Friedrich-Alexander-Universität
Fachbereich Wirtschafts- und
Sozialwissenschaften | WiSo

Career Day 2022 on Thursday, June 21st at the WiSo Campus, Lange Gasse 20

Gain information about job applications and career entry!

– **Company Contact Fair:** Getting to know various national and international companies and making direct contact with potential employers
– **Career Lounge:** Exclusive one-on-one conversations with compan
representatives
from seven interesting companies
– **Lectures und Workshops:** Helpful tips from experienced coaches
aspects of starting a career
– **CV-Checks:** Get your CV checked to obtain personalised feedback
– **JobWall:** Job advertisements from 16 companies for working students
internship ...

egistration start
31.05.2022
on StudOn

You can also find all the information here:

WiSo CAREER DAY

Previously on Introduction to Linked Data…

- We have learned about RDF and how to publish and access data
- The notion of RDF instance explains the meaning of blank nodes
- We have learned to write SPARQL queries against an RDF dataset

Lecture 3: Syntax of RDF Triples

Definition 6 (RDF Triple, RDF Graph)**.** *Let* $\mathcal{U}$ *be the set of URIs,* $\mathcal{B}$ *the set of blank nodes, and* $\mathcal{L}$ *the set of RDF literals. A tuple* $\langle s, p, o \rangle \in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ *is called an RDF triple, where* s *is the subject,* p *is the predicate and* o *is the object. A set of RDF triples is called RDF graph.*

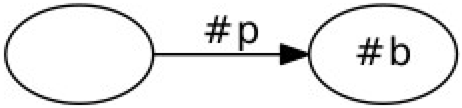
Lecture 3: RDF Instance Mapping

- To understand how blank nodes are handled, we start with the notion of an instance of a graph
- For that, we need the notion of RDF Instance Mapping

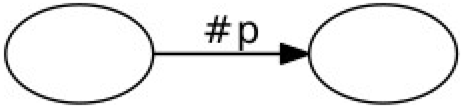
Definition 10 (RDF Instance Mapping, RDF Instance). *A partial function from blank nodes to RDF terms $\sigma : \mathcal{B} \mapsto \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ is called an RDF instance mapping. We write $\sigma(G)$ to denote an RDF graph obtained from graph G by replacing each blank node x in G with $\sigma(x)$. We call $\sigma(G)$ an instance of G.*

Lecture 3: Example RDF Instance Mapping

- Graph G0: _:a <#p> <#b> .



- Graph G1: _:bn1 <#p> _:bn2 .



- G0 is an instance of G1, assuming the RDF instance mapping :
  - (_:bn1) = _:a
  - (_:bn2) = <#b>

Lecture 3: Subgraph

- A subgraph of an RDF graph is a subset of the triples in the graph.

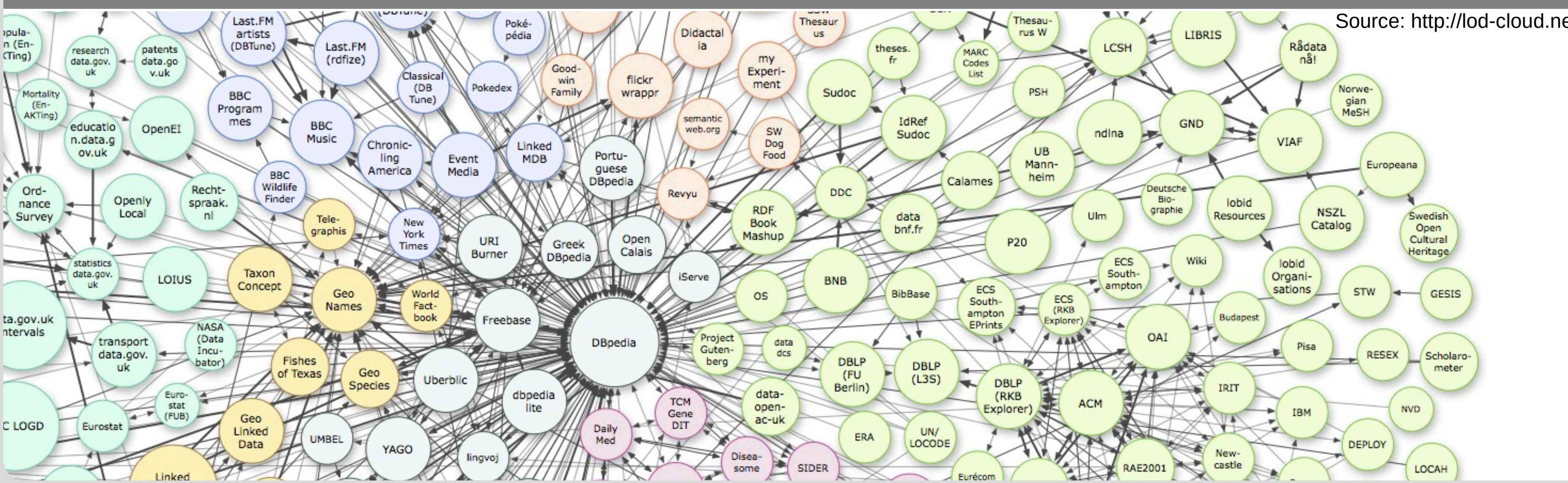
- Note that “subgraph” is to be taken literally. Datatyped literals are compared as they are, without taking into account that the same value might have different **lexical representations**. For example:

`{:a :b “01”^^xsd:integer .}` is **not** a subgraph of `{:a :b “1”^^xsd:integer .}`

C06 SPARQL Query Processing
What are correct solutions to queries?

Version 2021-05-12
Lecturer: Prof. Andreas Harth

CHAIR OF TECHNICAL INFORMATION SYSTEMS

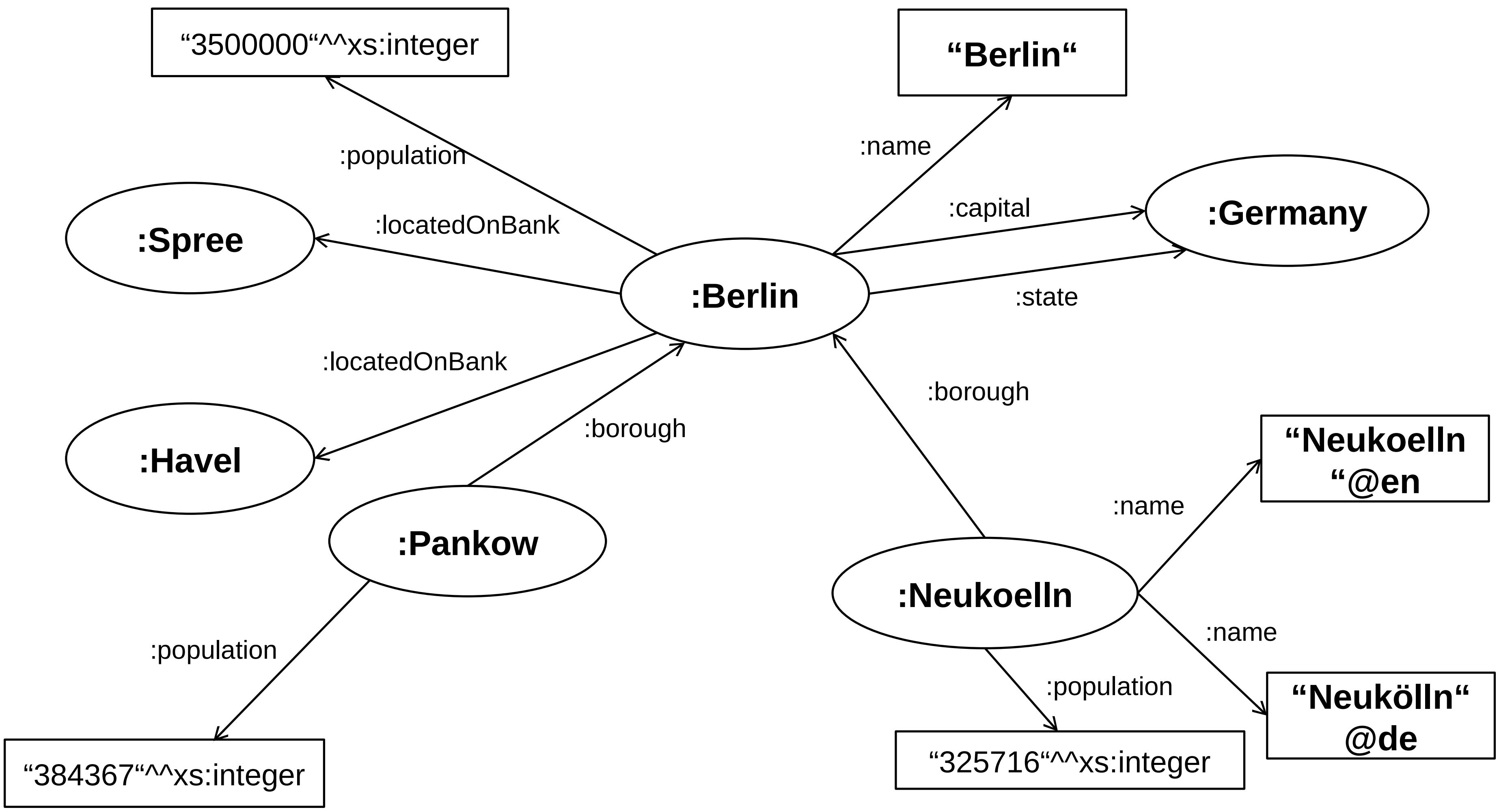


Source: http://lod-cloud.net

CC - Creative Commons Licensing

- This set of slides is part of the lecture „Semantic Web Technologies“ held at Karlsruhe Institute of Technology
- The content of the lecture was prepared by PD Dr. Andreas Harth based on his book „Introduction to Linked Data“
- The initial slides were prepared by Lars Heling
- Additional work on the slides by Maribel Acosta and Andreas Harth

Remember: The cities.ttl Graph



An Example Query

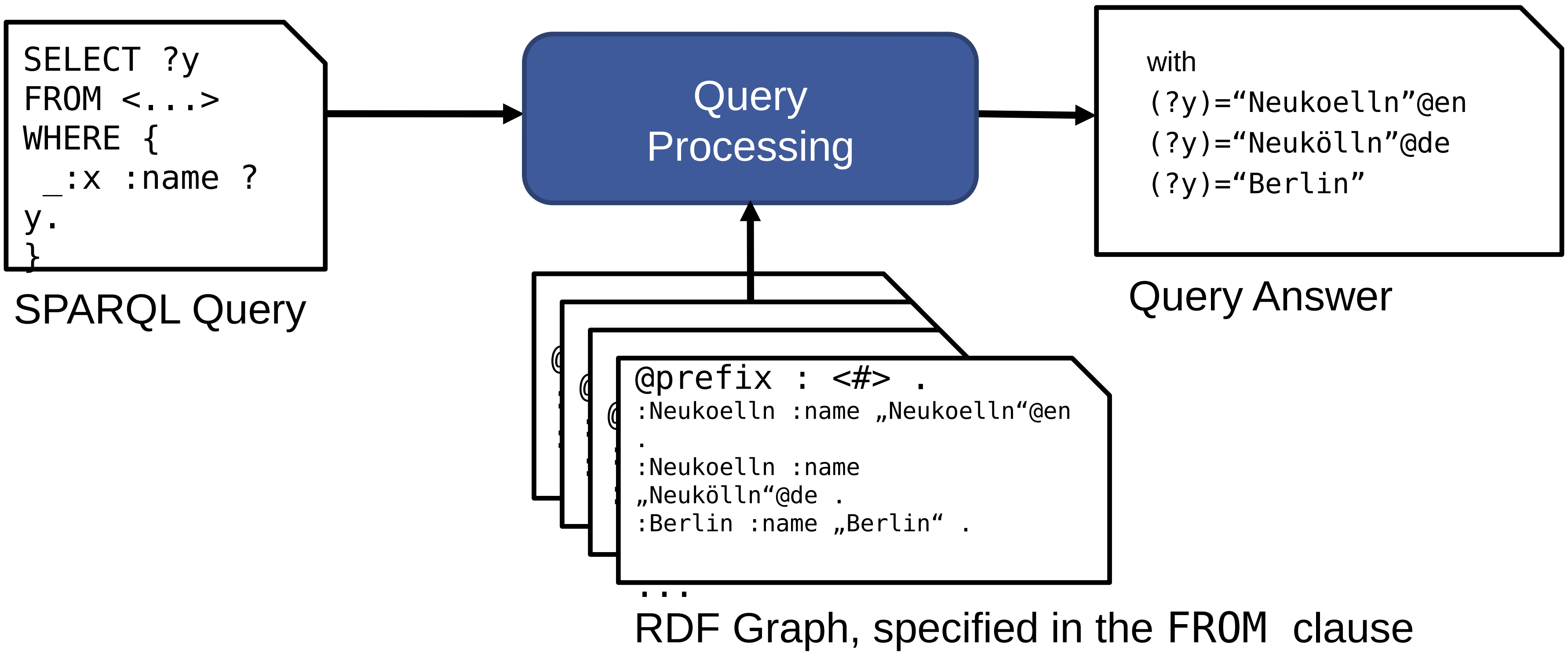
- Based on the document `http://example.org/cities.ttl`, the following query shall be executed:

```
PREFIX : <http://example.org/cities.ttl#>
SELECT ?y
FROM <http://example.org/cities.ttl>
WHERE {
      _:x :name ?y .
}
```

- What are the **results** of the query?

Processing a SPARQL Query

- Processing a SPARQL query against an RDF dataset yields the **solutions** (or answers) of the query.



How to Compute the Solutions of a Query?

Computing the solutions to a SPARQL query against a graph specified in the `FROM` clause involves the following steps:

- Translation of the SPARQL query into an abstract SPARQL query, including an algebra expression.
- Evaluation of the algebra expression for the graph patterns (conditions specified in the `WHERE` clause of the query).
- Accessing the RDF dataset and processing the query form.

Agenda

1. **Basic Graph Pattern Matching**
2. Translating Queries to Abstract Queries
3. Evaluating Graph Pattern Algebra Expressions
4. Processing Abstract Queries

From SPARQL Queries to Algebra Expressions

We start with:

- The algebra expression obtained for the conditions specified in the `WHERE` clause (graph patterns).

For the sake of simplicity, we ignore the prefix declarations.

```
PREFIX :
<http://example.org/cities.ttl#>
SELECT ?y
FROM <http://example.org/cities.ttl>
WHERE {
  ?borough :borough ?berlin .
  ?berlin  :name    "Berlin" .
}
```

Basic Graph Pattern

Basic Graph Patterns (BGPs)

- Translate the following graph pattern into an algebra expression.

```
{ ?borough :borough ?berlin .
  ?berlin  :name    "Berlin" .}
```

Solution: Let us consider

```
tp1 = ?borough :borough ?berlin .
tp2 = ?berlin  :name    "Berlin" .
```

Then, the resulting algebra expression is:

BGP(tp1 . tp2)

We deal with blank nodes in BGPs later; for now, assume blank nodes are variables.

Definition: Basic Graph Pattern

Definition 12 (Variable, Triple Pattern, Basic Graph Pattern). *Let $\mathcal{V}$ be an infinite set of variables; $\mathcal{V} \cap (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}) = \emptyset$. A triple $p \in (\mathcal{U} \cup \mathcal{B} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$ is called a triple pattern. A set of triple patterns is called basic graph pattern (BGP).*

- Example of triple patterns:

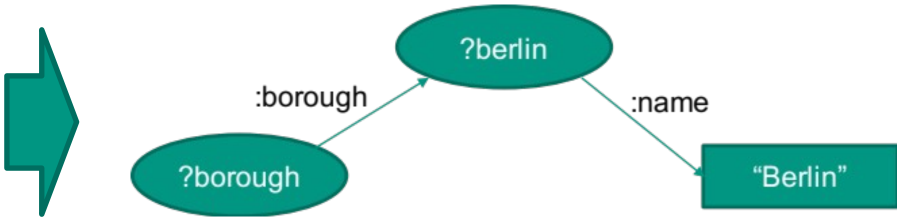
```
tp1 = ?borough :borough ?berlin .
tp2 = ?berlin  :name    "Berlin" .
```

- The entire BGP is:

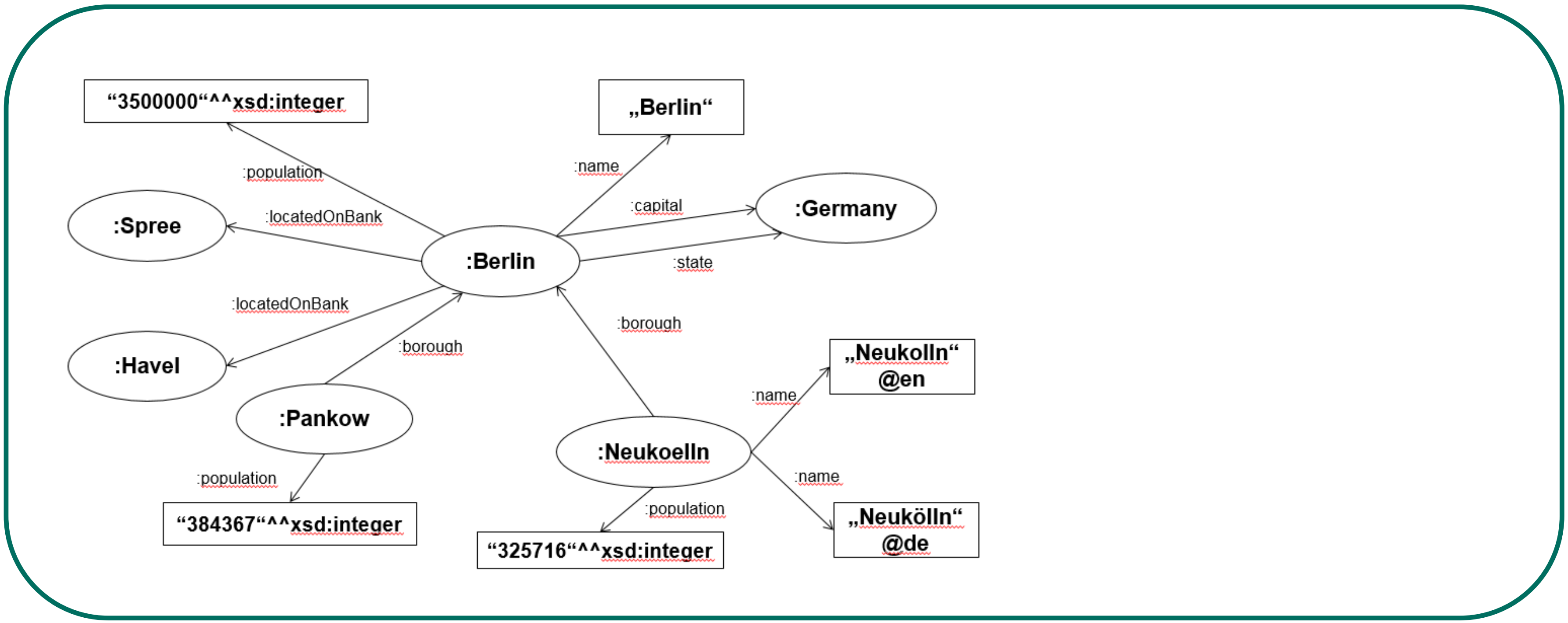
```
?borough :borough ?berlin .
?berlin  :name    "Berlin" .
```

Pattern Matching: Intuition (1)

```
WHERE {
    ?borough :borough ?berlin .
    ?berlin  :name    "Berlin" .
}
```

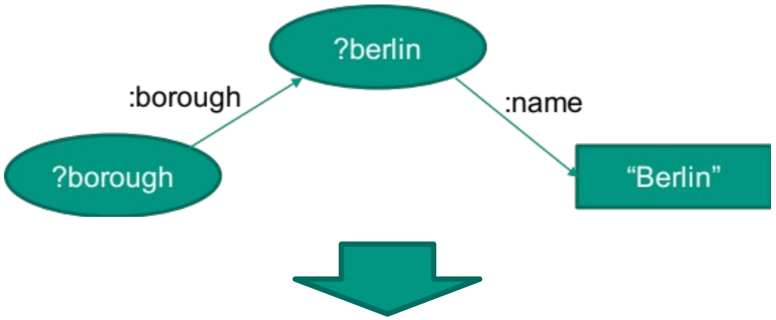


Default graph:

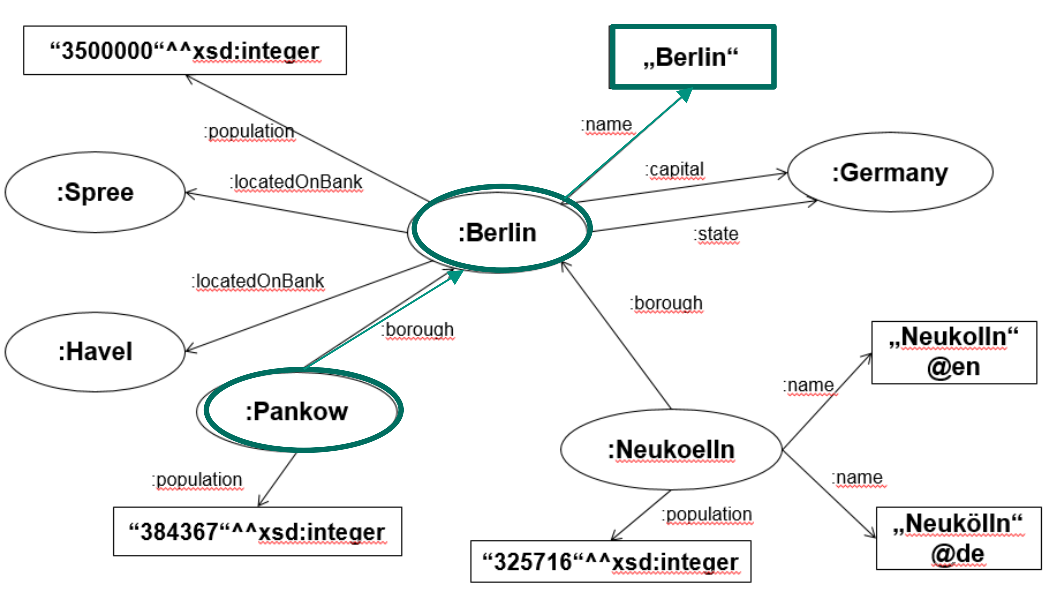


Pattern Matching: Intuition (2)

```
WHERE {
    ?borough :borough ?berlin .
    ?berlin  :name    "Berlin" .
}
```



Default graph:

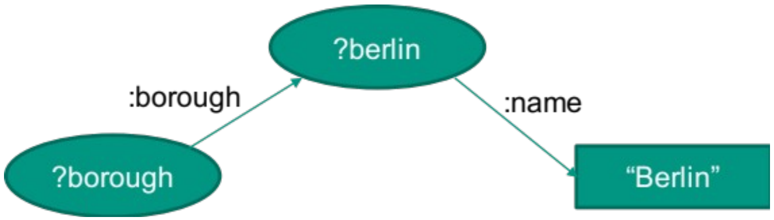


Solution 1:

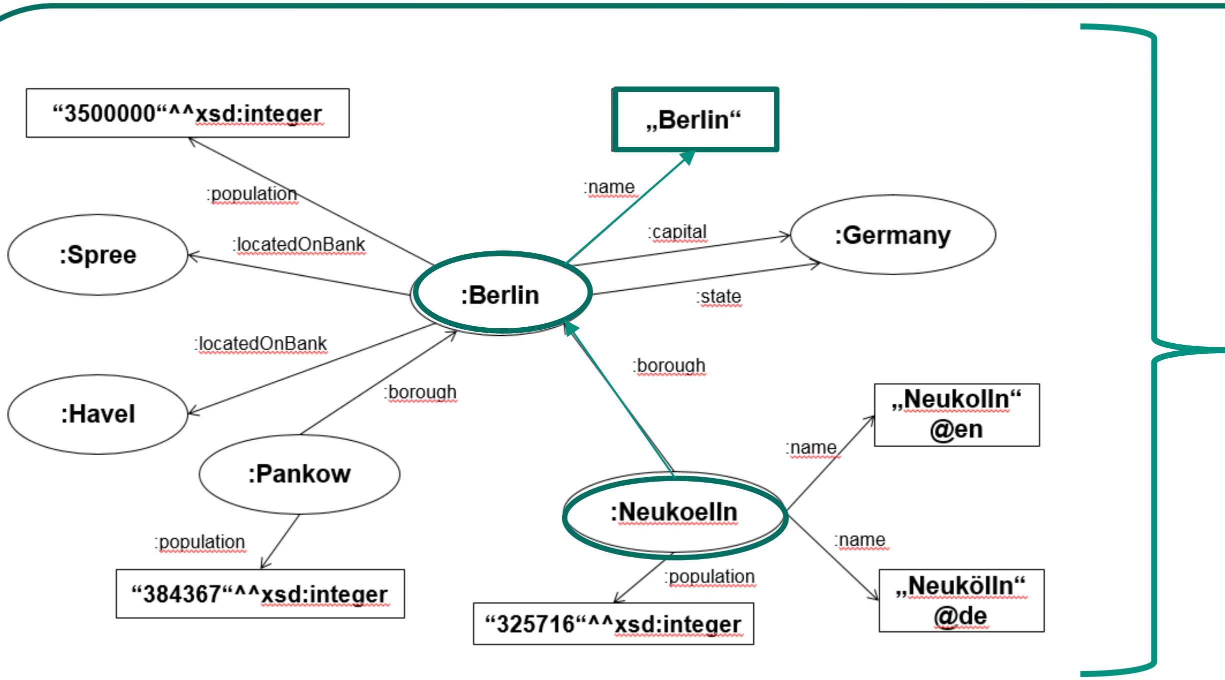
?borough = :Pankow
?berlin = :Berlin

Pattern Matching: Intuition (3)

```
WHERE {
    ?borough :borough ?berlin .
    ?berlin  :name    "Berlin" .
}
```



Default graph:

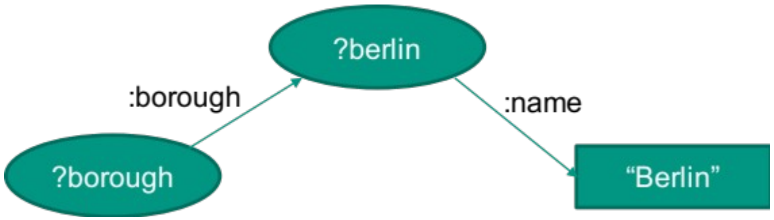


Solution 2:

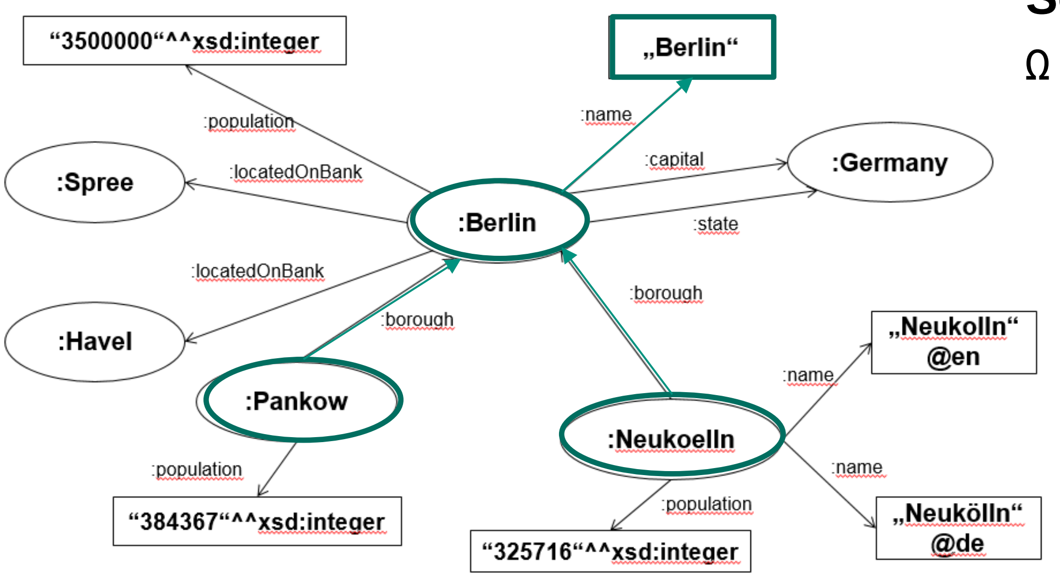
?borough = :Neukoelln
?berlin = :Berlin

Pattern Matching: Intuition (4)

```
WHERE {
    ?borough :borough ?berlin .
    ?berlin  :name    "Berlin" .
}
```



Default graph:



Solutions:

Ω = { μ1, μ2 } with

μ1(?borough) = :Pankow,
μ1(?berlin) = :Berlin
μ2(?borough) = :Neukoelln,
μ2(?berlin) = :Berlin

Definition: Solution Mapping

Definition 13 (Solution Mapping, Solution Sequence). *A partial function from variables to RDF terms $\mu : \mathcal{V} \mapsto \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ is called a solution mapping. A solution sequence Ω is a collection of solution mappings.*

- A collection is either:
  - a set (no duplicates allowed, elements not ordered)
  - a bag (duplicates allowed, elements not ordered) or
  - a list (duplicates allowed, elements are ordered)

- We assume that the solution sequence is a set, unless otherwise noted

Solution Mapping: Example

Definition 13 (Solution Mapping, Solution Sequence). *A partial function from variables to RDF terms* $\mu : \mathcal{V} \mapsto \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ *is called a solution mapping. A solution sequence* Ω *is a collection of solution mappings.*

- Consider the solution sequence Ω consisting of the following solution mappings μ1 and μ2:

```
μ1(?borough) = :Pankow, μ1(?berlin) = :Berlin
μ2(?borough) = :Neukoelln, μ2(?berlin) = :Berlin
```

- The domain of μ1 and μ2 are as follows:

```
dom(μ1) = {?borough, ?berlin}
dom(μ2) = {?borough, ?berlin}
```

Definition: Basic Graph Pattern Matching

Definition 14 (Basic Graph Pattern Matching). *Let P be a basic graph pattern and G be an RDF graph. A partial function μ is a solution of matching P on the graph G if:*

- *the domain of μ is the set of variables in P, and*
- *there exists a mapping σ of blank nodes in P to RDF terms ($\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$) in G, so that*
- *the graph $\mu(\sigma(P))$ is a subgraph of G.*

We write $\mu(P)$ to denote an RDF graph obtained from BGP P by replacing each variable x in P with $\mu(x)$.

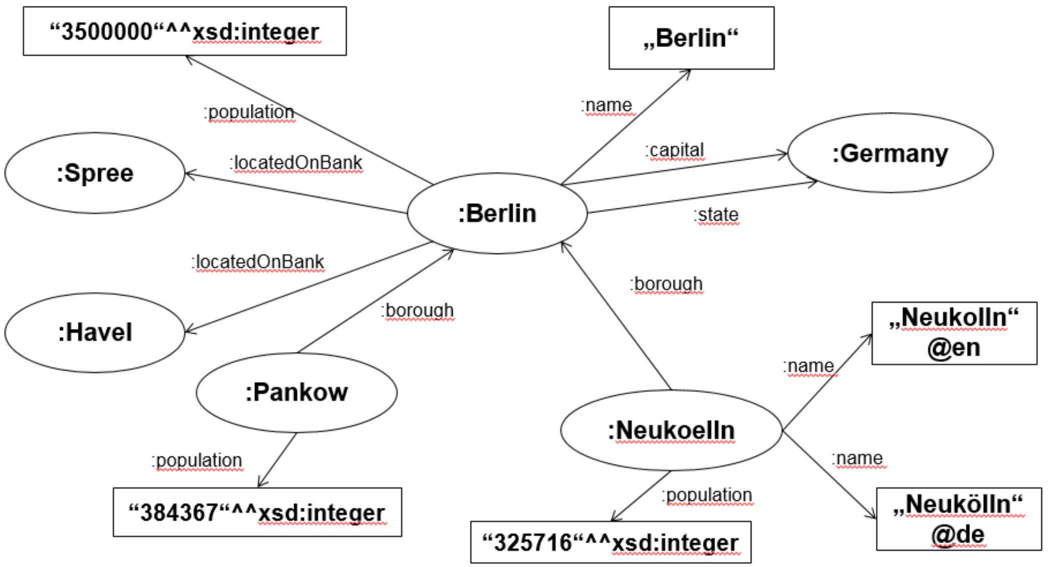
Example of BGP Matching (1)

- Consider BGP P = `?borough :borough ?berlin .`
  `?berlin  :name    "Berlin" .`
- Consider solution `µ1`
  `µ1(?borough) = :Pankow, µ1(?berlin) = :Berlin`

- Conditions:

1. the domain of `µ1` is the set of variables in P
2. no blank nodes, so empty mapping σ
3. the graph `µ(P)` is a subgraph of G

RDF Graph G:



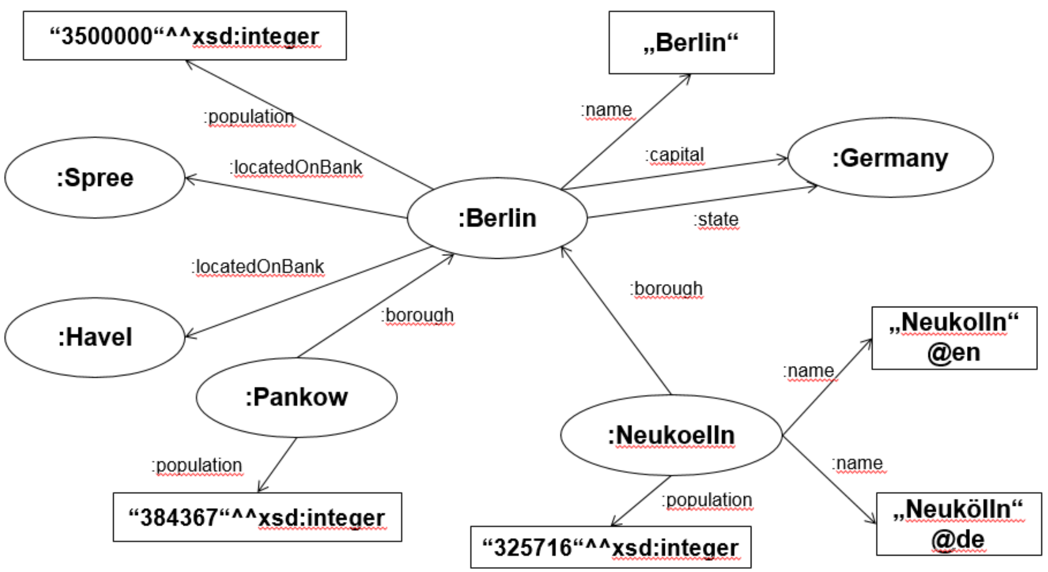
Example of BGP Matching (2)

- Consider P = `?borough :borough ?berlin .`
  `?berlin :name "Berlin" .`
- Consider solution µ2
  `µ2(?borough) = :Neukoelln, µ2(?berlin) = :Berlin`

- Conditions:

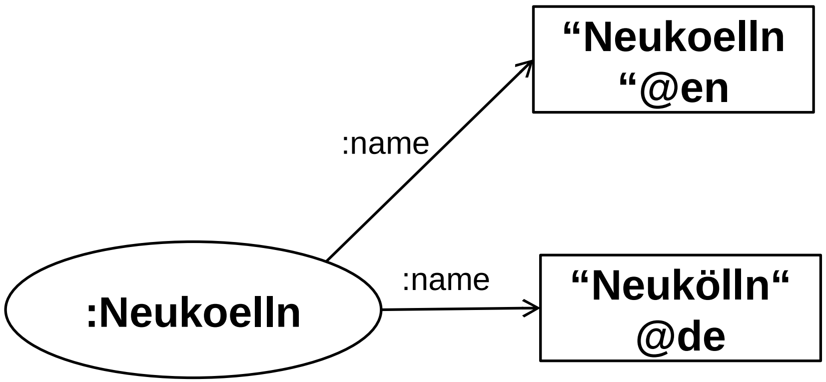
1. the domain of `µ1` is the set of variables in P
2. no blank nodes, so empty mapping σ
3. the graph `µ(P)` is a subgraph of G

RDF Graph G:



Think-Pair-Share

RDF graph G:



- Given BGP `P1 = _:x :name ?y .`
- Demonstrate that `P1` matches `G`.

Definition 14 (Basic Graph Pattern Matching)**.** *Let P be a basic graph pattern and G be an RDF graph. A partial function μ is a solution of matching P on the graph G if:*

- *the domain of μ is the set of variables in P, and*
- *there exists a mapping σ of blank nodes in P to RDF terms $(\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ in G, so that*
- *the graph $\mu(\sigma(P))$ is a subgraph of G.*

Agenda

1. Basic Graph Pattern Matching
2. **Translating Queries to Abstract Queries**
3. Evaluating Graph Pattern Algebra Expressions
4. Processing Abstract Queries

From SPARQL Queries to Abstract Queries

For processing entire SPARQL queries we require a more structured representation of a query string:

- The algebra expression obtained for the conditions specified in the `WHERE` clause (graph patterns) and the solution modifiers.
- The query form.
- The RDF dataset over which the query is evaluated.

For the sake of simplicity, we ignore the prefix declarations.

```
PREFIX :
<http://example.org/cities#>
SELECT ?y
FROM <http://example.org/cities.ttl>
WHERE {
  _:x :borough :Berlin .
  _:x :name ?y .
}
```

Query form

Description of RDF dataset

Algebra expression for graph patterns

Named Graphs and RDF Dataset

Definition 15 (Named Graph, RDF Dataset). *Let $\mathcal{G}$ be the set of RDF graphs and $\mathcal{U}$ be the set of URIs. A pair $\langle g, u \rangle \in \mathcal{G} \times \mathcal{U}$ is called a named graph. An RDF dataset consists of a (possibly empty) set of named graphs (with distinct names) and a default graph $g \in \mathcal{G}$ without a name.*

SPARQL Abstract Query

Definition 16 (SPARQL Abstract Query). *A SPARQL abstract query is a tuple $\langle M, D, QF \rangle$, where M is a SPARQL algebra expression, D is an RDF dataset and QF is a query form.*

- We start with algebra expressions of the graph pattern in the WHERE clause of a query.
- We consider query forms and RDF dataset descriptions later.

Example: Basic Graph Pattern (BGP)

- Translate the following graph pattern into an algebra expression.

```
{ _:b :borough :Berlin .
  _:b :name ?y .}
```

Solution: Let us consider

```
tp1 = _:b :borough :Berlin .
tp2 = _:b :name ?y .
```

Then, the resulting algebra expression is:

BGP(tp1 . tp2)

Example: JOIN

- Translate the following graph pattern into an algebra expression.

```
{{ ?x :borough :Berlin .
   ?x :name ?y .}
 { ?x rdf:type ?t .}}
```

Solution: Let us consider

```
tp1 = ?x :borough :Berlin .
tp2 = ?x :name ?y .
tp3 = ?x rdf:type ?t .
```

Then, the resulting algebra expression is:

JOIN(BGP(tp1 . tp2), BGP(tp3))

Example: UNION

- Translate the following graph pattern into an algebra expression.

```
{ ?x :borough :Berlin .
 { ?x :name ?y . }
 UNION
 { ?x :population ?z . } }
```

Solution: Let us consider

tp1 = `?x :borough :Berlin .`

tp2 = `?x :name ?y .`

tp3 = `?x :population ?z .`

Then, the resulting algebra expression is:

JOIN(BGP(tp1), UNION(BGP(tp2), BGP(tp3)))

Example: OPTIONAL

- Translate the following graph pattern into an algebra expression.

```
{ ?x :borough :Berlin .
 { ?x :name ?y . }
 OPTIONAL
 { ?x :population ?z . } }
```

Solution: Let us consider

```
tp1 = ?x :borough :Berlin .
tp2 = ?x :name ?y .
tp3 = ?x :population ?z .
```

Then, the resulting algebra expression is:

JOIN(BGP(tp1), LEFTJOIN(BGP(tp2), BGP(tp3), true))

Example: FILTER

- Translate the following graph pattern into an algebra expression.

```
{ ?x :borough :Berlin .
 { ?x :name ?y . }
 UNION
 { ?x :population ?z . FILTER (?z >
350000)} }
```

Solution: Let us consider

tp1 = ?x :borough :Berlin .

tp2 = ?x :name ?y .

tp3 = ?x :population ?z .

Then, the resulting algebra expression is:

JOIN(BGP(tp1), UNION(BGP(tp2), FILTER(?z>350000, BGP(tp3))))

Example: EXTEND

- Translate the following graph pattern into an algebra expression.

```
{ ?x :borough :Berlin .
  ?x :population ?z .
  BIND (?z/1000000 AS ?m) }
```

Solution: Let us consider

```
tp1 = ?x :borough :Berlin .
tp2 = ?x :population ?z .
```

Then, the resulting algebra expression is:

EXTEND(BGP(tp1 . tp2),?z/1000000, ?m*)*

Example: GRAPH

- Translate the following graph pattern into an algebra expression.

```
{ GRAPH ?g {
    ?x :borough :Berlin .
    ?x :population ?z . } }
```

Solution: Let us consider

```
tp1 = ?x :borough :Berlin .
tp2 = ?x :population ?z .
```

Then, the resulting algebra expression is:

GRAPH(?g, BGP(tp1 . tp2))

Selected Algebra Expressions for Graph Patterns

Symbol	Description
BGP(P)	Adjacent triple patterns *P* form a Basic Graph Pattern.
JOIN(M1, M2)	Conjunctive combination of algebra expression *M1, M2 (M1 and M2)*.
UNION(M1, M2)	Alternative combination of algebra expressions *M1, M2* (*M1 or M2*).
LEFTJOIN(M1, M2, expr)	Combination of algebra expression M1 with optional algebra expression M2 and filter condition *expr*.
FILTER(expr, M)	Use filter condition *expr* on algebra expression *M*.
EXTEND(M, expr, var)	Add extension expression *expr* to algebra expression *M* for .
GRAPH(g, M)	Apply the algebra expression *M* to graphs from

Translating Query Forms

Selected query forms.

Symbol	Description
SELECT(vars)	Return only variables from graph pattern solutions.
CONSTRUCT(pat)	Construct an RDF graph from the template pattern *pat* and the graph pattern solutions.
ASK	Return true if there exist at least one solution to the graph pattern. Return false otherwise.

Translating Dataset Clauses to Dataset Descriptions

- Translate the RDF dataset clause into a dataset description.

```
PREFIX : <http://example.org/cities.ttl#>
SELECT ?x ?p
FROM <http://example.org/cities.ttl>
FROM <http://example.org/munich.ttl>
WHERE {
    ?x :population ?p .
}
```

Translating Dataset Clauses to Dataset Descriptions

- Translate the RDF dataset clause into a dataset description.

```
PREFIX : <http://example.org/cities.ttl#>
SELECT ?x ?p
FROM <http://example.org/cities.ttl>
FROM <http://example.org/munich.ttl>
WHERE {
    ?x :population ?p .
}
```

- **Solution:** The RDF dataset description consists of the default graph with the RDF merge of `http://example.org/cities.ttl` and `http://example.org/munich.ttl`. The dataset has no named graphs.

Think-Pair-Share

- Translate the WHERE clause of the following SPARQL query into an SPARQL algebra expression:

```
PREFIX : <http://example.org/cities.ttl#>
SELECT ?x ?p
FROM <http://example.org/cities.ttl>
WHERE {
  ?x :population ?p .
  { ?x :borough :Berlin . }
  UNION
  { ?x :borough :Muenchen . }
  FILTER (?p > 100000)
}
```

Agenda

1. Basic Graph Pattern Matching
2. Translating Queries to Abstract Queries
3. **Evaluating Graph Pattern Algebra Expressions**
4. Processing Abstract Queries

Evaluating Algebra Expressions

- To evaluate the algebra expression, the *eval()* function is introduced
- The **input** of *eval()* is a **graph** and an **algebra expression**
- The **output** of *eval()* is a **solution sequence** Ω (a set of solution mappings)

The eval() Function

- Let D be an RDF dataset
- Let D(G) be an RDF dataset with the active graph G
- Let M be a SPARQL algebra expression
- Let Ω be a solution sequence

- The SPARQL recommendation defines *eval()* as:

  $$\Omega = \mathit{eval}(D(G), M)$$

- The *eval()* function is applied recursively over the algebra expression to evaluate each the expression (defined as algebra operators)

Recursive Procedure

- Let P be a basic graph pattern
- Let M, M1, M2 be SPARQL algebra expressions
- Let expr be a function expression
- The initially active graph is the default graph of D

- Now, apply *eval*() recursively:
  - *eval*(D(G), BGP(P)) = Ω (see definition Basic Graph Pattern Matching)
  - *eval*(D(G), JOIN(M1, M2)) = Join(*eval*(D(G), M1), *eval*(D(G), M2))
  - *eval*(D(G), UNION(M1, M2)) = Union(*eval*(D(G), M1), *eval*(D(G), M2))
  - *eval*(D(G), LEFTJOIN(M1, M2, expr)) = LeftJoin(*eval*(D(G), M1), *eval*(D(G), M2), expr)
  - *eval*(D(G), FILTER (expr, M)) = Filter(expr, *eval*(D(G), M), D(G))
  - *eval*(D(G), EXTEND(M, expr, var)) = Extend(*eval*(D(G), M), var, expr)
  - *eval*(D(G), GRAPH (g, M)) = change the active graph G

Compatibility of Solution Mappings

Definition (Solution Mapping Compatibility). *Two mappings* μ_1, μ_2 *are compatible, written* $\mu_1 \sim \mu_2$, *if* $\forall x \in dom(\mu_1) \cap dom(\mu_2) : \mu_1(x) = \mu_2(x)$.

- In other words: Two solution mappings are *compatible* if variables with the same name are bound to the same RDF term.
- **Example:**

Consider the following solution mappings µ1 and µ2:

```
µ1(?x) = :Neukoelln, µ1(?y) = "Neukoelln"@en
µ2(?x) = :Neukoelln
```

µ1 ~ µ2?

We have that `dom(µ1) = {?x, ?y}` and `dom(µ2) = {?x}`. The variables in common are: `dom(µ1)` ∩ `dom(µ2) = {?x}`

```
µ1(?x) = :Neukoelln and µ2(?x) = :Neukoelln
```

Therefore, µ1 ~ µ2.

Think-Pair-Share: Compatibility of Solution Mappings

μ1	μ2	μ1 ~ μ2? Why?
μ1(?x) = :Sonne, μ1(?y) = :Erde	μ2(?y) = :Erde, μ2(?z) = :Mond	
μ1(?x) = :Sonne, μ1(?y) = :Erde	μ2(?y) = :Mars, μ2(?z) = :Deimos	
μ1(?s1) = :Mars	μ2(?s2) = :Venus	
μ1(?s1) = :Mars, μ1(?s2) = :Erde	μ2(?s1) = :Mars, μ2(?s2) = :Erde, μ2(?s3) = :Sonne	

The Empty Solution Mapping $\mu_\emptyset$

- The empty solution mapping, denoted $\mu_\emptyset$, is the mapping with empty domain, i.e., $\mathtt{dom}(\mu_\emptyset) = \emptyset$.

- By definition of *compatibility*, we obtain that $\mu_\emptyset$ is compatible with all solution mappings.

- **Example:**

Consider the following solution mappings μ1 and $\mu_\emptyset$:

```
μ1(?x) = :Neukoelln, μ1(?y) = "Neukoelln"@en
μ∅ = { }
```

μ1 and $\mu_\emptyset$ are compatible, i.e., $\mu1 \sim \mu_\emptyset$

Definition of Operators

Operator	Definition
$Union(\Omega_1, \Omega_2)$	$\{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$
$Join(\Omega_1, \Omega_2)$	$\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1, \mu_2 \text{ are compatible}\}$
$Diff(\Omega_1, \Omega_2, F)$	$\{\mu \vert \mu \in \Omega_1$ such that for all $\mu' \in \Omega_2$, either μ and μ' are not compatible
	or μ and μ' are compatible and $F(merge(\mu, \mu'))$ is `false` $\}$
$LeftJoin(\Omega_1, \Omega_2, F)$	$Filter(F, Join(\Omega_1, \Omega_2)) \cup Diff(\Omega_1, \Omega_2, F)$
$Filter(expr, \Omega)$	$\{\mu \mid \mu \in \Omega$ and $expr(\mu)$ is an expression that evaluates to `true`$\}$
$Extend(\mu, var, expr)$	$\mu \cup \{(var, value) \mid var \notin dom(\mu) \text{ and } value = expr(\mu)\}$
	μ if $var \notin dom(\mu)$ and $expr(\mu)$ is an error
	undefined when $var \in dom(\mu)$
$Extend(\Omega, var, expr)$	$\{Extend(\mu, var, expr) \mid \mu \in \Omega\}$

Example: *Join* Operator

- Consider the following sets of solution mappings:

```
Ω1 = { µ1, µ2, µ3 }    with
  µ1(?x) = :Pankow,    µ1(?y) = :Berlin
  µ2(?x) = :Neukoelln, µ2(?y) = :Berlin
  µ3(?x) = :Barcelona, µ3(?y) = :Catalonia

Ω2 = { µ4 } with
  µ4(?y) = :Berlin, µ4(?z) = :Germany
```

- Compute `Join(Ω1, Ω2)`

Compatible mappings!

$$Join(\Omega_l, \Omega_r) \quad := \quad \{\mu_l \cup \mu_r \mid \mu_l \in \Omega_l, \mu_r \in \Omega_r : \mu_l \sim \mu_r\}$$

```
Join(Ω1, Ω2) = {
    µ14(?x)=:Pankow, µ14(?y)=:Berlin, µ14(?z)=:Germany
   µ24(?x)=:Neukoelln, µ24(?y)=:Berlin, µ24(?
z)=:Germany }
```

Filter Operator: expr(μ)

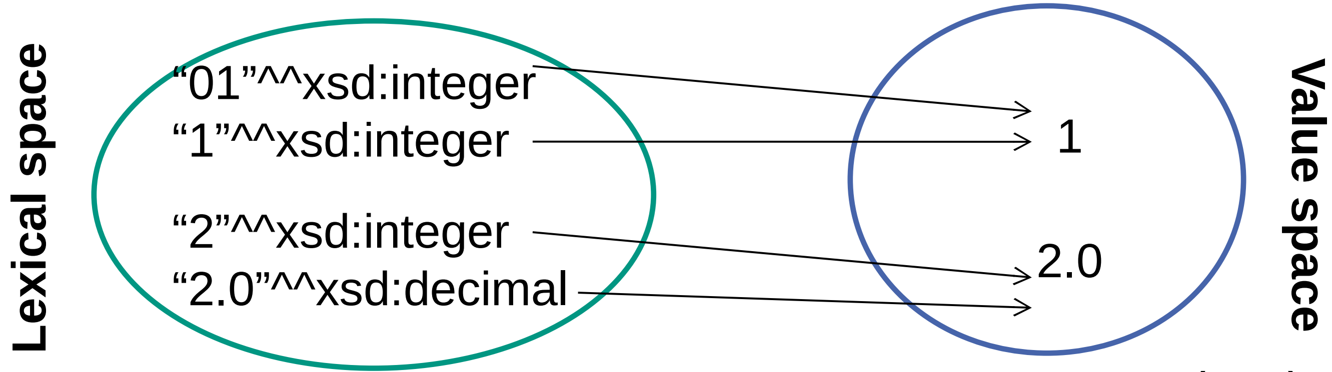
- `expr(μ)` evaluates the instantiation of variables in `expr` with the values of `μ.` Returns `TRUE`, `FALSE`, or `error.`

- For example:

  `μ1(?b) = "01"^^xsd:integer` and `F = (?b = 1)`, then

  `F(μ1)` evaluates `(“01”^^xsd:integer = 1)`

- Unlike pattern matching, `expr(μ)` considers the value space of RDF literals (and not the lexical space).



- Taking into account the value space, we conclude that `F(μ1)` from the example evaluates to `TRUE.`

Type Conversion

* SPARQL does automatic type casting between xsd:float and xsd:decimal.

- Some datatypes (e.g., xsd:float and xsd:decimal) share a lexical space (although they have a different value space!)
- To be able to compare literals of different (incompatible) datatypes, we can use casting.
- For example, consider `foo.ttl`:

```
@prefix : <foo#> .
:s :p "9.90" ; :q 9.90 .
```

- Now, we can compare the two literals as follows:

```
PREFIX : <foo#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
ASK FROM <foo.ttl>
WHERE { :s :p ?o1 .   :s :q ?o2 .
        FILTER (xsd:decimal(?o1) = ?o2)
}
```

Example: *Filter* Operator

- Consider the following set of solution mappings:

  `Ω1 = { µ1, µ2 }` with

  `µ1(?a) = :Pankow,    µ1(?b) = "384367"^^xsd:integer`

  `µ2(?a) = :Neukoelln, µ1(?b) = "325716"^^xsd:integer`

- Compute `Filter((?b > 350000), Ω1)`

$$Filter(expr, \Omega) \quad := \quad \{\mu |\ \mu \in \Omega \ \ and\ expr(\mu)\ evaluates\ to\ true\}$$

`For µ1: ("384367"^^xsd:integer > 350000) TRUE`

`For µ2: ("325716"^^xsd:integer > 350000) FALSE`

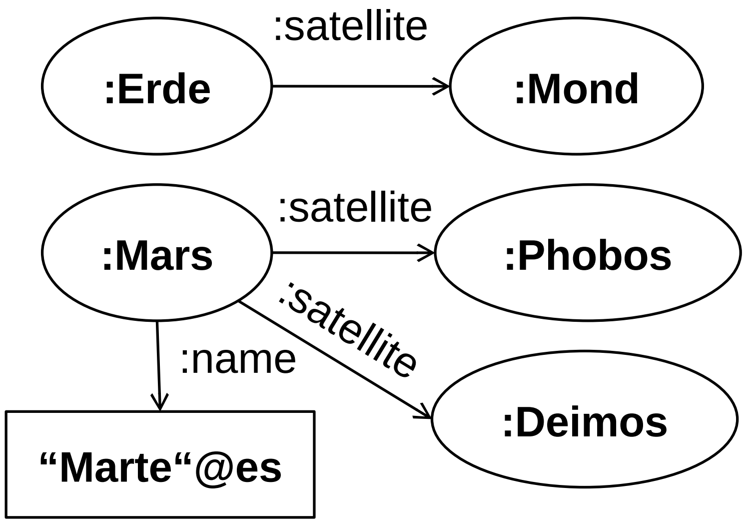
Only `µ1` is part of the solution:

`Filter((?b > 350000), Ω1) = { µ1 }` with

`µ1(?a) = :Pankow, µ1(?b) = "384367"^^xsd:integer`

Example: Evaluating a JOIN Expression (1)

Given is D(G):



Evaluate the following SPARQL algebra expression against D(G):

```
JOIN(BGP(P1), BGP(P2))
With
P1 = ?p :satellite ?s1 .
P2 = ?p :satellite ?s2 .
```

Solution: We have to compute **`Ω=eval(D(G), JOIN(BGP(P1), BGP(P2)))`**.

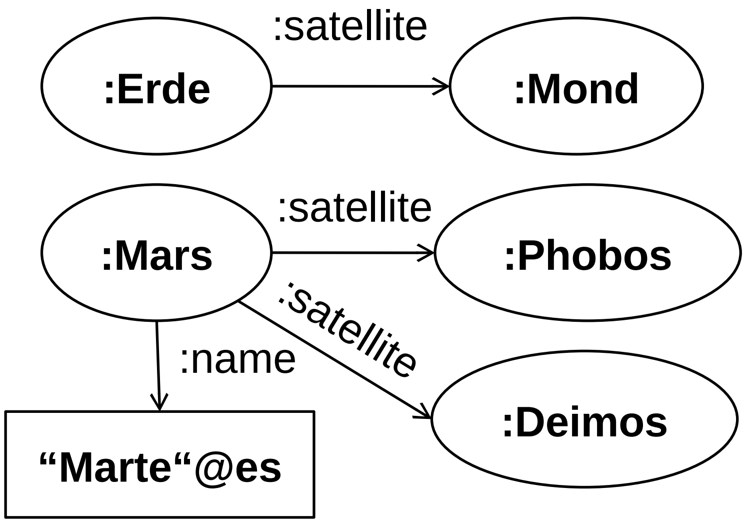
By definition of *eval* for `JOIN` expressions we obtain that:

$$eval(D(G), JOIN(P_1, P_2)) := Join(eval(D(G), P_1), eval(D(G), P_2))$$

We have to compute `Ω1=eval(D(G), P1)` and `Ω2=eval(D(G), P2)` and then `Join` the results `Ω1` and `Ω2`.

Example: Evaluating a JOIN Expression (2)

Given is D(G):



Evaluate the following SPARQL algebra expression against D(G):

```
    JOIN(BGP(P1), BGP(P2))
With
    P1 = ?p :satellite ?s1 .
    P2 = ?p :satellite ?s2 .
```

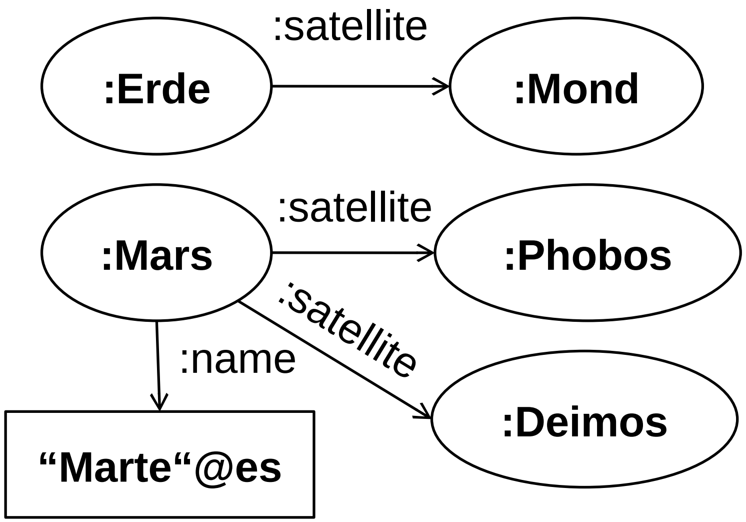
Solution (Cont): Let’s compute **`Ω1 = eval(D(G), P1)`**.

P1 is a triple pattern. The definition of *eval* for triple patterns says that we have to apply pattern matching: `Ω1 = { µ1, µ2, µ3 }` with

```
µ1(?p) = :Erde, µ1(?s1) = :Mond
µ2(?p) = :Mars, µ2(?s1) = :Phobos
µ3(?p) = :Mars, µ3(?s1) = :Deimos
```

Example: Evaluating a JOIN Expression (3)

Given is D(G):



Evaluate the following SPARQL algebra expression against D(G):

```
    P1 JOIN P2
With
    P1 = ?p :satellite ?s1 .
    P2 = ?p :satellite ?s2 .
```

Solution (Cont): Now we compute **`Ω2 = eval(D(G), P2)`**.

P2 is a triple pattern. The definition of *eval* for triple patterns says that we have to apply pattern matching: Ω2 `= { µ4, µ5, µ6 }` with

```
µ4(?p) = :Erde, µ4(?s2) = :Mond
µ5(?p) = :Mars, µ5(?s2) = :Phobos
µ6(?p) = :Mars, µ6(?s2) = :Deimos
```

Example: Evaluating a JOIN Expression (4)

Solution (Cont): Now we compute **`Ω=Join(Ω1, Ω2)`**, with:

```
Ω1 = { μ1, μ2, μ3 }    with
  μ1(?p) = :Erde, μ1(?s1) = :Mond
  μ2(?p) = :Mars, μ2(?s1) = :Phobos
  μ3(?p) = :Mars, μ3(?s1) = :Deimos
```

```
Ω2 = { μ4, μ5, μ6 }    with
  μ4(?p) = :Erde, μ4(?s2) = :Mond
  μ5(?p) = :Mars, μ5(?s2) = :Phobos
  μ6(?p) = :Mars, μ6(?s2) = :Deimos
```

Only the compatible mappings are part of Ω:

```
Ω = { … }
```

Example: Evaluating a JOIN Expression (4)

Solution (Cont): Now we compute **`Ω=Join(Ω1, Ω2)`**, with:

```
Ω1 = { μ1, μ2, μ3 }    with
  μ1(?p) = :Erde, μ1(?s1) = :Mond
  μ2(?p) = :Mars, μ2(?s1) = :Phobos
  μ3(?p) = :Mars, μ3(?s1) = :Deimos
```

Compatible!

```
Ω2 = { μ4, μ5, μ6 }    with
  μ4(?p) = :Erde, μ4(?s2) = :Mond
  μ5(?p) = :Mars, μ5(?s2) = :Phobos
  μ6(?p) = :Mars, μ6(?s2) = :Deimos
```

Only the compatible mappings are part of Ω:

```
Ω = { μ14, … }    with
  μ14(?p) = :Erde, μ14(?s1) = :Mond,   μ14(?s2) = :Mond
```

Example: Evaluating a JOIN Expression (4)

Solution (Cont): Now we compute **`Ω=Join`**(**`Ω1, Ω2)`**, with:

```
Ω1 = { μ1, μ2, μ3 }    with
  μ1(?p) = :Erde, μ1(?s1) = :Mond
  μ2(?p) = :Mars, μ2(?s1) = :Phobos
  μ3(?p) = :Mars, μ3(?s1) = :Deimos

Ω2 = { μ4, μ5, μ6 }    with
  μ4(?p) = :Erde, μ4(?s2) = :Mond
  μ5(?p) = :Mars, μ5(?s2) = :Phobos
  μ6(?p) = :Mars, μ6(?s2) = :Deimos
```

Compatible!

Only the compatible mappings are part of Ω:

```
Ω = { μ14, μ25, … }    with
  μ14(?p) = :Erde, μ14(?s1) = :Mond,   μ14(?s2) = :Mond
  μ25(?p) = :Mars, μ25(?s1) = :Phobos, μ25(?s2) = :Phobos
```

Example: Evaluating a JOIN Expression (4)

Solution (Cont): Now we compute **`Ω=Join(Ω1, Ω2)`**, with:

`Ω1 = { µ1, µ2, µ3 }` with

```
µ1(?p) = :Erde, µ1(?s1) = :Mond
µ2(?p) = :Mars, µ2(?s1) = :Phobos
µ3(?p) = :Mars, µ3(?s1) = :Deimos
```

`Ω2 = { µ4, µ5, µ6 }` with

```
µ4(?p) = :Erde, µ4(?s2) = :Mond
µ5(?p) = :Mars, µ5(?s2) = :Phobos
µ6(?p) = :Mars, µ6(?s2) = :Deimos
```

Compatible!

Only the compatible mappings are part of Ω:

`Ω = { µ14, µ25, µ26, … }` with

```
µ14(?p) = :Erde, µ14(?s1) = :Mond,   µ14(?s2) = :Mond
µ25(?p) = :Mars, µ25(?s1) = :Phobos, µ25(?s2) = :Phobos
µ26(?p) = :Mars, µ26(?s1) = :Phobos, µ26(?s2) = :Deimos
```

Example: Evaluating a JOIN Expression (4)

Solution (Cont): Now we compute **`Ω=Join(Ω1, Ω2)`**, with:

```
Ω1 = { μ1, μ2, μ3 }    with
  μ1(?p) = :Erde, μ1(?s1) = :Mond
  μ2(?p) = :Mars, μ2(?s1) = :Phobos
  μ3(?p) = :Mars, μ3(?s1) = :Deimos
```

```
Ω2 = { μ4, μ5, μ6 }    with
  μ4(?p) = :Erde, μ4(?s2) = :Mond
  μ5(?p) = :Mars, μ5(?s2) = :Phobos
  μ6(?p) = :Mars, μ6(?s2) = :Deimos
```

Compatible!

Only the compatible mappings are part of Ω:

```
Ω = { μ14, μ25, μ26, μ35, … }    with
  μ14(?p) = :Erde, μ14(?s1) = :Mond,   μ14(?s2) = :Mond
  μ25(?p) = :Mars, μ25(?s1) = :Phobos, μ25(?s2) = :Phobos
  μ26(?p) = :Mars, μ26(?s1) = :Phobos, μ26(?s2) = :Deimos
  μ35(?p) = :Mars, μ35(?s1) = :Deimos, μ35(?s2) = :Phobos
```

Example: Evaluating a JOIN Expression (4)

Solution (Cont): Now we compute **`Ω=Join(Ω1, Ω2)`**, with:

```
Ω1 = { µ1, µ2, µ3 }    with
  µ1(?p) = :Erde, µ1(?s1) = :Mond
  µ2(?p) = :Mars, µ2(?s1) = :Phobos
  µ3(?p) = :Mars, µ3(?s1) = :Deimos

Ω2 = { µ4, µ5, µ6 }    with
  µ4(?p) = :Erde, µ4(?s2) = :Mond
  µ5(?p) = :Mars, µ5(?s2) = :Phobos
  µ6(?p) = :Mars, µ6(?s2) = :Deimos
```

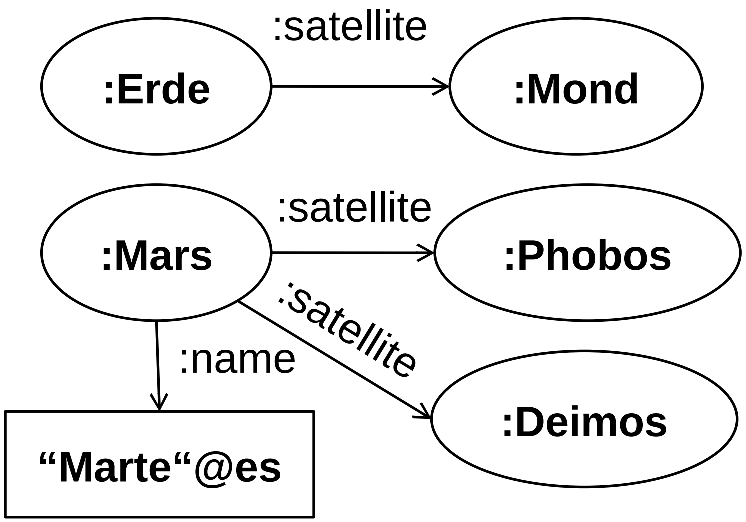
Compatible!

Only the compatible mappings are part of Ω:

```
Ω = { µ14, µ25, µ26, µ35, µ36 }    with
  µ14(?p) = :Erde, µ14(?s1) = :Mond,   µ14(?s2) = :Mond
  µ25(?p) = :Mars, µ25(?s1) = :Phobos, µ25(?s2) = :Phobos
  µ26(?p) = :Mars, µ26(?s1) = :Phobos, µ26(?s2) = :Deimos
  µ35(?p) = :Mars, µ35(?s1) = :Deimos, µ35(?s2) = :Phobos
  µ36(?p) = :Mars, µ36(?s1) = :Deimos, µ36(?s2) = :Deimos
```

Example: Evaluating a JOIN Expression (5)

Given is D(G):



Evaluate the following SPARQL algebra expression against D(G):

```
JOIN(BGP(P1), BGP(P2))
```

With

```
P1 = ?p :satellite ?s1 .
P2 = ?p :satellite ?s2 .
```

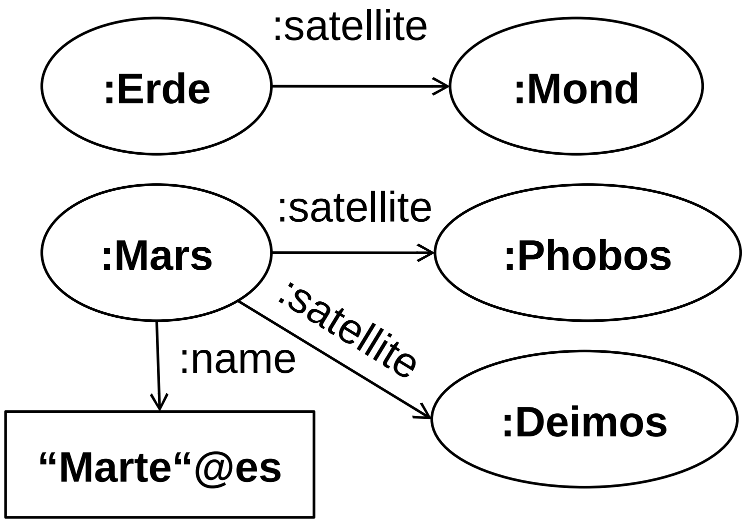
Solution (Cont): The result of evaluating `JOIN(BGP(P1), BGP(P2))` against `D(G)` is:

```
Ω = { µ14, µ25, µ26, µ35, µ36 }
```
with

```
µ14(?p) = :Erde, µ14(?s1) = :Mond,   µ14(?s2) = :Mond
µ25(?p) = :Mars, µ25(?s1) = :Phobos, µ25(?s2) = :Phobos
µ26(?p) = :Mars, µ26(?s1) = :Phobos, µ26(?s2) = :Deimos
µ35(?p) = :Mars, µ35(?s1) = :Deimos, µ35(?s2) = :Phobos
µ36(?p) = :Mars, µ36(?s1) = :Deimos, µ36(?s2) = :Deimos
```

Example: Evaluating a FILTER Expression (1)

Given is D(G):



Evaluate the following SPARQL algebra expression against D(G):

```
FILTER(E,
      JOIN(BGP(P1), BGP(P2)))
```

With

```
P1 = ?p :satellite ?s1 .
P2 = ?p :satellite ?s2 .
E  = (?s1!=?s2)
```

Solution: We have to compute **`Ω=eval(D(G),`** `FILTER(E,JOIN(BGP(P1), BGP(P2)))`**`).`**

By definition of *eval* for `FILTER` expressions, we obtain that:

$$eval(D(G), FILTER(expr, P_1)) \quad := \quad Filter(expr, eval(D(G), P_1))$$

We have to compute `Ω1=eval(D(G), JOIN(BGP(P1), BGP(P2)))` and then apply `Filter` of the expression E over the results `Ω1`.

Example: Evaluating a FILTER Expression (2)

Solution (Cont): We computed **`Ω=eval(D(G),`** `JOIN(BGP(P1), BGP(P2))`**`)`**, in the previous example:

`Ω = { µ14, µ25, µ26, µ35, µ36 }` with

```
µ14(?p) = :Erde, µ14(?s1) = :Mond,   µ14(?s2) = :Mond
µ25(?p) = :Mars, µ25(?s1) = :Phobos, µ25(?s2) = :Phobos
µ26(?p) = :Mars, µ26(?s1) = :Phobos, µ26(?s2) = :Deimos
µ35(?p) = :Mars, µ35(?s1) = :Deimos, µ35(?s2) = :Phobos
µ36(?p) = :Mars, µ36(?s1) = :Deimos, µ36(?s2) = :Deimos
```

Now we have to compute **`Ω=Filter(E, Ω)`**, with **`E= (?s1!=?s2).`** For each solution mapping µ in Ω, we check if **`?s1!=?s2`**. The solution mappings µ for which `E(µ)` evaluates to `TRUE` are part of `Ω.`

Example: Evaluating a FILTER Expression (3)

Solution (Cont): We computed **`Ω=eval(D(G),`** `JOIN(BGP(P1), BGP(P2))`**`)`**, in the previous example:

`Ω = { μ14, μ25, μ26, μ35, μ36 }` with

```
μ14(?p) = :Erde, μ14(?s1) = :Mond,   μ14(?s2) = :Mond   FALSE
μ25(?p) = :Mars, μ25(?s1) = :Phobos, μ25(?s2) = :Phobos FALSE
μ26(?p) = :Mars, μ26(?s1) = :Phobos, μ26(?s2) = :Deimos TRUE
μ35(?p) = :Mars, μ35(?s1) = :Deimos, μ35(?s2) = :Phobos TRUE
μ36(?p) = :Mars, μ36(?s1) = :Deimos, μ36(?s2) = :Deimos FALSE
```

Now we have to compute **`Ω=Filter(E, Ω)`**, with **`E= (?s1!=?s2).`** For each solution mapping μ in Ω, we check if **`?s1!=?s2`**. The solution mappings μ for which E(μ) evaluates to TRUE are part of `Ω.`

`Ω = {       μ26, μ35 }` with

```
μ26(?p) = :Mars, μ26(?s1) = :Phobos, μ26(?s2) = :Deimos
μ35(?p) = :Mars, μ35(?s1) = :Deimos, μ35(?s2) = :Phobos
```

Agenda

1. Basic Graph Pattern Matching
2. Translating Queries to Abstract Queries
3. Evaluating Graph Pattern Algebra Expressions
4. **Processing Abstract Queries**

SPARQL Query Processing: End-to-End

- Evaluating algebra expressions via *eval()* is at the core of the definitions around SPARQL query processing.
- However, getting the solution sequence Ω is only a part of query processing.

- Other features need consideration, for example the implementation of BGP matching (the definition 14 of BGP matching was not an operational one).
- Handling of RDF datasets is another feature, also handling of GRAPH.
- We also have not considered solution modifiers (LIMIT, ORDER BY…)
- Finally, the final results of the have to be serialised, e.g., as TSV for SELECT queries.

User Agent Algorithm for SELECT Query Processing

input: SPARQL SELECT query q
output: solution sequence with result to q
set $default$:= URIs in `FROM` clause of q
set $named$:= URIs in `FROM NAMED` clause of q
RDF dataset $D := \mathsf{access}(default, named)$
SPARQL algebra expression $P := \mathsf{translate}(q)$
solution sequence $\Omega := \mathsf{eval}(D, P)$
$\Omega :=$ project out selected variables from Ω
return Ω

Recap (1)

- **Pattern Matching:**
  - Pattern matching: How to match triple patterns against an RDF graph. Considers the lexical space of RDF literals (exact match)
  - The SPARQL algebra:
    - Solution mappings (μ): from variables to RDF terms
    - RDF instance mappings (σ): from blank nodes to RDF terms
    - Solution sequences (Ω): set of solution mappings
    - Algebra operators to combine sets of solution mappings
- Translating SPARQL queries into **algebra expressions**:
  - Algebra expressions of SPARQL graph patterns: to translate the conditions specified in the `WHERE` clause to algebra expressions
  - Handling of SPARQL query forms: to translate the query form (`SELECT, ASK, CONSTRUCT`)

Recap (2)

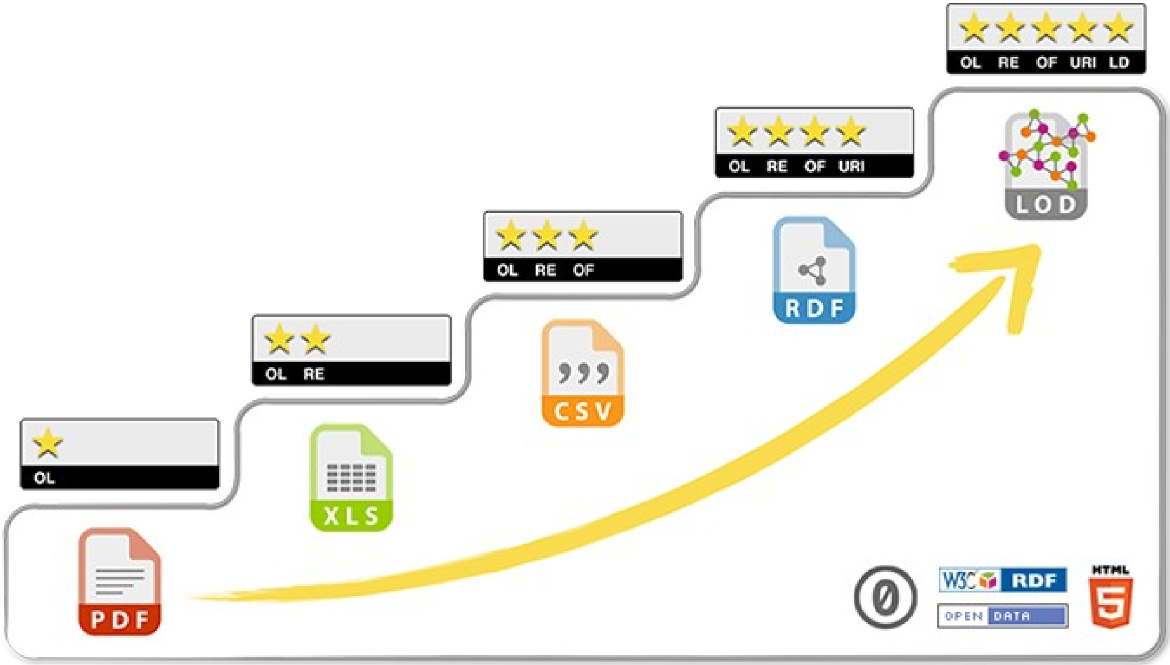
- **Evaluating Algebra Expressions of SPARQL Graph Patterns**
  - The *eval()* function for algebra expressions of SPARQL graph patterns
  - *eval()* for *FILTER* expressions considers the value space of RDF literals
- **Processing Abstract Queries**
  - Practical considerations for user agents that process queries
- We left out a lot of SPARQL features (intentionally!)

Learning Goals

- G 5.1 Explain the formal definition of basic graph pattern matching show how to match basic graph patterns to graphs.
- G 5.2 Translate a given SPARQL WHERE clause, including UNION, OPTIONAL, FILTER and BIND AS clauses, to a SPARQL algebra expression.
- G 5.3 Given a query, explain the handling of the RDF dataset with FROM and FROM NAMED clauses in conjunction with GRAPH.
- G 5.4 Evaluate a SPARQL algebra expression on a given RDF dataset and specify the solutions of the entire algebra expression and also of partial expressions.
- G 5.5 Generate the final results to a SPARQL abstract query, taking into account the solution sequence of the graph pattern algebra expression and the query form.

Outlook – Chapter 6

- Chapter 6 is concerned with building data integration systems.
- Many organisations (EU, countries, cities) provide data under open licenses.
- We consider what architectures are suitable for accessing Linked Data published under open licenses on the web.



https://5stardata.info/en/