ?Y) \\ (?X, \mathit{reach}, ?Y) & \leftarrow & (?X, \mathit{reach}, ?Z),\ (?Z, \text{connected}, ?Y) \end{array}$$

SPARQL 1.0 has very limited navigational capabilities

Assume a graph with cities and connections with RDF triples like:

$$(C_1, \text{connected}, C_2)$$

query: is city B *reachable* from A by a *sequence* of connections?

- Known fact: SPARQL 1.0 cannot express this query!
- Follows easily from locality of FO-logic

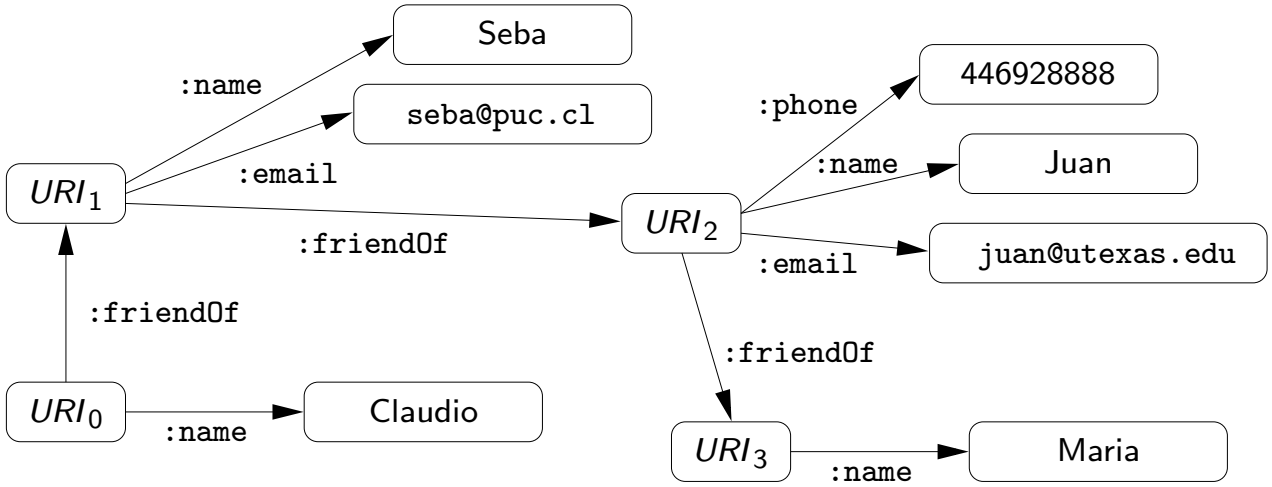
You (should) already know that Datalog can express this query.

We can consider a new predicate *reach* and the program

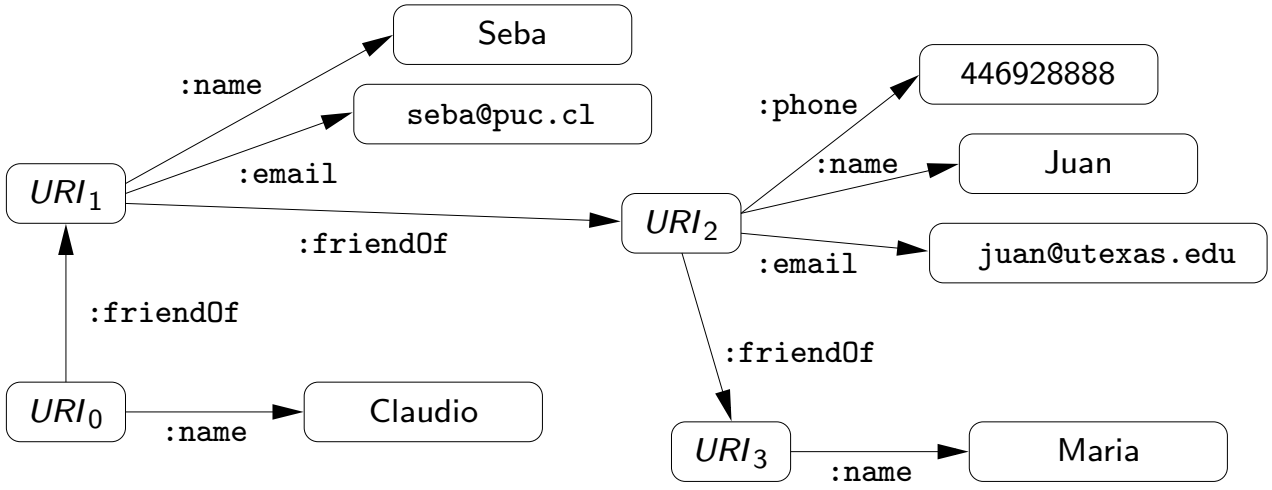
$$\begin{array}{lcl} (?X, \mathit{reach}, ?Y) & \leftarrow & (?X, \text{connected}, ?Y) \\ (?X, \mathit{reach}, ?Y) & \leftarrow & (?X, \mathit{reach}, ?Z),\ (?Z, \text{connected}, ?Y) \end{array}$$

but do we want to integrate Datalog and SPARQL?

SPARQL 1.1 provides an alternative way for navigating

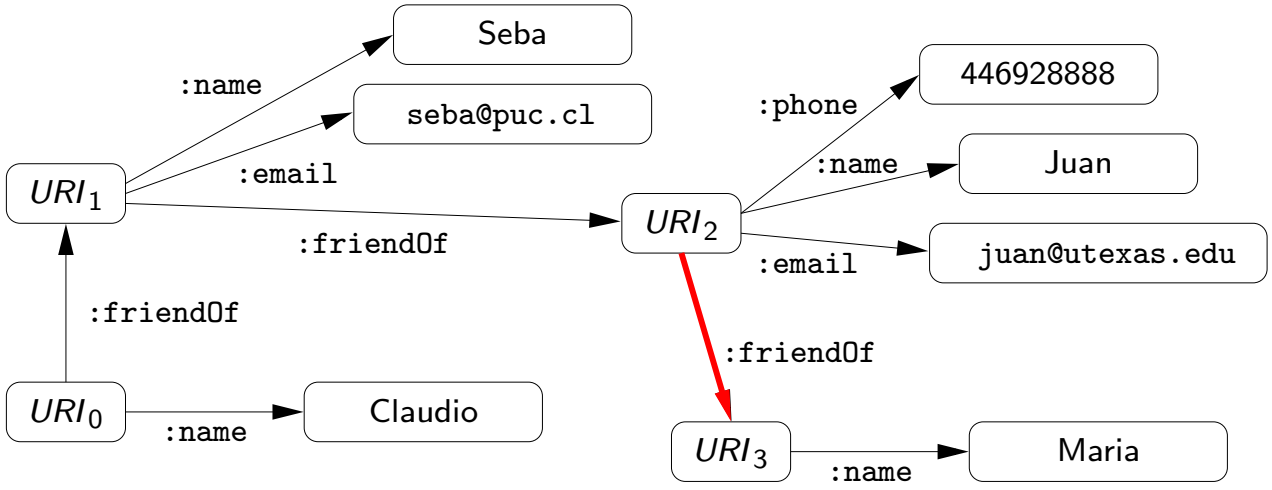


SPARQL 1.1 provides an alternative way for navigating



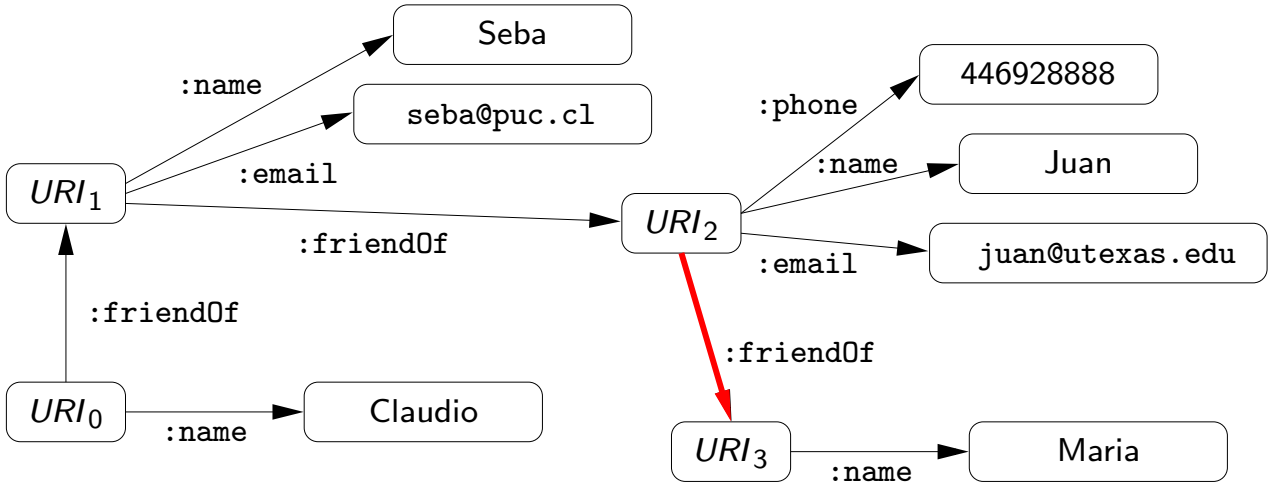

```
SELECT ?X
WHERE
{
   ?X  :friendOf   ?Y .
   ?Y  :name   "Maria" .
}
```

SPARQL 1.1 provides an alternative way for navigating



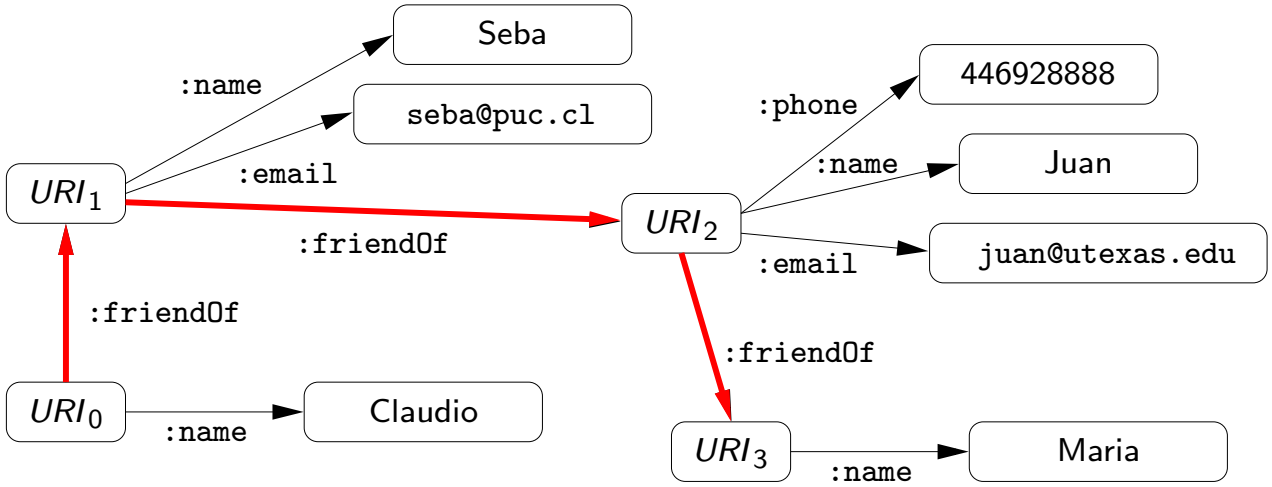

```
SELECT ?X
WHERE
{
   ?X  :friendOf   ?Y .
   ?Y  :name   "Maria" .
}
```

SPARQL 1.1 provides an alternative way for navigating



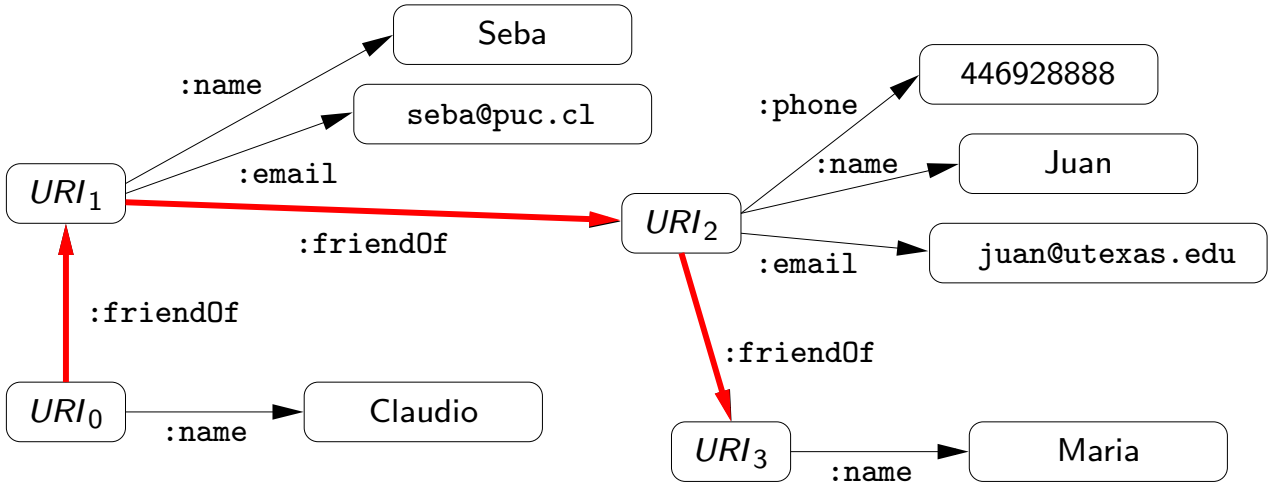

```
SELECT ?X
WHERE
{
   ?X (:friendOf)* ?Y .
   ?Y  :name   "Maria" .
}
```

SPARQL 1.1 provides an alternative way for navigating



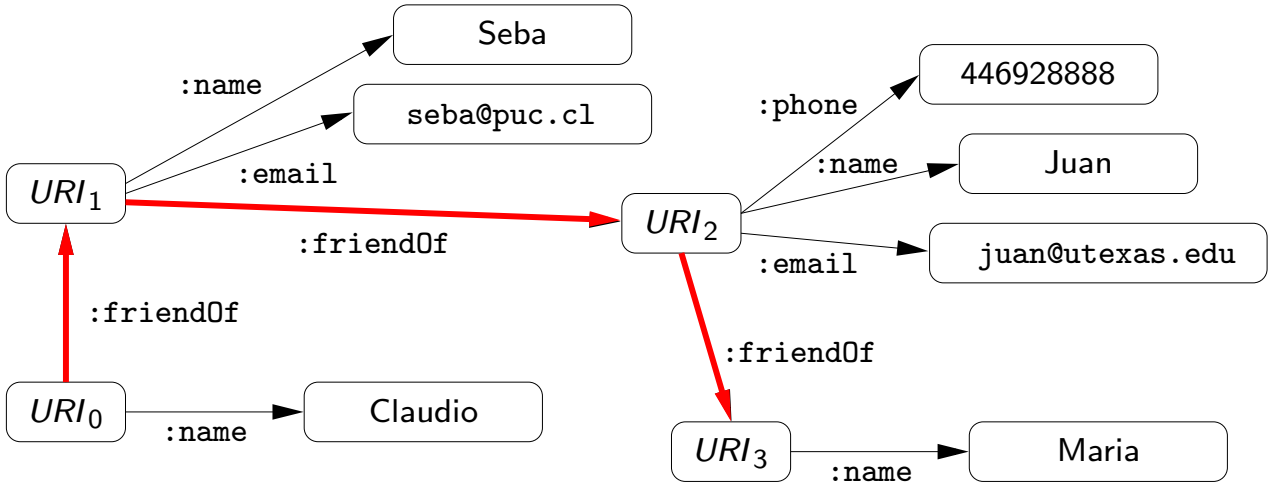

```
SELECT ?X
WHERE
{
   ?X (:friendOf)* ?Y .
   ?Y  :name   "Maria" .
}
```

SPARQL 1.1 provides an alternative way for navigating




```
SELECT ?X
WHERE
{
   ?X (:friendOf)* ?Y .      ← SPARQL 1.1 property path
   ?Y  :name   "Maria" .
}
```

SPARQL 1.1 provides an alternative way for navigating




```
SELECT ?X
WHERE
{
   ?X (:friendOf)* ?Y .      ← SPARQL 1.1 property path
   ?Y  :name   "Maria" .
}
```

Idea: navigate RDF graphs using *regular expressions*

General navigation using regular expressions

Regular expressions define sets of strings using

- concatenation: /
- disjunction: |
- Kleene star: *

Example

Consider strings composed of symbols `a` and `b`

`a/(b)*/a`

defines strings of the form $\texttt{abbb}\cdots\texttt{bbba}$.

General navigation using regular expressions

Regular expressions define sets of strings using

- concatenation: /
- disjunction: |
- Kleene star: *

Example

Consider strings composed of symbols `a` and `b`

`a/(b)*/a`

defines strings of the form `abbb`$\cdots$`bbba`.

Idea: use regular expressions to define paths

- a path `p` satisfies a regular expression `r` if the string composed of the sequence of edges of `p` satisfies expression `r`

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

ancestors of `John`

```
{ John (:father|:mother)* ?X }
```

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X Paris }`

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X (:train|:plane)* Paris }`

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X (:train|:plane)*/:bus Paris }`

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X (:train|:plane)*/:bus/(:train|:plane)* Paris }`

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X (:train|:plane)*/:bus/(:train|:plane)* Paris }`

  Exercise: cities that reach Paris with an *even number* of connections

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X (:train|:plane)*/:bus/(:train|:plane)* Paris }`

  Exercise: cities that reach Paris with an *even number* of connections

Mixing regular expressions and SPARQL operators
gives interesting expressive power:

Persons in my professional network that attended the same school

Interesting navigational queries

- RDF graph with `:father`, `:mother` edges:

  ancestors of `John`

  `{ John (:father|:mother)* ?X }`

- Connections between cities via `:train`, `:bus`, `:plane`

  Cities that reach `Paris` with exactly one `:bus` connection

  `{ ?X (:train|:plane)*/:bus/(:train|:plane)* Paris }`

  Exercise: cities that reach Paris with an *even number* of connections

Mixing regular expressions and SPARQL operators
gives interesting expressive power:

Persons in my professional network that attended the same school

```
{ ?X (:conn)* ?Y .
  ?X (:conn)* ?Z .
  ?Y :sameSchool ?Z }
```

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

A bit of history on W3C standardization of property paths:

- Mid 2010: W3C defines an informal semantics for paths

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

A bit of history on W3C standardization of property paths:

- Mid 2010: W3C defines an informal semantics for paths
- Late 2010: several discussions on possible drawbacks on the non-standard definition by the W3C

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

A bit of history on W3C standardization of property paths:

- Mid 2010: W3C defines an informal semantics for paths
- Late 2010: several discussions on possible drawbacks on the non-standard definition by the W3C
- Early 2011: first formal semantics by the W3C

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

A bit of history on W3C standardization of property paths:

- Mid 2010: W3C defines an informal semantics for paths
- Late 2010: several discussions on possible drawbacks on the non-standard definition by the W3C
- Early 2011: first formal semantics by the W3C
- Late 2011: empirical and theoretical study on SPARQL 1.1 property paths showing unfeasibility of evaluation

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

A bit of history on W3C standardization of property paths:

- Mid 2010: W3C defines an informal semantics for paths
- Late 2010: several discussions on possible drawbacks on the non-standard definition by the W3C
- Early 2011: first formal semantics by the W3C
- Late 2011: empirical and theoretical study on SPARQL 1.1 property paths showing unfeasibility of evaluation
- Mid 2012: semantics change to overcome the raised issues

As always, we need a (formal) semantics

- Regular expressions in SPARQL queries seem reasonable
- We need to agree in the meaning of these new queries

A bit of history on W3C standardization of property paths:

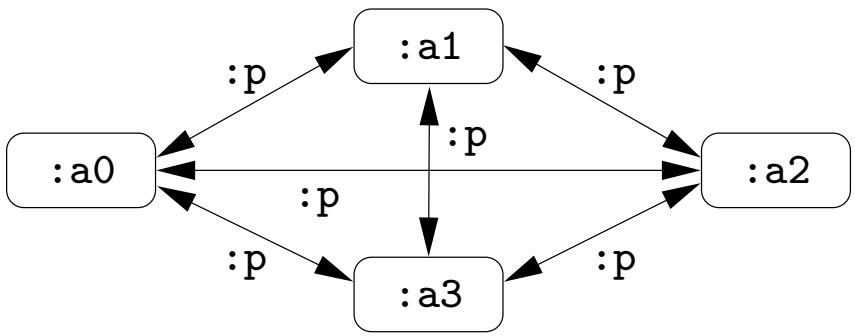
- Mid 2010: W3C defines an informal semantics for paths
- Late 2010: several discussions on possible drawbacks on the non-standard definition by the W3C
- Early 2011: first formal semantics by the W3C
- Late 2011: empirical and theoretical study on SPARQL 1.1 property paths showing unfeasibility of evaluation
- Mid 2012: semantics change to overcome the raised issues

The following experimental study is based on [ACP12].

SPARQL 1.1 implementations had a poor performance

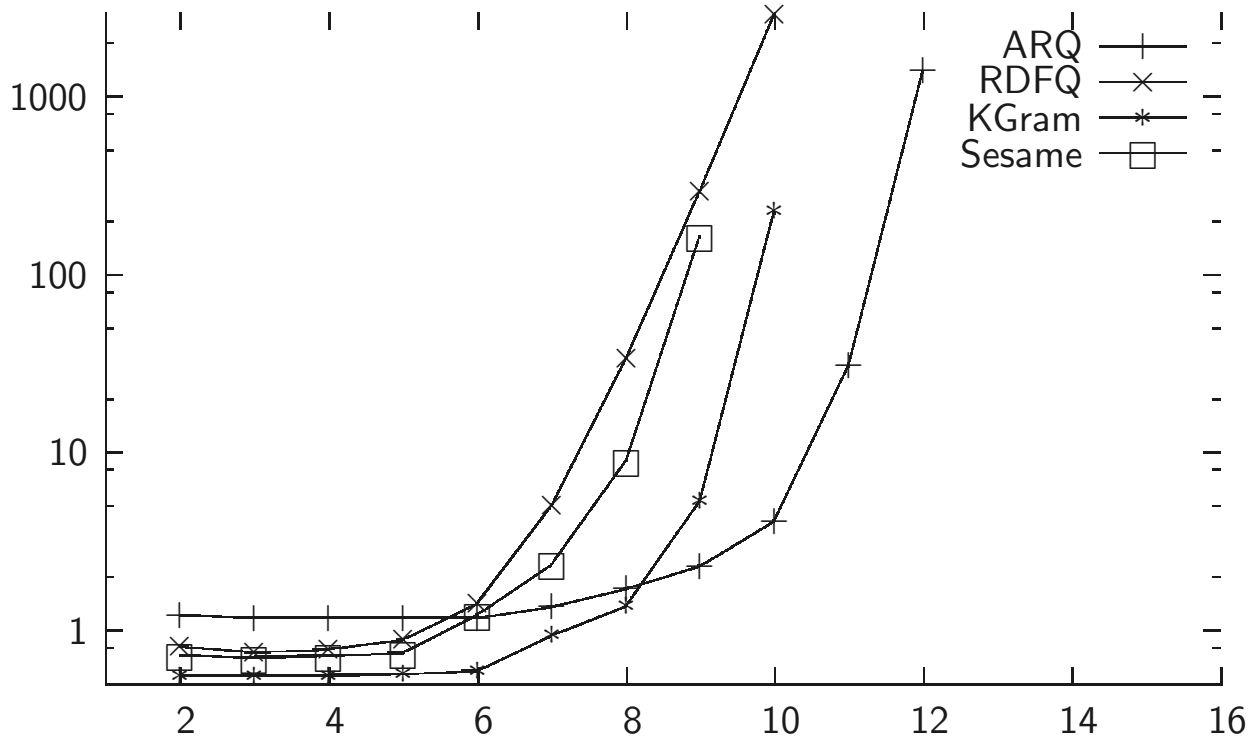
Data:

- *cliques* (complete graphs) of different size
- from 2 nodes (87 bytes) to 13 nodes (970 bytes)



RDF clique with 4 nodes (127 bytes)

SPARQL 1.1 implementations had a poor performance



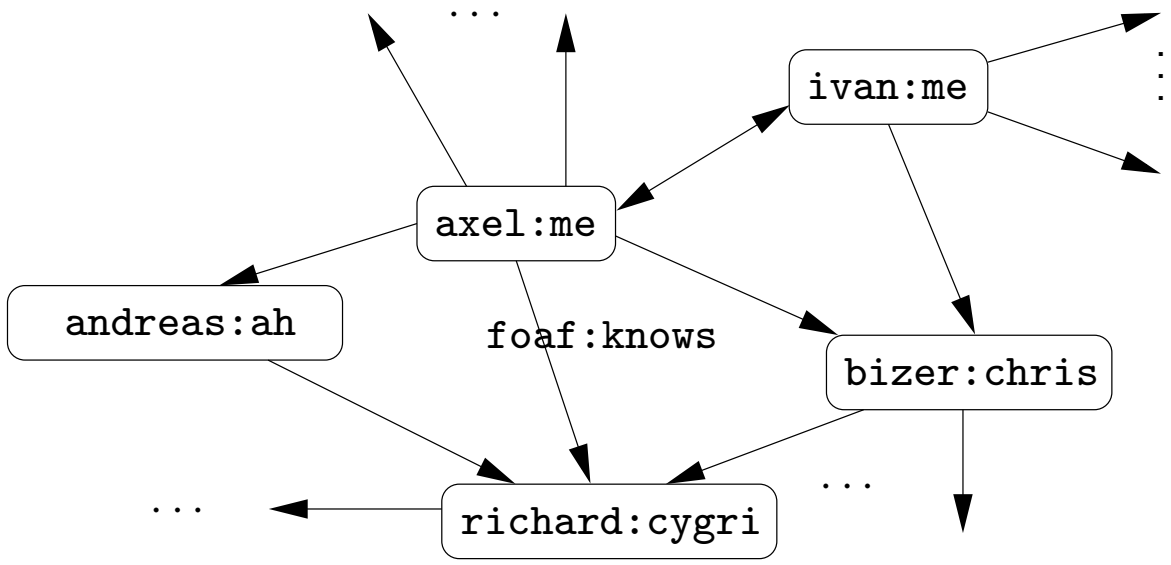
[ACP12]

```
SELECT * WHERE { :a0 (:p)* :a1 }
```

Poor performance with real Web data of small size

Data:

- Social Network data given by `foaf:knows` links
- Crawled from Axel Polleres' foaf document (3 steps)
- Different documents, deleting some nodes



Poor performance with real Web data of small size

```
SELECT * WHERE { axel:me (foaf:knows)* ?x }
```

Poor performance with real Web data of small size

```
SELECT * WHERE { axel:me (foaf:knows)* ?x }
```

Input	ARQ	RDFQ	Kgram	Sesame
9.2KB	5.13	75.70	313.37	–
10.9KB	8.20	325.83	–	–
11.4KB	65.87	–	–	–
13.2KB	292.43	–	–	–
14.8KB	–	–	–	–
17.2KB	–	–	–	–
20.5KB	–	–	–	–
25.8KB	–	–	–	–

[ACP12]

(time in seconds, timeout = 1hr)

Poor performance with real Web data of small size

```
SELECT * WHERE { axel:me (foaf:knows)* ?x }
```

Input	ARQ	RDFQ	Kgram	Sesame
9.2KB	5.13	75.70	313.37	–
10.9KB	8.20	325.83	–	–
11.4KB	65.87	–	–	–
13.2KB	292.43	–	–	–
14.8KB	–	–	–	–
17.2KB	–	–	–	–
20.5KB	–	–	–	–
25.8KB	–	–	–	–

[ACP12]

(time in seconds, timeout = 1hr)

Is this a problem of these particular implementations?

This is a problem of the specification

[ACP12]

Any implementation that follows SPARQL 1.1 standard (as of January 2012) is doomed to show the same behavior

This is a problem of the specification

[ACP12]

Any implementation that follows SPARQL 1.1 standard (as of January 2012) is doomed to show the same behavior

The main sources of complexity is *counting*

This is a problem of the specification

[ACP12]

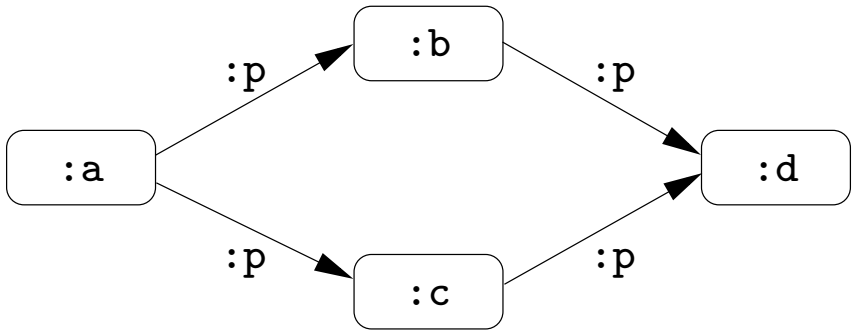
Any implementation that follows SPARQL 1.1 standard (as of January 2012) is doomed to show the same behavior

The main sources of complexity is *counting*

Impact on W3C standard:

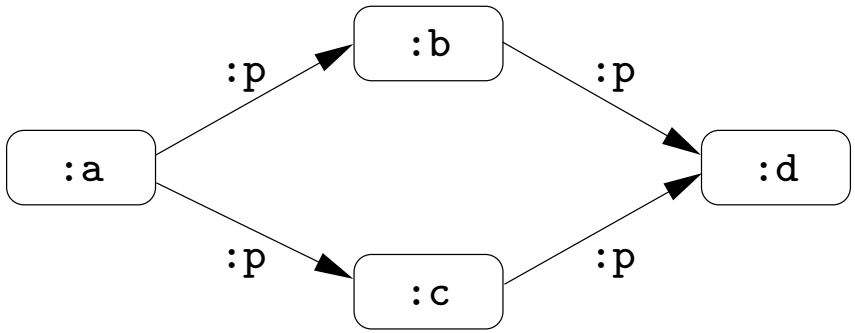
- Standard semantics of SPARQL 1.1 property paths was changed in *July 2012* to overcome these issues

SPARQL 1.1 property paths match regular expressions but also *count*




```
SELECT ?X
WHERE { :a  (:p)*  ?X }
```

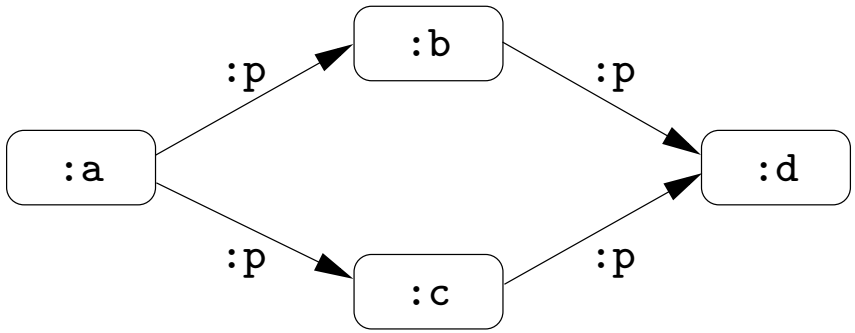
SPARQL 1.1 property paths match regular expressions but also *count*




```
SELECT ?X
WHERE { :a  (:p)*  ?X }
```

?X
:a
:b
:c
:d
:d

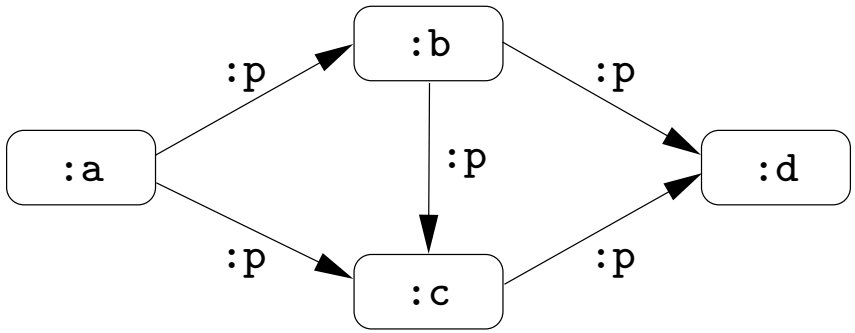
SPARQL 1.1 property paths match regular expressions but also *count*




```
SELECT ?X
WHERE { :a  (:p)*  ?X }
```

?X
:a
:b
:c
:d
:d

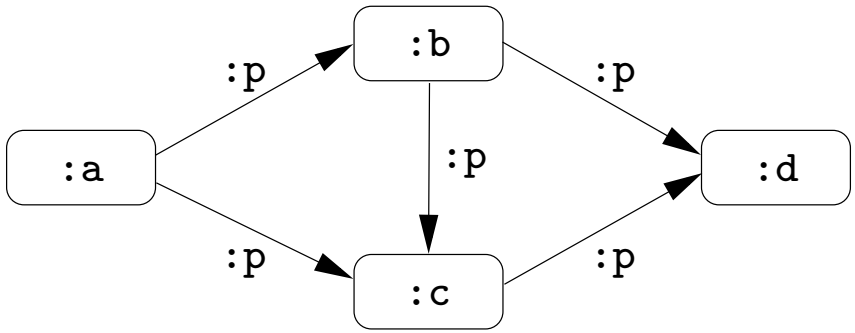
SPARQL 1.1 property paths match regular expressions but also *count*




```
SELECT ?X
WHERE { :a  (:p)*  ?X }
```

?X
:a
:b
:c
:d
:d

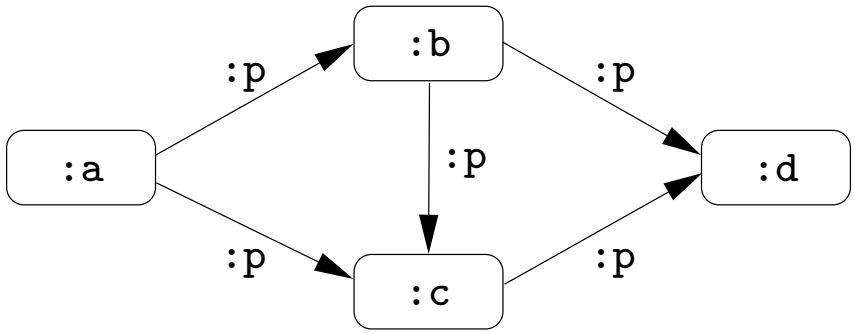
SPARQL 1.1 property paths match regular expressions but also *count*




```
SELECT ?X
WHERE { :a  (:p)*  ?X }
```

?X
:a
:b
:c
:d
:d
:c
:d

SPARQL 1.1 property paths match regular expressions but also *count*




```
SELECT ?X
WHERE { :a  (:p)*  ?X }
```

?X
:a
:b
:c
:d
:d
:c
:d

But what if we have cycles?

SPARQL 1.1 document provides a special procedure to handle cycles (and make the count)

Evaluation of *path∗*

"the algorithm extends the multiset of results by one application of path. If a node has been visited for path, it is not a candidate for another step. A node can be visited multiple times if different paths visit it."

SPARQL 1.1 Last Call (Jan 2012)

SPARQL 1.1 document provides a special procedure to handle cycles (and make the count)

Evaluation of *path∗*

"the algorithm extends the multiset of results by one application of path. If a node has been visited for path, it is not a candidate for another step. A node can be visited multiple times if different paths visit it."

SPARQL 1.1 Last Call (Jan 2012)

- W3C document provides a procedure (`ArbitraryLengthPath`)
- This procedure was formalized in [ACP12]

Counting the number of solutions...

Data: Clique of size n

```
{ :a0 (:p)* :a1 }
```

every solution is a copy of the *empty mapping* $\mu_\emptyset$ (| | in ARQ)

Counting the number of solutions...

Data: Clique of size n

```
{ :a0 (:p)* :a1 }
```

n	# Sol.
9	13,700
10	109,601
11	986,410
12	9,864,101
13	–

every solution is a copy of the *empty mapping* $\mu_\emptyset$ (| | in ARQ)

Counting the number of solutions...

Data: Clique of size n

```
{ :a0 (:p)* :a1 }      { :a0 ((:p)*)* :a1 }
```

n	# Sol.
9	13,700
10	109,601
11	986,410
12	9,864,101
13	–

every solution is a copy of the *empty mapping* $\mu_\emptyset$ (| | in ARQ)

Counting the number of solutions...

Data: Clique of size n

`{ :a0 (:p)* :a1 }`

n	# Sol.
9	13,700
10	109,601
11	986,410
12	9,864,101
13	–

`{ :a0 ((:p)*)* :a1 }`

n	# Sol
2	1
3	6
4	305
5	418,576
6	–

every solution is a copy of the *empty mapping* $\mu_\emptyset$ (| | in ARQ)

Counting the number of solutions...

Data: Clique of size n

`{ :a0 (:p)* :a1 }` `{ :a0 ((:p)*)* :a1 }` `{ :a0 (((:p)*)*)* :a1 }`

n	# Sol.
9	13,700
10	109,601
11	986,410
12	9,864,101
13	–

n	# Sol
2	1
3	6
4	305
5	418,576
6	–

every solution is a copy of the *empty mapping* $\mu_\emptyset$ (| | in ARQ)

Counting the number of solutions...

Data: Clique of size n

`{ :a0 (:p)* :a1 }`

n	# Sol.
9	13,700
10	109,601
11	986,410
12	9,864,101
13	–

`{ :a0 ((:p)*)* :a1 }`

n	# Sol
2	1
3	6
4	305
5	418,576
6	–

`{ :a0 (((:p)*)*)* :a1 }`

n	# Sol.
2	1
3	42
4	–

every solution is a copy of the *empty mapping* $\mu_\emptyset$ (| | in ARQ)

More on counting the number of solutions...

Data: foaf links crawled from the Web

```
{ axel:me (foaf:knows)* ?x }
```

More on counting the number of solutions...

Data: foaf links crawled from the Web

```
{ axel:me (foaf:knows)* ?x }
```

File	# URIs	# Sol.	Output Size
9.2KB	38	29,817	2MB
10.9KB	43	122,631	8.4MB
11.4KB	47	1,739,331	120MB
13.2KB	52	8,511,943	587MB
14.8KB	54	–	–

More on counting the number of solutions...

Data: foaf links crawled from the Web

```
{ axel:me (foaf:knows)* ?x }
```

File	# URIs	# Sol.	Output Size
9.2KB	38	29,817	2MB
10.9KB	43	122,631	8.4MB
11.4KB	47	1,739,331	120MB
13.2KB	52	8,511,943	587MB
14.8KB	54	–	–

What is really happening here?

More on counting the number of solutions...

Data: foaf links crawled from the Web

```
{ axel:me (foaf:knows)* ?x }
```

File	# URIs	# Sol.	Output Size
9.2KB	38	29,817	2MB
10.9KB	43	122,631	8.4MB
11.4KB	47	1,739,331	120MB
13.2KB	52	8,511,943	587MB
14.8KB	54	–	–

What is really happening here?

Theory can help!

A bit more on complexity classes...

Complexity can be measured by using *counting-complexity classes*

NP	**#P**
SAT: is a propositional formula satisfiable?	COUNTSAT: how many assignments satisfy a propositional formula?

A bit more on complexity classes...

Complexity can be measured by using *counting-complexity classes*

NP	**#P**
SAT: is a propositional formula satisfiable?	COUNTSAT: how many assignments satisfy a propositional formula?

Formally

A function $f(\cdot)$ on strings is in **#P** if there exists a polynomial-time non-deterministic TM M such that

$$f(x) = \text{number of accepting computations of } M \text{ with input } x$$

A bit more on complexity classes...

Complexity can be measured by using *counting-complexity classes*

NP	**#P**
SAT: is a propositional formula satisfiable?	COUNTSAT: how many assignments satisfy a propositional formula?

Formally

A function $f(\cdot)$ on strings is in **#P** if there exists a polynomial-time non-deterministic TM M such that

$$f(x) = \text{number of accepting computations of } M \text{ with input } x$$

- COUNTSAT is **#P**-complete

Counting problem for property paths

CountW3C

Input: RDF graph G
Property path triple $\{$ `:a` *path* `:b` $\}$

Output: Count the number of solutions of $\{$ `:a` *path* `:b` $\}$ over G
(according to the semantics proposed by W3C)

The complexity of property paths is *intractable*

Theorem (ACP12)

$\textsc{CountW3C}$ *is outside* **#P**

The complexity of property paths is *intractable*

Theorem (ACP12)

$\textsc{CountW3C}$ *is outside* **#P**

$\textsc{CountW3C}$ is hard to solve even if **P** = **NP**

A *doubly exponential* lower bound for counting

- Let $path_s$ be a property path of the form

$$(\cdots((:p)*)*)\cdots)*$$

with s nested stars

A *doubly exponential* lower bound for counting

- Let $path_s$ be a property path of the form

$$(\cdots((:p)*)*)\cdots)*$$

  with s nested stars
- Let K_n be a clique with n nodes

A *doubly exponential* lower bound for counting

- Let $path_s$ be a property path of the form

  $$(\cdots((\texttt{:p})*)*)\cdots)*$$

  with s nested stars
- Let K_n be a clique with n nodes
- Let $CountClique(s, n)$ be the number of solutions of $\{\ \texttt{:a0}\ \ path_s\ \ \texttt{:a1}\ \}$ over K_n

A *doubly exponential* lower bound for counting

- Let $path_s$ be a property path of the form

$$(\cdots((\texttt{:p})*)*)\cdots)*$$

  with s nested stars
- Let K_n be a clique with n nodes
- Let $CountClique(s, n)$ be the number of solutions of $\{\ \texttt{:a0}\ \ path_s\ \ \texttt{:a1}\ \}$ over K_n

Lemma (ACP12)

$$CountClique(s, n) \quad \geq \quad (n-2)!^{(n-1)^{s-1}}$$

A *doubly exponential* lower bound for counting

- Let $path_s$ be a property path of the form

$$(\cdots((\texttt{:p})*)*)\cdots)*$$

  with s nested stars
- Let K_n be a clique with n nodes
- Let $CountClique(s, n)$ be the number of solutions of $\{\ \texttt{:a0}\ \ path_s\ \ \texttt{:a1}\ \}$ over K_n

Lemma (ACP12)

$$CountClique(s, n) \quad \geq \quad (n-2)!^{(n-1)^{s-1}}$$

In [ACP12]: A recursive formula for calculating $CountClique(s, n)$

We can explain the experimental results

CountClique(s, n) allows to *fill in the blanks*

```
{ :a0 ((:p)*)* :a1 }
```

n	# Sol.
2	1
3	6
4	305
5	418,576
6	–
7	–
8	–

We can explain the experimental results

CountClique(s, n) allows to *fill in the blanks*

$\{$ `:a0 ((:p)*)* :a1` $\}$

n	# Sol.	
2	1	✓
3	6	
4	305	
5	418,576	
6	–	
7	–	
8	–	

We can explain the experimental results

CountClique(s, n) allows to *fill in the blanks*

$\{$ `:a0 ((:p)*)* :a1` $\}$

n	# Sol.	
2	1	✓
3	6	✓
4	305	
5	418,576	
6	–	
7	–	
8	–	

We can explain the experimental results

CountClique(s, n) allows to *fill in the blanks*

```
{ :a0 ((:p)*)* :a1 }
```

n	# Sol.	
2	1	✓
3	6	✓
4	305	✓
5	418,576	
6	–	
7	–	
8	–	

We can explain the experimental results

CountClique(s, n) allows to *fill in the blanks*

`{ :a0 ((:p)*)* :a1 }`

n	# Sol.	
2	1	✓
3	6	✓
4	305	✓
5	418,576	✓
6	–	
7	–	
8	–	

We can explain the experimental results

CountClique(s, n) allows to *fill in the blanks*

$\{$ `:a0 ((:p)*)* :a1` $\}$

n	# Sol.		
2	1	✓	
3	6	✓	
4	305	✓	
5	418,576	✓	
6	–	$\leftarrow$	28×10^9
7	–		
8	–		

We can explain the experimental results

CountClique(s, n) allows to *fill in the blanks*

$\{$ `:a0 ((:p)*)* :a1` $\}$

n	# Sol.		
2	1	✓	
3	6	✓	
4	305	✓	
5	418,576	✓	
6	–	$\leftarrow$	28×10^9
7	–	$\leftarrow$	144×10^{15}
8	–		

We can explain the experimental results

CountClique(s, n) allows to *fill in the blanks*

$\{$ `:a0 ((:p)*)* :a1` $\}$

n	# Sol.		
2	1	✓	
3	6	✓	
4	305	✓	
5	418,576	✓	
6	–	$\leftarrow$	28×10^9
7	–	$\leftarrow$	144×10^{15}
8	–	$\leftarrow$	79×10^{24}

We can explain the experimental results

CountClique(s, n) allows to *fill in the blanks*

$$\{\texttt{:a0 ((:p)*)* :a1}\}$$

n	# Sol.		
2	1	✓	
3	6	✓	
4	305	✓	
5	418,576	✓	
6	–	←	28×10^9
7	–	←	144×10^{15}
8	–	←	79×10^{24}

79 Yottabytes for the answer over a file of 379 bytes

We can explain the experimental results

CountClique(s, n) allows to *fill in the blanks*

$$\{\texttt{:a0 ((:p)*)* :a1}\}$$

n	# Sol.		
2	1	✓	
3	6	✓	
4	305	✓	
5	418,576	✓	
6	–	$\leftarrow$	28×10^9
7	–	$\leftarrow$	144×10^{15}
8	–	$\leftarrow$	79×10^{24}

79 Yottabytes for the answer over a file of 379 bytes

1 Yottabyte $>$ the estimated capacity of all digital storage in the world

Data complexity of property path is still intractable

Common assumption in Databases:

- queries are much smaller than data sources

Data complexity of property path is still intractable

Common assumption in Databases:

- queries are much smaller than data sources

Data complexity

- measure the complexity considering the query fixed

Data complexity of property path is still intractable

Common assumption in Databases:

- queries are much smaller than data sources

Data complexity

- measure the complexity considering the query fixed
- Data complexity of SQL, XPath, SPARQL 1.0, etc. are all polynomial

Data complexity of property path is still intractable

Common assumption in Databases:

- queries are much smaller than data sources

Data complexity

- measure the complexity considering the query fixed
- Data complexity of SQL, XPath, SPARQL 1.0, etc. are all polynomial

Theorem (ACP12)

Data complexity of $\textsc{CountW3C}$ is **#P**-complete

Data complexity of property path is still intractable

Common assumption in Databases:

- queries are much smaller than data sources

Data complexity

- measure the complexity considering the query fixed
- Data complexity of SQL, XPath, SPARQL 1.0, etc. are all polynomial

Theorem (ACP12)

Data complexity of $\textsc{CountW3C}$ is **#P**-complete

Corollary

SPARQL 1.1 query evaluation (as of January 2012) is intractable in Data Complexity

Existential semantics: a possible alternative

Possible solution

Do not count

Just check whether *there exists* a path satisfying the property path expression

Existential semantics: a possible alternative

Possible solution

Do not count

Just check whether *there exists* a path satisfying the property path expression

Years of experiences (theory and practice) in:

- Graph Databases
- XML
- SPARQL 1.0 (PSPARQL, Gleen)

+ equivalent regular expressions giving equivalent results

Existential semantics: decision problems

Input: RDF graph G
Property path triple $\{$ `:a` *path* `:b` $\}$

ExistsPath

Question: Is there a path from `:a` to `:b` in G satisfying the regular expression *path*?

ExistsW3C

Question: Is the number of solutions of $\{$ `:a` *path* `:b` $\}$ over G greater than 0 (according to W3C semantics)?

Evaluating existential paths is tractable

Theorem (well-known result)

ExistsPath can be solved in $O(|G| \times |path|)$

Can be proved by using automata theory:

Evaluating existential paths is tractable

Theorem (well-known result)

ExistsPath can be solved in $O(|G| \times |path|)$

Can be proved by using automata theory:

1. consider G as an NFA with `:a` initial state and `:b` final state

Evaluating existential paths is tractable

Theorem (well-known result)

$\textsc{ExistsPath}$ can be solved in $O(|G| \times |path|)$

Can be proved by using automata theory:

1. consider G as an NFA with `:a` initial state and `:b` final state
2. construct from $path$ an NFA $\mathcal{A}_{path}$

Evaluating existential paths is tractable

Theorem (well-known result)

ExistsPath can be solved in $O(|G| \times |path|)$

Can be proved by using automata theory:

1. consider G as an NFA with `:a` initial state and `:b` final state
2. construct from *path* an NFA $\mathcal{A}_{path}$
3. construct the *product automaton* $G \times \mathcal{A}_{path}$:

Evaluating existential paths is tractable

Theorem (well-known result)

ExistsPath can be solved in $O(|G| \times |path|)$

Can be proved by using automata theory:

1. consider G as an NFA with `:a` initial state and `:b` final state
2. construct from *path* an NFA $\mathcal{A}_{path}$
3. construct the *product automaton* $G \times \mathcal{A}_{path}$:
   - whenever $(x, r, y) \in G$ and (p, r, q) is a transition in $\mathcal{A}_{path}$ add a transition $((x, p), r, (y, q))$ to $G \times \mathcal{A}_{path}$

Evaluating existential paths is tractable

Theorem (well-known result)

ExistsPath can be solved in $O(|G| \times |path|)$

Can be proved by using automata theory:

1. consider G as an NFA with `:a` initial state and `:b` final state
2. construct from *path* an NFA $\mathcal{A}_{path}$
3. construct the *product automaton* $G \times \mathcal{A}_{path}$:
   - whenever $(x, r, y) \in G$ and (p, r, q) is a transition in $\mathcal{A}_{path}$ add a transition $((x, p), r, (y, q))$ to $G \times \mathcal{A}_{path}$
4. check if we can go from $(\texttt{:a}, q_0)$ to $(\texttt{:b}, q_f)$ in $G \times \mathcal{A}_{path}$

Evaluating existential paths is tractable

Theorem (well-known result)

ExistsPath can be solved in $O(|G| \times |path|)$

Can be proved by using automata theory:

1. consider G as an NFA with `:a` initial state and `:b` final state
2. construct from *path* an NFA $\mathcal{A}_{path}$
3. construct the *product automaton* $G \times \mathcal{A}_{path}$:
   - whenever $(x, r, y) \in G$ and (p, r, q) is a transition in $\mathcal{A}_{path}$ add a transition $((x, p), r, (y, q))$ to $G \times \mathcal{A}_{path}$
4. check if we can go from $(\texttt{:a}, q_0)$ to $(\texttt{:b}, q_f)$ in $G \times \mathcal{A}_{path}$ with q_0 initial state of $\mathcal{A}_{path}$ and q_f some final state of $\mathcal{A}_{path}$

Relationship between ExistsPath and ExistsW3C

Theorem (ACP12)

ExistsPath and ExistsW3C are *equivalent* decision problems

Relationship between ExistsPath and ExistsW3C

Theorem (ACP12)

ExistsPath and ExistsW3C are *equivalent* decision problems

Corollary (ACP12)

ExistsW3C can be solved in $O(|G| \times |path|)$

So there are possibilities for optimization

SPARQL includes an operator to eliminate duplicates (`DISTINCT`)

So there are possibilities for optimization

SPARQL includes an operator to eliminate duplicates (`DISTINCT`)

Corollary

Property path queries with `SELECT DISTINCT`
can be efficiently evaluated

So there are possibilities for optimization

SPARQL includes an operator to eliminate duplicates (`DISTINCT`)

Corollary

Property path queries with `SELECT DISTINCT` can be efficiently evaluated

And we can also use `DISTINCT` over general queries

Theorem

`SELECT DISTINCT` SPARQL 1.1 queries are tractable in Data Complexity

SPARQL 1.1 implementations
do not take advantage of `SELECT DISTINCT`

```
SELECT DISTINCT * WHERE { axel:me (foaf:knows)* ?x }
```

Input	ARQ	RDFQ	Kgram	Sesame	Psparql	Gleen

SPARQL 1.1 implementations
do not take advantage of `SELECT DISTINCT`

```
SELECT DISTINCT * WHERE { axel:me (foaf:knows)* ?x }
```

Input	ARQ	RDFQ	Kgram	Sesame	Psparql	Gleen

SPARQL 1.1 implementations do not take advantage of `SELECT DISTINCT`

```
SELECT DISTINCT * WHERE { axel:me (foaf:knows)* ?x }
```

Input	ARQ	RDFQ	Kgram	Sesame	Psparql	Gleen
9.2KB	2.24	47.31	2.37	–	0.29	1.39
10.9KB	2.60	204.95	6.43	–	0.30	1.32
11.4KB	6.88	3222.47	80.73	–	0.30	1.34
13.2KB	24.42	–	394.61	–	0.31	1.38
14.8KB	–	–	–	–	0.33	1.38
17.2KB	–	–	–	–	0.35	1.42
20.5KB	–	–	–	–	0.44	1.50
25.8KB	–	–	–	–	0.45	1.52

SPARQL 1.1 implementations do not take advantage of `SELECT DISTINCT`

```
SELECT DISTINCT * WHERE { axel:me (foaf:knows)* ?x }
```

Input	ARQ	RDFQ	Kgram	Sesame	Psparql	Gleen
9.2KB	2.24	47.31	2.37	–	0.29	1.39
10.9KB	2.60	204.95	6.43	–	0.30	1.32
11.4KB	6.88	3222.47	80.73	–	0.30	1.34
13.2KB	24.42	–	394.61	–	0.31	1.38
14.8KB	–	–	–	–	0.33	1.38
17.2KB	–	–	–	–	0.35	1.42
20.5KB	–	–	–	–	0.44	1.50
25.8KB	–	–	–	–	0.45	1.52

Optimization possibilities can remain hidden in a complicated specification

New semantics for property paths (July 2012)

- Paths constructed only from / and | should be counted
- As soon as * is used, duplicates are eliminated (from that part of the expression)

New semantics for property paths (July 2012)

- Paths constructed only from / and | should be counted
- As soon as * is used, duplicates are eliminated (from that part of the expression)

For example, consider $path_1$, $path_2$, and $path_3$ not using * then when evaluating

$$\{ \texttt{:a} \ path_1/(path_2)*/path_3 \ \texttt{:b} \}$$

one should (intuitively):

New semantics for property paths (July 2012)

- Paths constructed only from / and | should be counted
- As soon as * is used, duplicates are eliminated (from that part of the expression)

For example, consider $path_1$, $path_2$, and $path_3$ not using * then when evaluating

$$\{ \texttt{:a} \ path_1/(path_2)*/path_3 \ \texttt{:b} \}$$

one should (intuitively):

- consider *all the paths* from :a to some intermediate :c1 satisfying $path_1$

New semantics for property paths (July 2012)

- Paths constructed only from / and | should be counted
- As soon as * is used, duplicates are eliminated (from that part of the expression)

For example, consider $path_1$, $path_2$, and $path_3$ not using * then when evaluating

$$\{ \texttt{:a} \ path_1/(path_2)*/path_3 \ \texttt{:b} \}$$

one should (intuitively):

- consider *all the paths* from :a to some intermediate :c1 satisfying $path_1$
- check if there exists :c2 reachable from :c1 following $(path_2)*$

New semantics for property paths (July 2012)

- Paths constructed only from / and | should be counted
- As soon as * is used, duplicates are eliminated (from that part of the expression)

For example, consider $path_1$, $path_2$, and $path_3$ not using * then when evaluating

$$\{ \texttt{:a}\ \ path_1/(path_2)*/path_3\ \ \texttt{:b}\ \}$$

one should (intuitively):

- consider *all the paths* from :a to some intermediate :c1 satisfying $path_1$
- check if there exists :c2 reachable from :c1 following $(path_2)*$
- consider *all the paths* from :c2 to :b satisfying $path_3$

New semantics for property paths (July 2012)

- Paths constructed only from / and | should be counted
- As soon as * is used, duplicates are eliminated (from that part of the expression)

For example, consider $path_1$, $path_2$, and $path_3$ not using * then when evaluating

$$\{ \texttt{:a} \ path_1/(path_2)*/path_3 \ \texttt{:b} \}$$

one should (intuitively):

- consider *all the paths* from :a to some intermediate :c1 satisfying $path_1$
- check if there exists :c2 reachable from :c1 following $(path_2)*$
- consider *all the paths* from :c2 to :b satisfying $path_3$
- make the count (to produce copies)

Is the new semantics the right one?

W3C definitely wants to count paths.
Are there more reasonable alternatives?

Is the new semantics the right one?

W3C definitely wants to count paths.
Are there more reasonable alternatives?

- to have different operators for counting (e.g. `.` and `||`) so the user can decide.

Is the new semantics the right one?

W3C definitely wants to count paths.
Are there more reasonable alternatives?

- to have different operators for counting (e.g. `.` and `||`) so the user can decide.
- the honest approach: just make the count and output *infininty* in the presence of cycles

Is the new semantics the right one?

W3C definitely wants to count paths.
Are there more reasonable alternatives?

- to have different operators for counting (e.g. `.` and `||`) so the user can decide.
- the honest approach: just make the count and output *infininty* in the presence of cycles
- count the number of *simple paths*

Is the new semantics the right one?

W3C definitely wants to count paths.
Are there more reasonable alternatives?

- to have different operators for counting (e.g. `.` and `||`) so the user can decide.
- the honest approach: just make the count and output *infininty* in the presence of cycles
- count the number of *simple paths*
- does someone from the audience have another in mind?

Is the new semantics the right one?

W3C definitely wants to count paths.
Are there more reasonable alternatives?

- to have different operators for counting (e.g. `.` and `||`) so the user can decide.
- the honest approach: just make the count and output *infininty* in the presence of cycles
- count the number of *simple paths*
- does someone from the audience have another in mind?

In all the above cases you have to decide what exactly you count:

Is the new semantics the right one?

W3C definitely wants to count paths.
Are there more reasonable alternatives?

- to have different operators for counting (e.g. `.` and `||`) so the user can decide.
- the honest approach: just make the count and output *infininty* in the presence of cycles
- count the number of *simple paths*
- does someone from the audience have another in mind?

In all the above cases you have to decide what exactly you count:

- the number of paths satisfying the expression?
- the number of ways that the expr can be satisfied? (W3C)

Is the new semantics the right one?

W3C definitely wants to count paths.
Are there more reasonable alternatives?

- to have different operators for counting (e.g. `.` and `||`) so the user can decide.
- the honest approach: just make the count and output *infininty* in the presence of cycles
- count the number of *simple paths*
- does someone from the audience have another in mind?

In all the above cases you have to decide what exactly you count:

- the number of paths satisfying the expression?
- the number of ways that the expr can be satisfied? (W3C)

(they are not the same! consider for example `(a|a)`)

Several work to do on navigational queries for graphs!

Other lines of research with open questions:

- Nested regular expressions and nSPARQL [PAG09,BPR12]
- Queries that can output paths (e.g. ECRPQs [BLHW10])
- More complexity results on counting paths [LM12]

What is the right language (and the right semantics)
for navigating RDF graphs?

Outline

Concluding remarks

- Federation and Navigation are fundamental features in SPARQL 1.1
- They need formalization and (serious) study

Concluding remarks

- Federation and Navigation are fundamental features in SPARQL 1.1
- They need formalization and (serious) study
- Do not runaway from Theory! it can really help (and has helped) to understand the implications of design decisions

Concluding remarks

- Federation and Navigation are fundamental features in SPARQL 1.1
- They need formalization and (serious) study
- Do not runaway from Theory! it can really help (and has helped) to understand the implications of design decisions

Big challenge:

- Can we integrate everything to effectively query Linked Data?
- Is SPARQL 1.1 the ultimate query language for Linked Data?

Concluding remarks

- Federation and Navigation are fundamental features in SPARQL 1.1
- They need formalization and (serious) study
- Do not runaway from Theory! it can really help (and has helped) to understand the implications of design decisions

Big challenge:

- Can we integrate everything to effectively query Linked Data?
- Is SPARQL 1.1 the ultimate query language for Linked Data?

A lot of work to do, so lets start! and I'm happy to collaborate! :-)

Federation and Navigation in SPARQL 1.1

Jorge Pérez

Assistant Professor
Department of Computer Science
Universidad de Chile

References

ACP12 Counting Beyond a Yottabyte ..., WWW 2012

AG08 The Expressive Power of SPARQL, ISWC 2008

BAC11 Sem & Opt of SPARQL 1.1 Federation Extensions, ISWC 2011

BLHMW10 Expressive Query Languages for Path Queries, PODS 2010

BPR12 Relative Expressiveness of Nested Regular Expressions, AMW 2012

LPSS12 Static Analysis and Optimization of SemWeb Queries, PODS 2012

LM12 Complexity of Evaluating Path Expressions in SPARQL, PODS 2012

PAG06-09 Semantics and Complexity of SPARQL, ISWC 2006, TODS 2009