SPARQL Query Language for RDF

Bernd Neumayr, Johannes Kepler University Linz

with contributions from Dieter Steiner

text taken from: **http://www.w3.org/TR/sparql11-query/**

SPARQL 1.1 Overview

- SPARQL 1.1 is a set of specifications that provide languages and protocols to **query and manipulate RDF** graph content **on the Web or in an RDF store**. The standard comprises the following specifications:
  - SPARQL 1.1 **Query Language** - A query language for RDF.
  - SPARQL 1.1 **Query Results JSON Format** and **SPARQL 1.1 Query Results CSV and TSV** Formats - Apart from the standard SPARQL Query Results XML Format [SPARQL-XML-Result], SPARQL 1.1 now allows three alternative popular formats to exchange answers to SPARQL queries, namely JSON, CSV (comma separated values) and TSV (tab separated values) which are described in these two documents.
  - SPARQL 1.1 **Federated Query** - A specification defining an extension of the SPARQL 1.1 Query Language for executing queries distributed over different SPARQL endpoints.
  - SPARQL 1.1 **Entailment Regimes** - A specification defining the semantics of SPARQL queries under entailment regimes such as RDF Schema, OWL, or RIF.
  - SPARQL 1.1 **Update Language** - An update language for RDF graphs.
  - SPARQL 1.1 **Protocol for RDF** - A protocol defining means for conveying arbitrary SPARQL queries and update requests to a SPARQL service.
  - SPARQL 1.1 **Service Description** - A specification defining a method for discovering and a vocabulary for describing SPARQL services.
  - SPARQL 1.1 **Graph Store HTTP Protocol** - As opposed to the full SPARQL protocol, this specification defines minimal means for managing RDF graph content directly via common HTTP operations.
  - SPARQL 1.1 **Test Cases** - A suite of tests, helpful for understanding corner cases in the specification and assessing whether a system is SPARQL 1.1 conformant

see: http://www.w3.org/TR/sparql11-overview/

Agenda

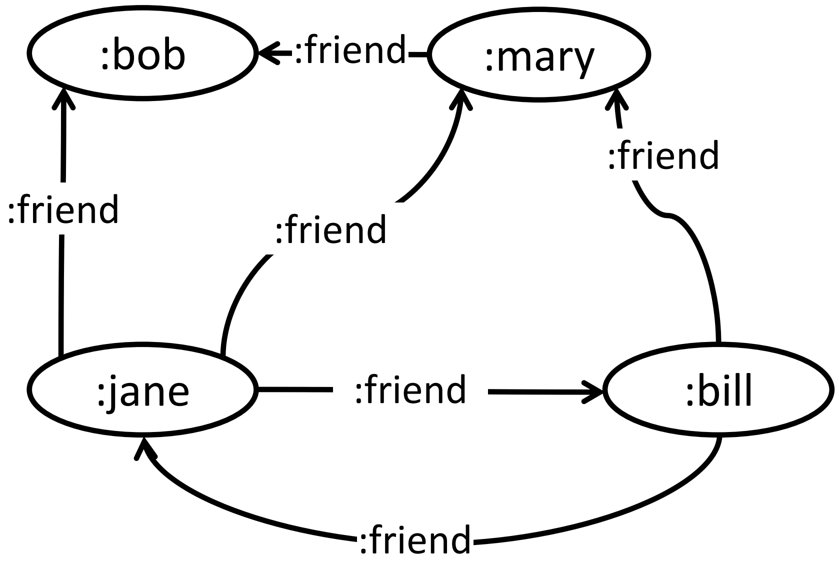
- Running Example: social relationships represented as RDF graph
- Basic graph patterns, pattern matching, solution sequence
- Query forms: SELECT, CONSTRUCT, ASK, DESCRIBE
- Optional graph patterns (OPTIONAL)
- Alternative graph patterns (UNION)
- Selection criteria (FILTER), Scope of FILTERs
- Negation: FILTER NOT EXISTS, MINUS
- Assignment using BIND, VALUES, and in SELECT clause
- Aggregates
- Subqueries and Multistep Aggregation
- Property Paths
- Blank Nodes

Example

RUNNING EXAMPLE: SOCIAL RELATIONSHIPS REPRESENTED AS RDF GRAPH

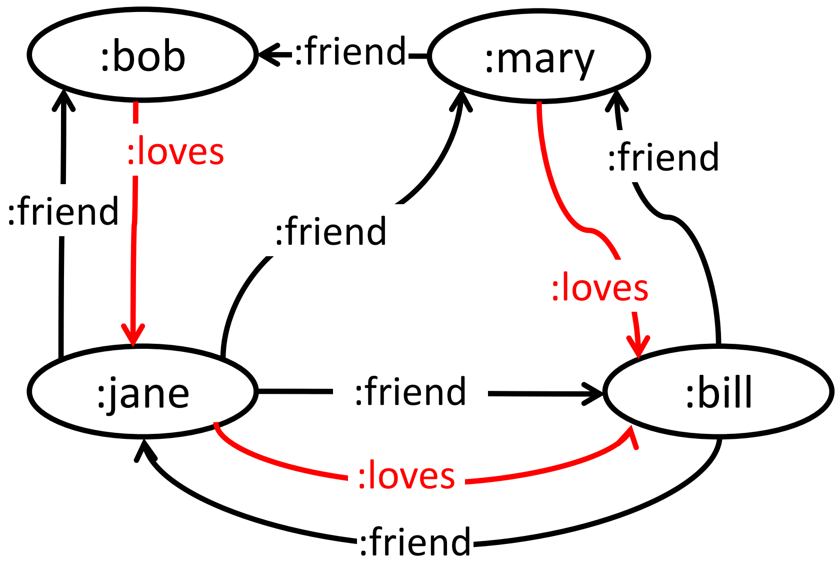
Running example: rather complicated social relationships in RDF

```
@prefix : <http://example.org/> .

:jane

  :friend :mary, :bob, :bill
              .

:mary

  :friend :bob
              .




:bill

  :friend :mary, :jane.
```



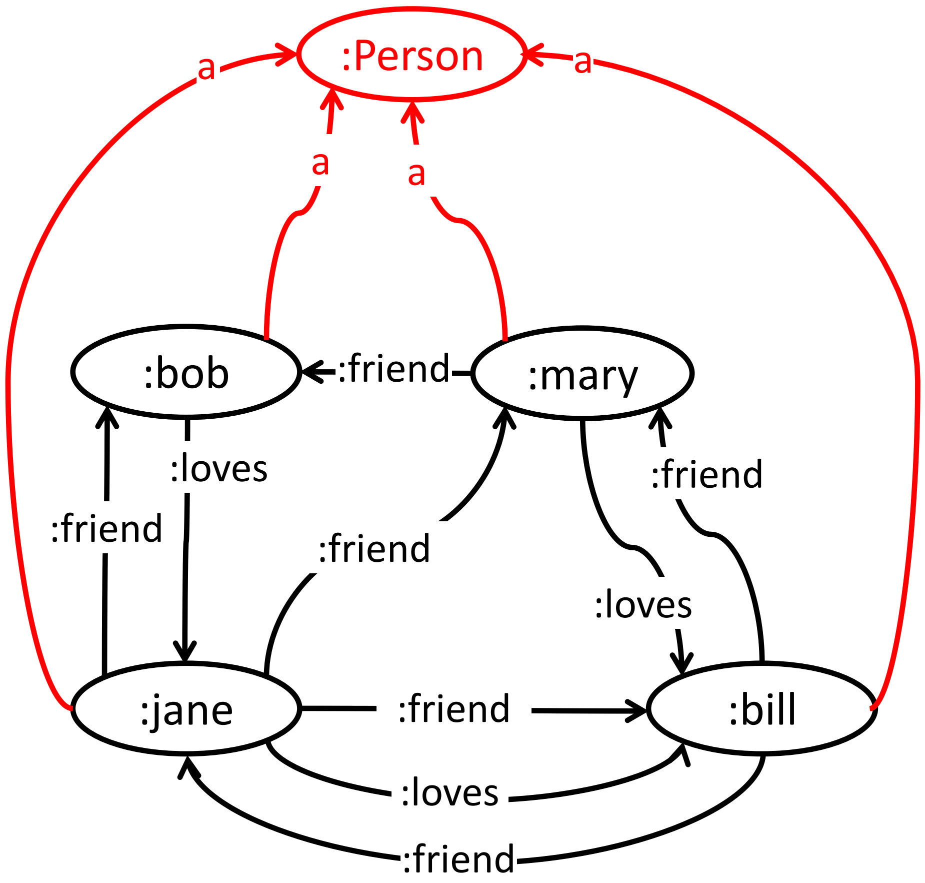
Running example: rather complicated social relationships in RDF

```
@prefix : <http://example.org/> .

:jane

   :friend :mary, :bob, :bill;
   :loves :bill.

:mary

   :friend :bob;
   :loves :bill.

:bob

   :loves :jane.

:bill

   :friend :mary, :jane.
```



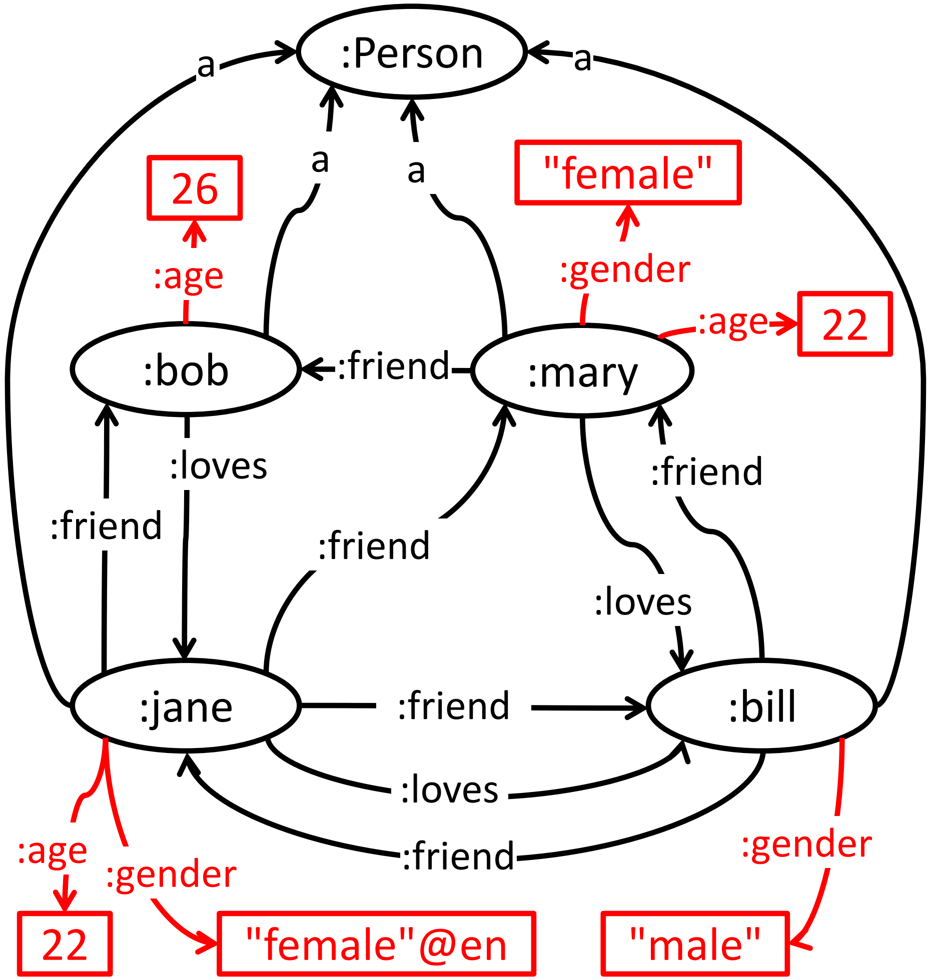
Running example: rather complicated social relationships in RDF

```
@prefix : <http://example.org/> .

:jane  a :Person;

  :friend :mary, :bob, :bill;
  :loves :bill.

:mary  a :Person;

  :friend :bob;
  :loves :bill.

:bob  a :Person;

  :loves :jane.

:bill  a :Person;

  :friend :mary, :jane.
```



Running example: rather complicated social relationships in RDF

```
@prefix : <http://example.org/> .

:jane  a :Person;
  :gender "female"@en;  :age 22;
  :friend :mary, :bob, :bill;
  :loves :bill.

:mary  a :Person;
  :gender "female";  :age 22;
  :friend :bob;
  :loves :bill.

:bob  a :Person;
  :age 26;
  :loves :jane.

:bill  a :Person;
  :gender "male";
  :friend :mary, :jane.
```



Pattern Matching

BASIC GRAPH PATTERNS, PATTERN MATCHING, SOLUTION SEQUENCE

SPARQL Query, Basic Graph Patterns, Solution Sequences

- Most forms of SPARQL query contain a **set of triple patterns** called a **basic graph pattern**.
- Triple patterns are like RDF triples except that each of the subject, predicate and object may be a **variable**.
- A basic graph pattern ***matches*** a subgraph of the RDF data when RDF terms from that subgraph may be substituted for the variables and the result is an RDF graph equivalent to the subgraph.
- The result of a query is a **solution sequence**, corresponding to the ways in which the query's graph pattern matches the data. Each **solution** gives one way in which the selected variables can be bound to RDF terms so that the query pattern matches the data. There may be zero, one or multiple solutions to a query.
- The **result set** of a SELECT query gives all the possible solutions.

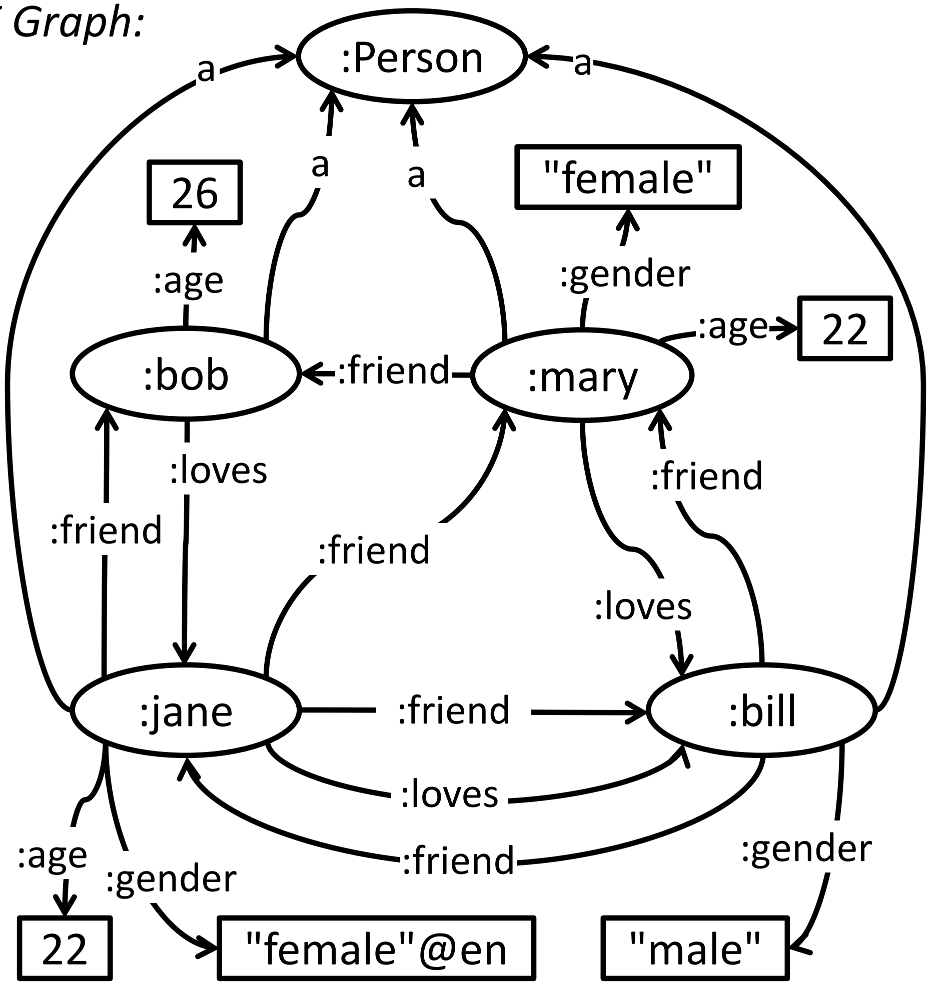
Evaluation of triple patterns

SPARQL Query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://example.org/>

SELECT ?x ?y
WHERE
  { ?x :loves ?y . }
```

Which X loves which Y?

RDF Graph:

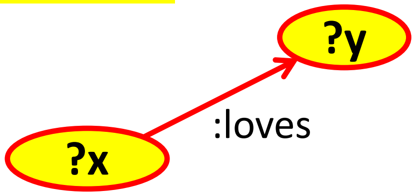


Evaluation of triple patterns

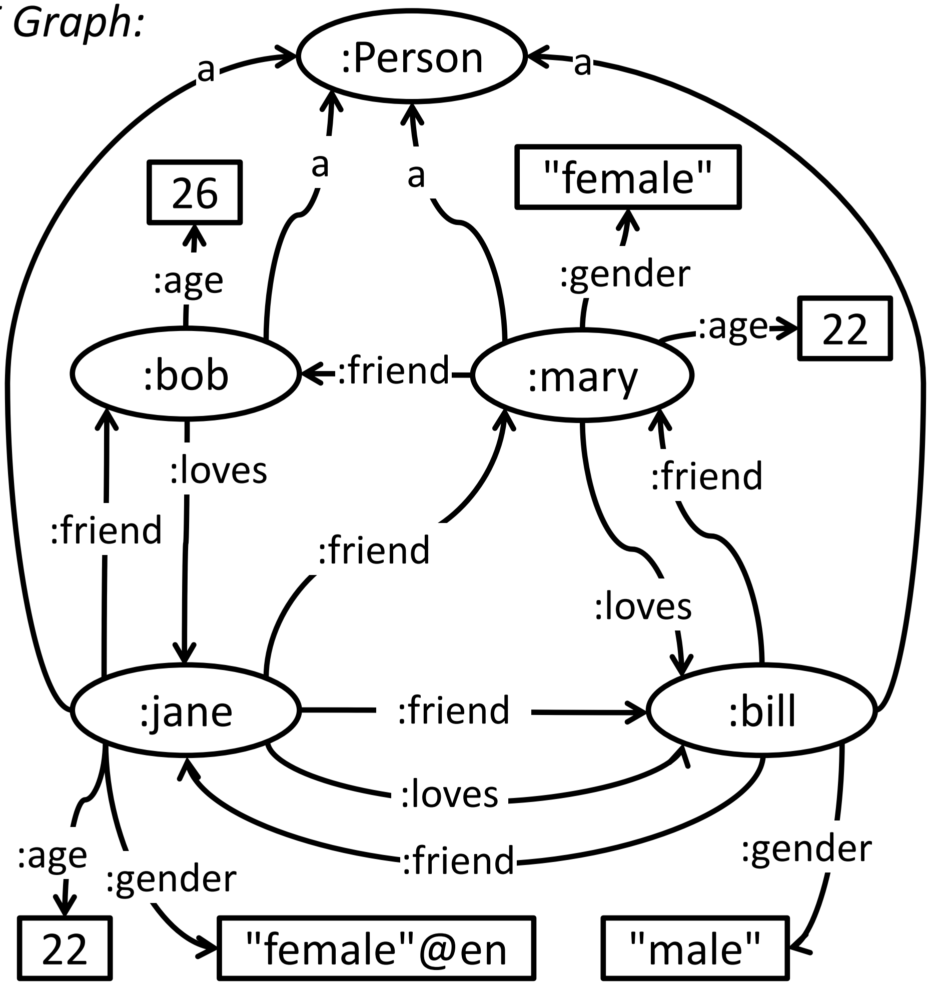
SPARQL Query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://example.org/>

SELECT ?x ?y
WHERE
  { ?x :loves ?y . }
```

Triple Pattern



RDF Graph:

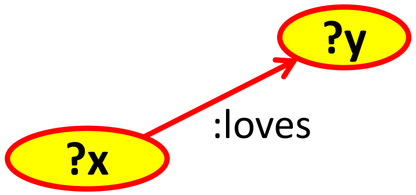


Evaluation of triple patterns

SPARQL Query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://example.org/>

SELECT ?x ?y
WHERE
  { ?x :loves ?y . }
```

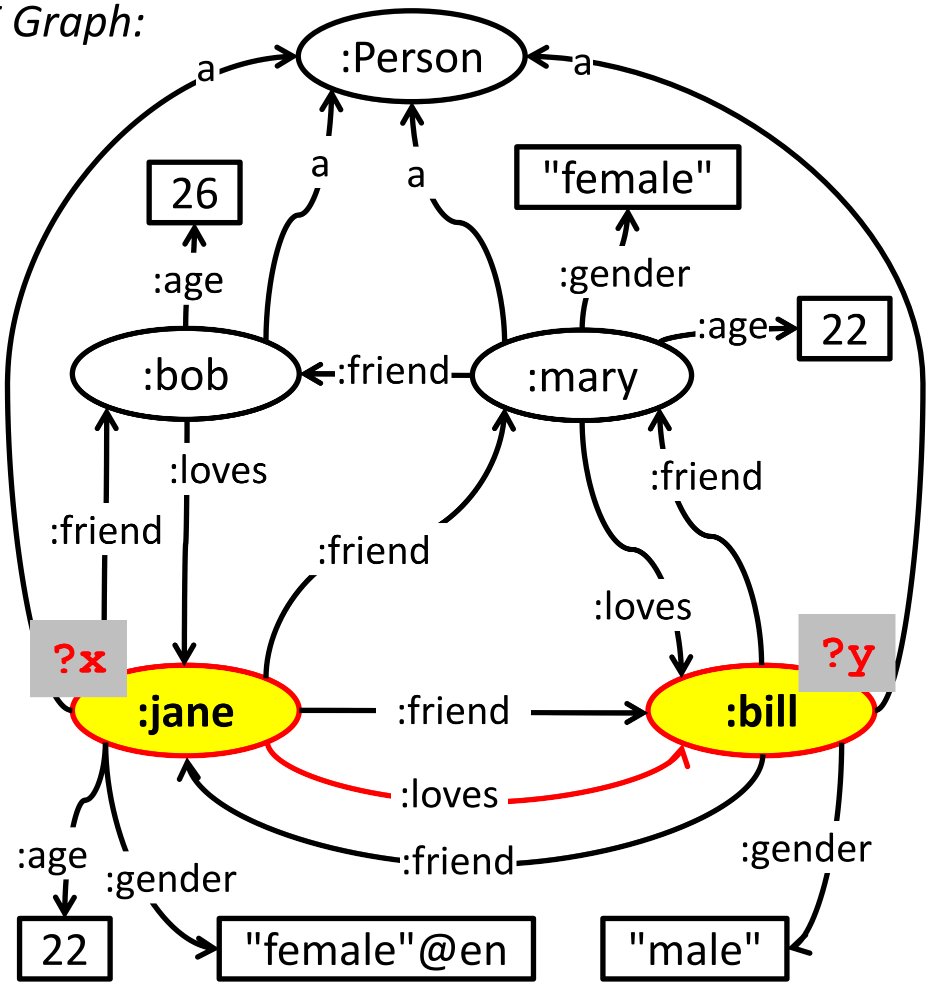
Triple Pattern



Solution Sequence:

```
-------------------
| x        | y        |
===================
| :jane    | :bill    |
|          |          |
|          |          |
-------------------
```

RDF Graph:

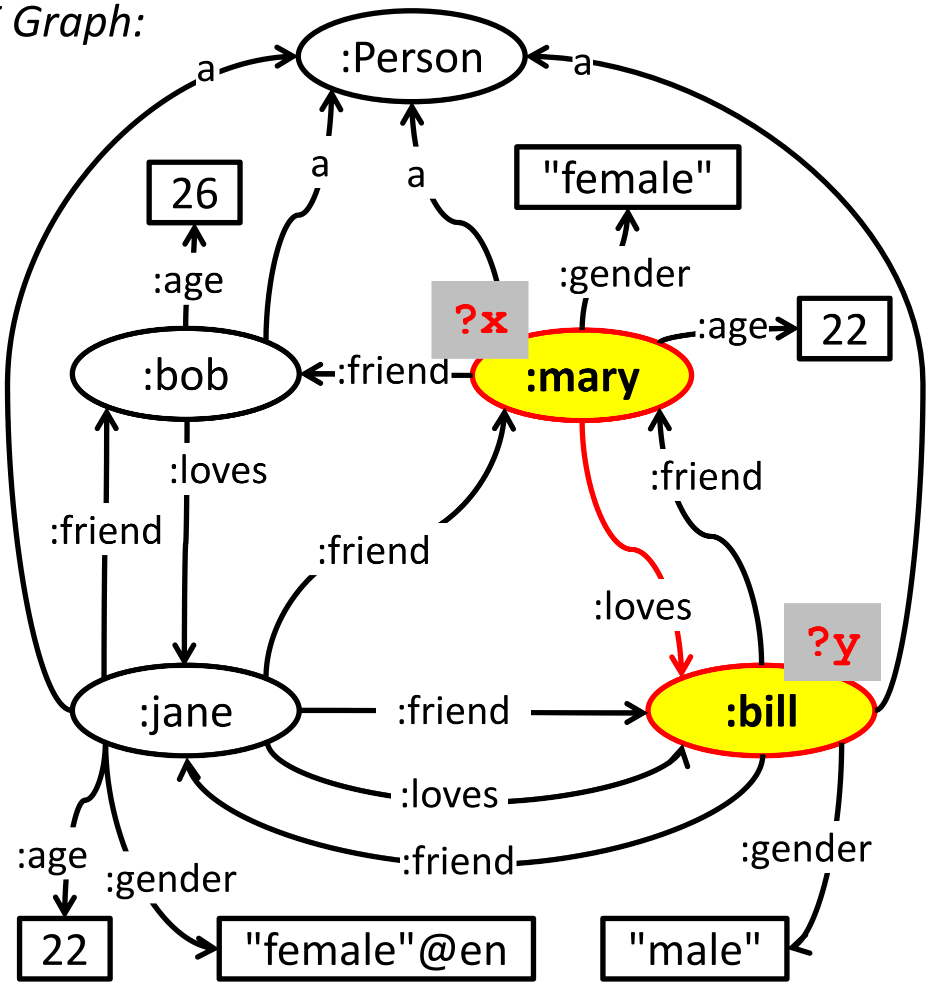


Evaluation of triple patterns

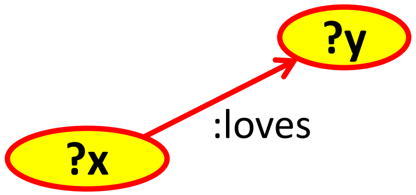
SPARQL Query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://example.org/>

SELECT ?x ?y
WHERE
  { ?x :loves ?y . }
```

RDF Graph:



Triple Pattern



Solution Sequence:

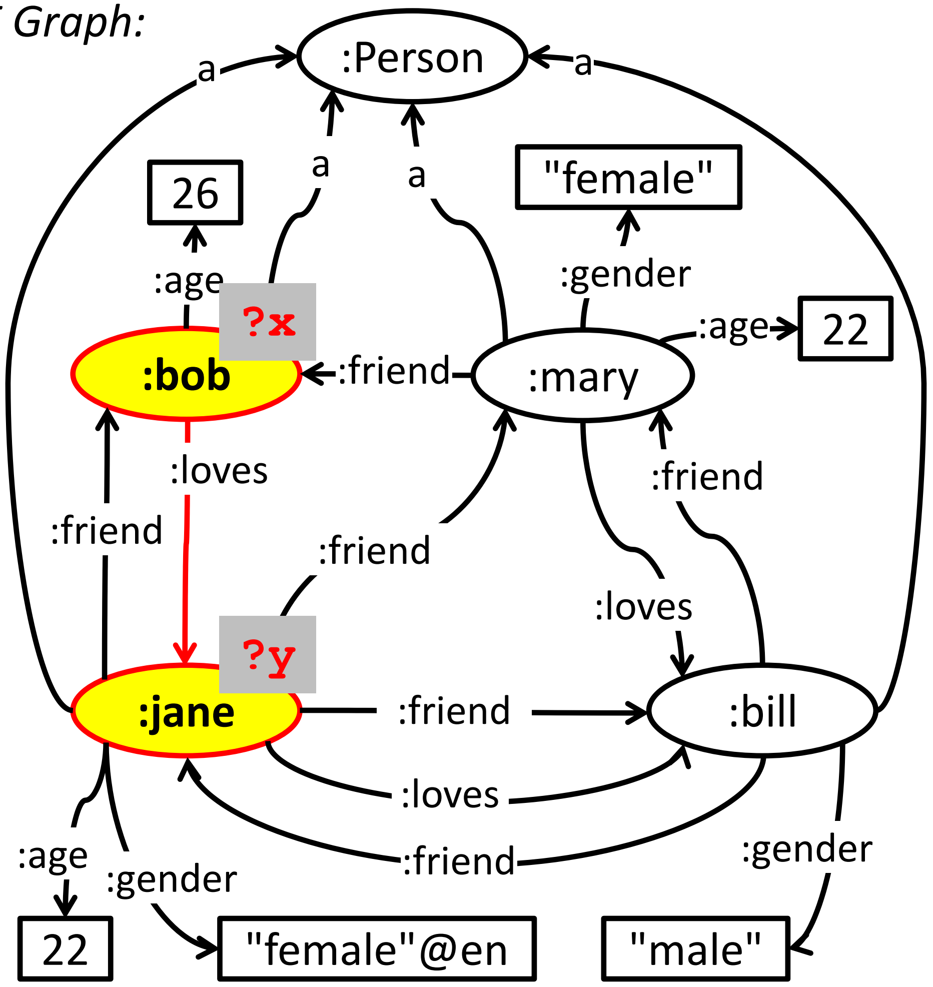
```
-------------------
| x      | y      |
===================
| :jane  | :bill  |
| :mary  | :bill  |
|        |        |
-------------------
```

Evaluation of triple patterns

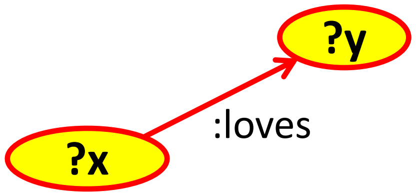
SPARQL Query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://example.org/>

SELECT ?x ?y
WHERE
  { ?x :loves ?y . }
```

RDF Graph:



Triple Pattern



Solution Sequence:

```
-------------------
| x       | y       |
===================
| :jane | :bill |
| :mary | :bill |
| :bob  | :jane |
-------------------
```

Evaluation of basic graph patterns

SPARQL Query:

```
SELECT *
WHERE
  { ?x a :Person; :loves ?y .
    ?y a :Person. }
```

ODER ganz spezifisch:

```
?x a :Person .
?y a :Person .
?x :loves ?y .
```

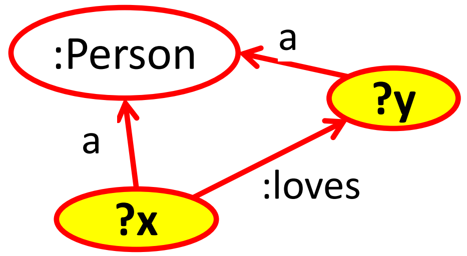
Which person X loves which person Y?

Evaluation of basic graph patterns

SPARQL Query:

```
SELECT *
WHERE
  { ?x a :Person; :loves ?y .
    ?y a :Person. }
```

Basic Graph Pattern (BGP):

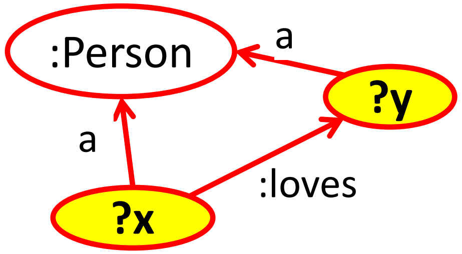


Evaluation of basic graph patterns

SPARQL Query:

```
    SELECT *
    WHERE
      { ?x a :Person; :loves ?y .
        ?y a :Person. }
```

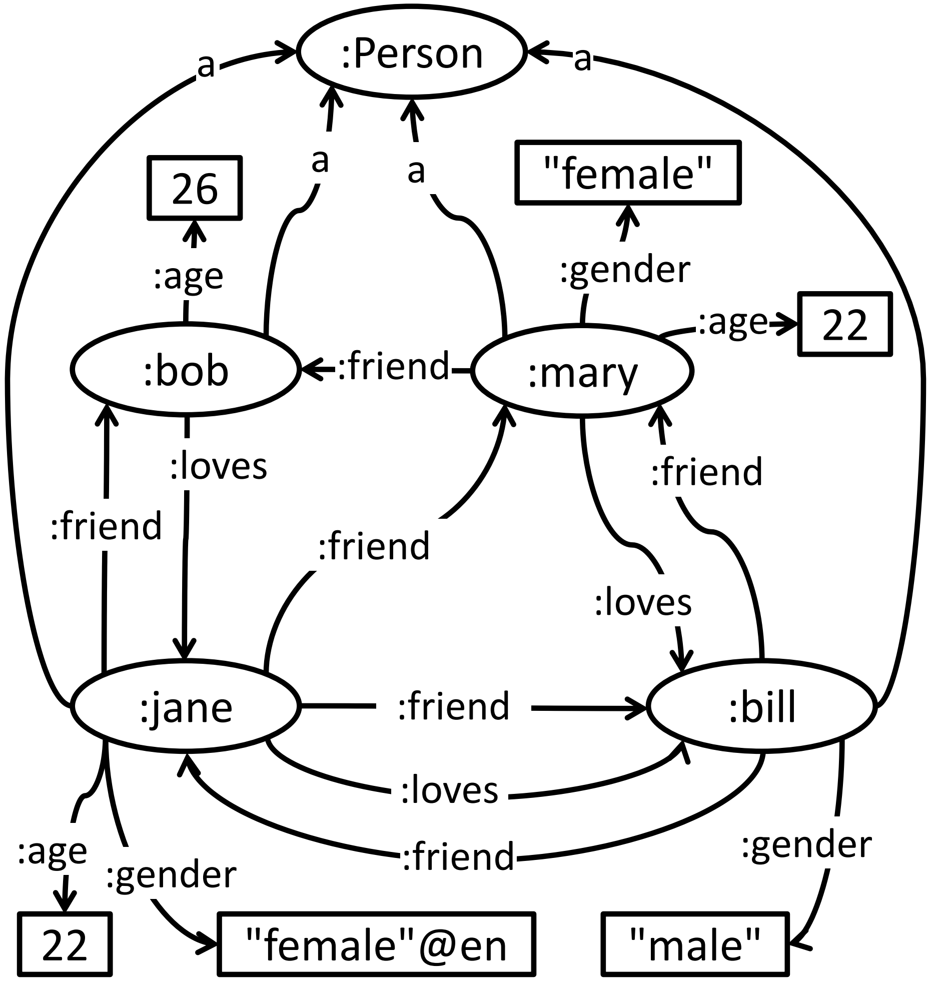
Basic Graph Pattern (BGP):



Solution Sequence:

```
-----------------
| x        | y        |
=================
|                     |
|                     |
|                     |
-----------------
```

RDF Graph:

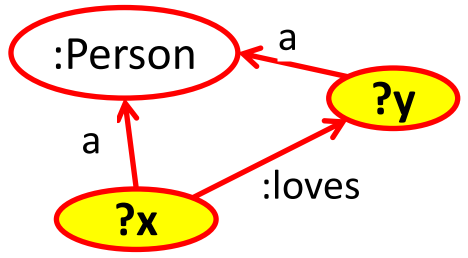


Evaluation of basic graph patterns

SPARQL Query:

```
SELECT *
WHERE
  { ?x a :Person; :loves ?y .
    ?y a :Person. }
```

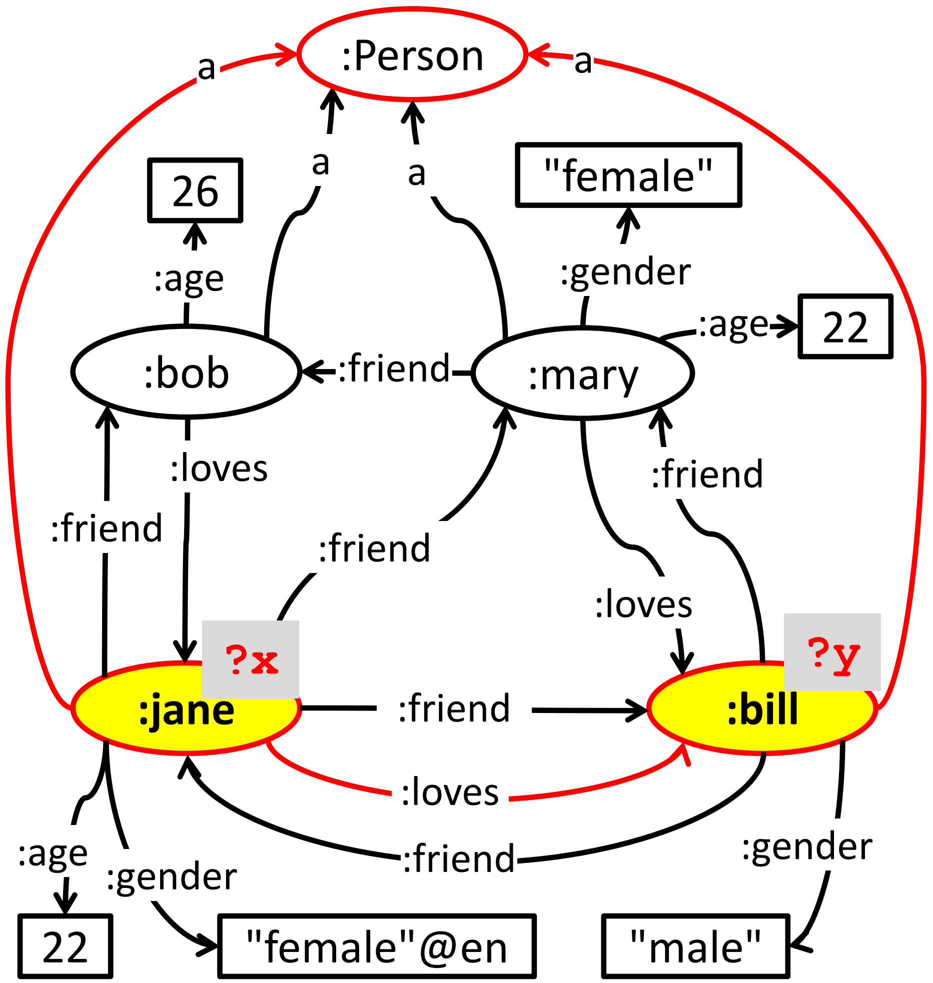
Basic Graph Pattern (BGP):



Solution Sequence:

```
-----------------
| x      | y      |
=================
| :jane  | :bill  |
|                 |
|                 |
-----------------
```

RDF Graph:

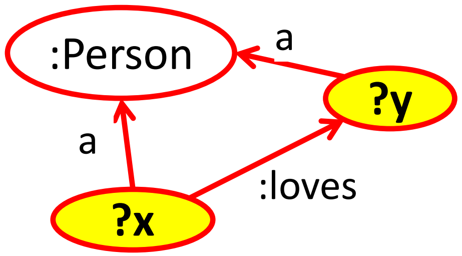


Evaluation of basic graph patterns

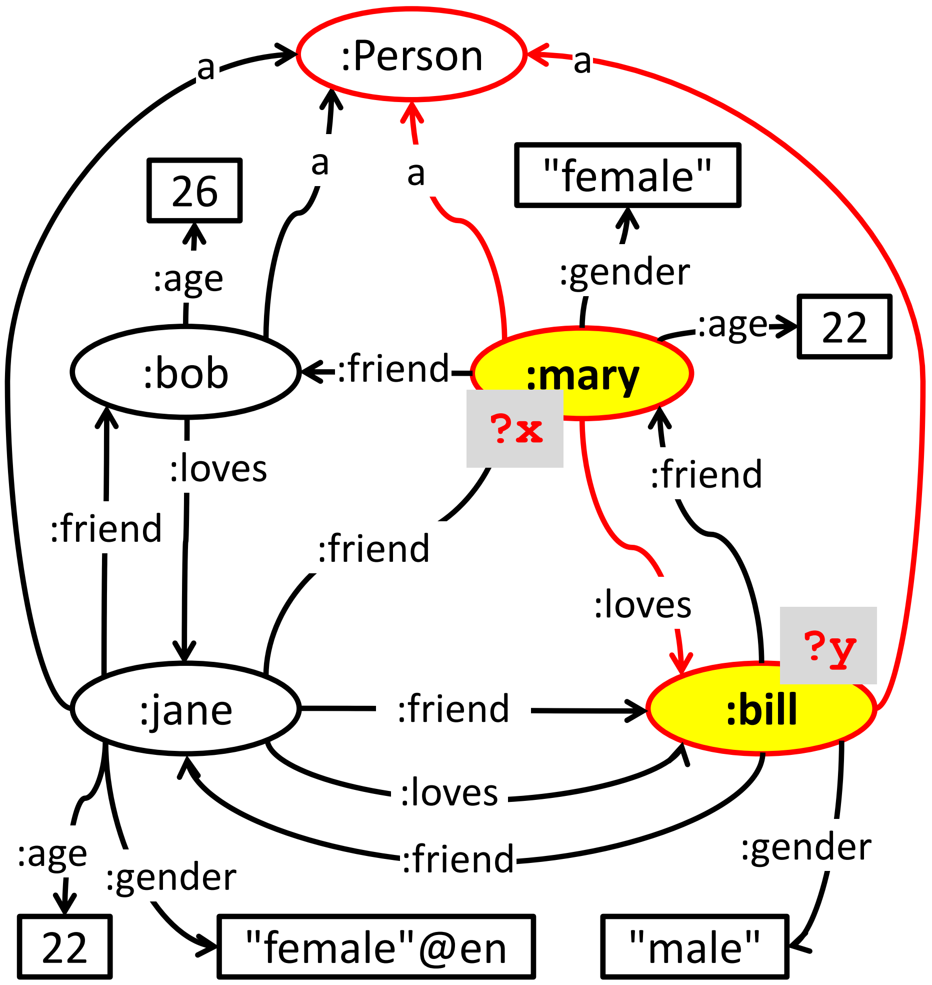
SPARQL Query:

```
    SELECT *
    WHERE
      { ?x a :Person; :loves ?y .
        ?y a :Person. }
```

Basic Graph Pattern (BGP):



RDF Graph:



Solution Sequence:

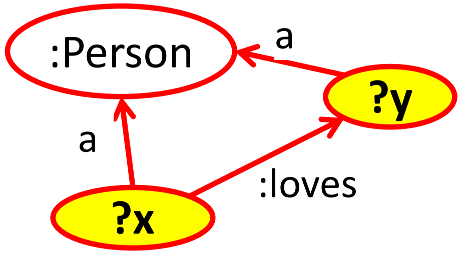
```
-----------------
| a      | b      |
=================
| :jane  | :bill  |
| :mary  | :bill  |
|                 |
-----------------
```

Evaluation of basic graph patterns

SPARQL Query:

```
SELECT *
WHERE
  { ?x a :Person; :loves ?y .
    ?y a :Person. }
```

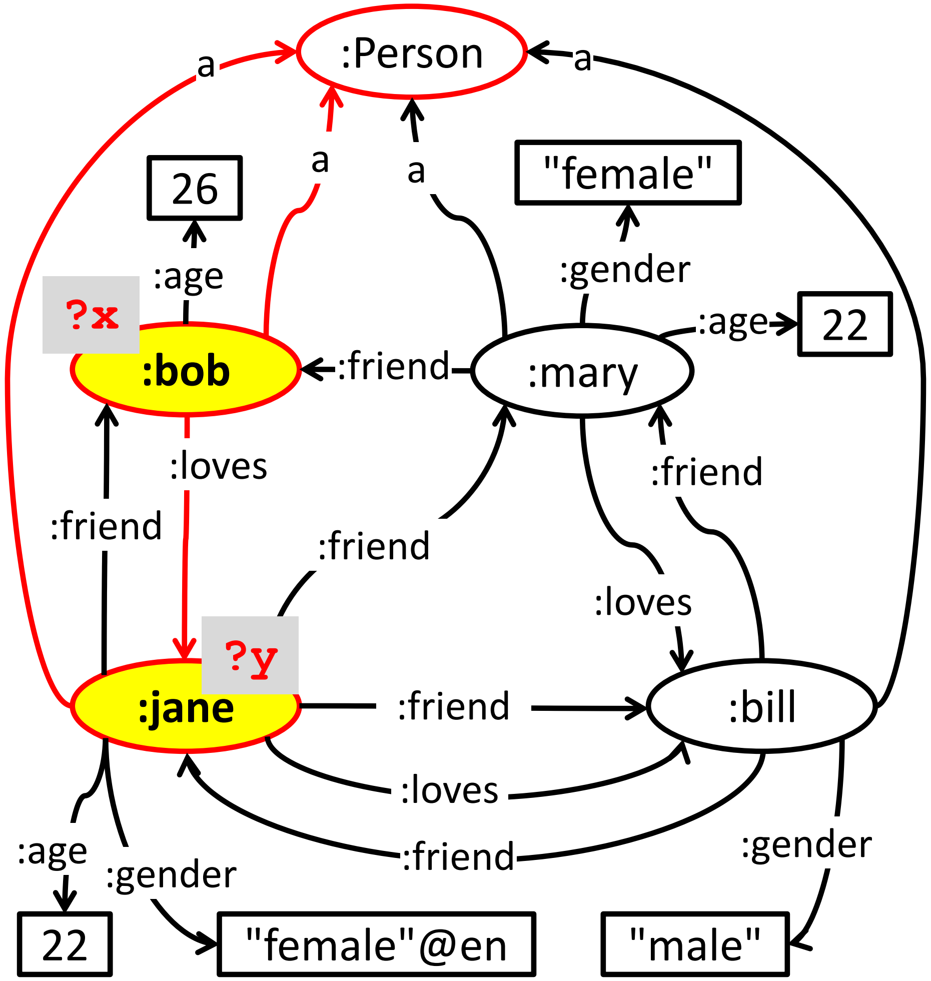
Basic Graph Pattern (BGP):



Solution Sequence:

```
-----------------
| a      | b      |
=================
| :jane  | :bill  |
| :mary  | :bill  |
| :bob   | :jane  |
-----------------
```

RDF Graph:



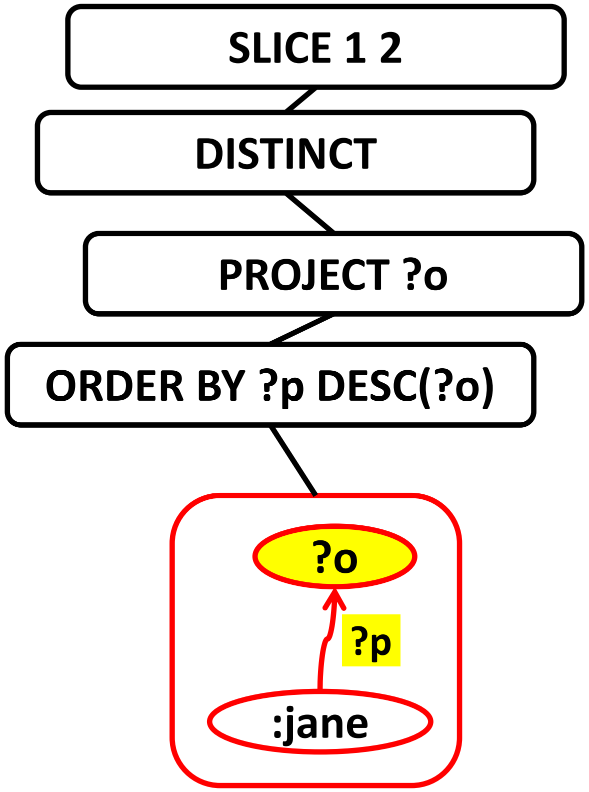
Solution Sequences and Modifiers

- Query patterns generate an unordered collection of solutions, each solution being a partial function from variables to RDF terms. These solutions are then treated as a sequence (a solution sequence), initially in no specific order; any sequence modifiers are then applied to create another sequence. Finally, this latter sequence is used to generate one of the results of a SPARQL query form.

- A solution sequence modifier is one of (applied in this order):
  - Order modifier: put the solutions in order (ORDER BY)
  - Projection modifier: choose certain variables (in the SELECT clause)
  - DISTINCT modifier: ensure solutions in the sequence are unique
  - REDUCED modifier: permit elimination of some non-distinct solutions (not discussed here)
  - OFFSET modifier: control where the solutions start from in the overall sequence of solutions
  - LIMIT modifier: restrict the number of solutions

Solution Sequences and Modifiers

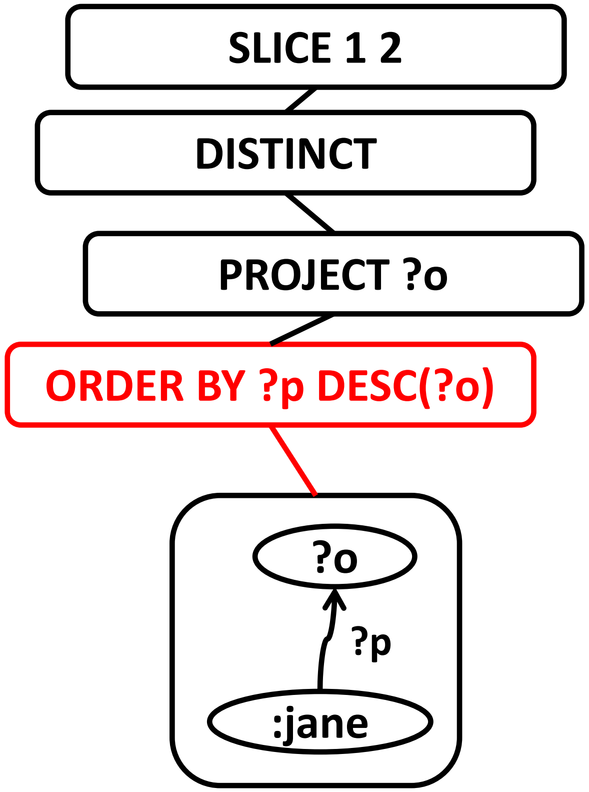
```
SELECT DISTINCT ?o
WHERE
  { :jane ?p ?o. }
ORDER BY ?p DESC(?o)
OFFSET 1 LIMIT 2
```



```
---------------------------
| p          | o           |
===========================
| :loves     | :bill       |
| :friend    | :bill       |
| :friend    | :bob        |
| :friend    | :mary       |
| :age       | 22          |
| :gender    | "female"@en |
| rdf:type   | :Person     |
---------------------------
```

Solution Sequences and Modifiers

```
SELECT DISTINCT ?o
WHERE
  { :jane ?p ?o. }
ORDER BY ?p DESC(?o)
OFFSET 1 LIMIT 2
```

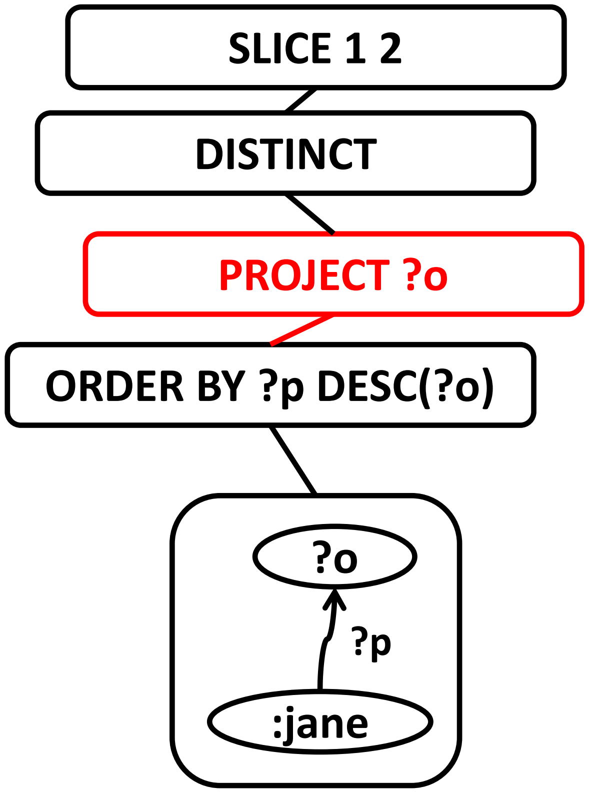


```
-----------------------------
| p         | o             |
=============================
| :age      | 22            |
| :friend   | :mary         |
| :friend   | :bob          |
| :friend   | :bill         |
| :gender   | "female"@en   |
| :loves    | :bill         |
| rdf:type  | :Person       |
-----------------------------
```

```
-----------------------------
| p         | o             |
=============================
| :loves    | :bill         |
| :friend   | :bill         |
| :friend   | :bob          |
| :friend   | :mary         |
| :age      | 22            |
| :gender   | "female"@en   |
| rdf:type  | :Person       |
-----------------------------
```

Solution Sequences and Modifiers

```
SELECT DISTINCT ?o
WHERE
  { :jane ?p ?o. }
ORDER BY ?p DESC(?o)
OFFSET 1 LIMIT 2
```



```
--------------------------
| p        | o             |
==========================
| :age     | 22            |
| :friend  | :mary         |
| :friend  | :bob          |
| :friend  | :bill         |
| :gender  | "female"@en   |
| :loves   | :bill         |
| rdf:type | :Person       |
--------------------------
```

```
--------------------------
| p        | o             |
==========================
| :loves   | :bill         |
| :friend  | :bill         |
| :friend  | :bob          |
| :friend  | :mary         |
| :age     | 22            |
| :gender  | "female"@en   |
| rdf:type | :Person       |
--------------------------
```

```
---------------
| o             |
===============
| 22            |
| :mary         |
| :bob          |
| :bill         |
| "female"@en   |
| :bill         |
| :Person       |
---------------
```

Solution Sequences and Modifiers

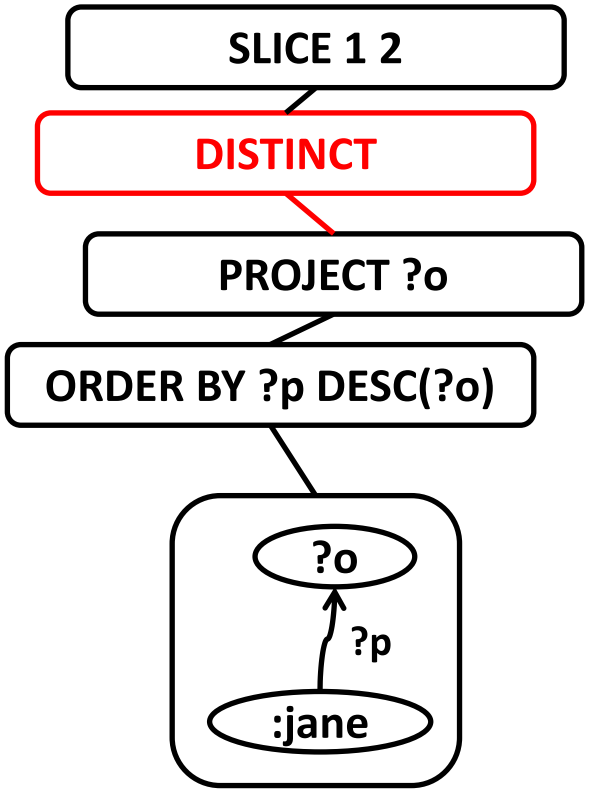
```
SELECT DISTINCT ?o
WHERE
  { :jane ?p ?o. }
ORDER BY ?p DESC(?o)
OFFSET 1 LIMIT 2
```

```
---------------
| o           |
===============
| 22          |
| :mary       |
| :bob        |
| :bill       |
| "female"@en |
| :Person     |
---------------
```

```
---------------
| o           |
===============
| 22          |
| :mary       |
| :bob        |
| :bill       |
| "female"@en |
| :bill       |
| :Person     |
---------------
```

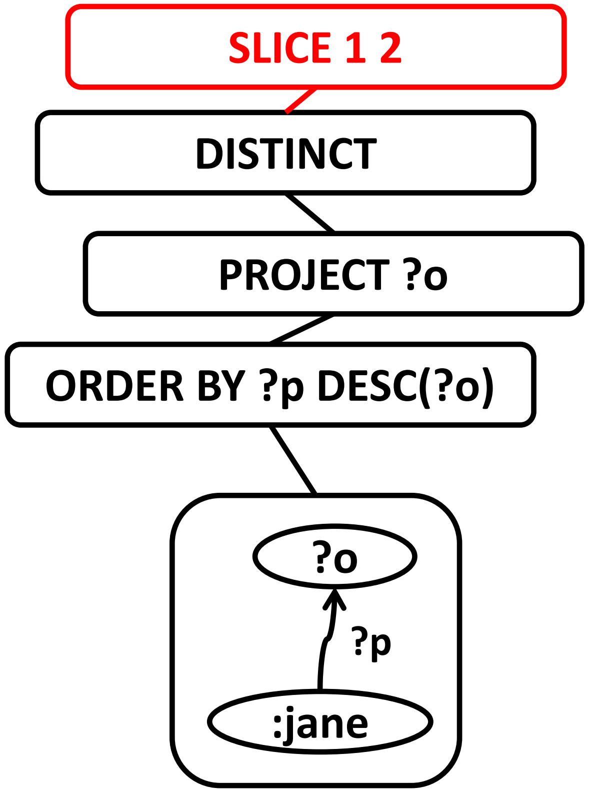
```
--------------------------
| p         | o          |
==========================
| :age      | 22         |
| :friend   | :mary      |
| :friend   | :bob       |
| :friend   | :bill      |
| :gender   | "female"@en|
| :loves    | :bill      |
| rdf:type  | :Person    |
--------------------------
```

```
--------------------------
| p         | o          |
==========================
| :loves    | :bill      |
| :friend   | :bill      |
| :friend   | :bob       |
| :friend   | :mary      |
| :age      | 22         |
| :gender   | "female"@en|
| rdf:type  | :Person    |
--------------------------
```



Solution Sequences and Modifiers

```
SELECT DISTINCT ?o
WHERE
  { :jane ?p ?o. }
ORDER BY ?p DESC(?o)
OFFSET 1 LIMIT 2
```



```
----------
| o      |
==========
| :mary  |
| :bob   |
----------
```

```
---------------
| o           |
===============
| 22          |
| :mary       |
| :bob        |
| :bill       |
| "female"@en |
| :Person     |
---------------
```

```
---------------
| o           |
===============
| 22          |
| :mary       |
| :bob        |
| :bill       |
| "female"@en |
| :bill       |
| :Person     |
---------------
```

```
----------------------------
| p          | o           |
============================
| :age       | 22          |
| :friend    | :mary       |
| :friend    | :bob        |
| :friend    | :bill       |
| :gender    | "female"@en |
| :loves     | :bill       |
| rdf:type   | :Person     |
----------------------------
```

```
----------------------------
| p          | o           |
============================
| :loves     | :bill       |
| :friend    | :bill       |
| :friend    | :bob        |
| :friend    | :mary       |
| :age       | 22          |
| :gender    | "female"@en |
| rdf:type   | :Person     |
----------------------------
```

Query Forms

QUERY FORMS: SELECT, CONSTRUCT, ASK, DESCRIBE

Query Forms

- SPARQL has four query forms. These query forms use the solutions from pattern matching to form result sets or RDF graphs.
- The query forms are:
  - **SELECT**

    Returns all, or a subset of, the variables bound in a query pattern match.
  - **CONSTRUCT**

    Returns an RDF graph constructed by substituting variables in a set of triple templates.
  - **ASK**

    Returns a boolean indicating whether a query pattern matches or not.
  - **DESCRIBE**

    Returns an RDF graph that describes the resources found.

Query Forms: SELECT

SPARQL Query:

```
SELECT ?b
WHERE
  { ?a  a :Person;
        :loves ?b . }
```

– **SELECT**
  Returns all, or a subset of, the variables bound in a query pattern match.

Solution Sequence from Pattern Matching :

```
-----------------
| a      | b      |
=================
| :jane | :bill |
| :mary | :bill |
| :bob  | :jane |
-----------------
```

Result Set:

. . .

Query Forms: SELECT

SPARQL Query:

```
SELECT ?b
WHERE
  { ?a  a :Person;
        :loves ?b . }
```

Solution Sequence from Pattern Matching :

```
-----------------
| a      | b     |
=================
| :jane | :bill |
| :mary | :bill |
| :bob  | :jane |
-----------------
```

Result Set:

```
---------
| b     |
=========
| :bill |
| :bill |
| :jane |
---------
```

Query Forms: CONSTRUCT

SPARQL Query:

```
CONSTRUCT
  { ?a :loves ?b.
    ?b :lovedBy ?a. }
WHERE
  { ?a  a :Person;
        :loves ?b . }
```

– **CONSTRUCT**
  Returns an RDF graph constructed by substituting variables in a set of triple templates.

Solution Sequence
from Pattern Matching :

```
-----------------
| a     | b     |
=================
| :jane | :bill |
| :mary | :bill |
| :bob  | :jane |
-----------------
```

Result Graph:

...

Query Forms: CONSTRUCT

SPARQL Query:

```
CONSTRUCT
  { ?a :loves ?b.
    ?b :lovedBy ?a. }
WHERE
  { ?a  a :Person;
        :loves ?b . }
```

Solution Sequence from Pattern Matching :

```
-----------------
| a       | b       |
=================
| :jane | :bill |
| :mary | :bill |
| :bob  | :jane |
-----------------
```

Result Graph:

```
:bill    :lovedBy  :jane
                         .


:jane
         :loves    :bill .
```

Query Forms: CONSTRUCT

SPARQL Query:

```
CONSTRUCT
  { ?a :loves ?b.
    ?b :lovedBy ?a. }
WHERE
  { ?a  a :Person;
        :loves ?b . }
```

Solution Sequence from Pattern Matching :

```
-------------------
| a        | b       |
===================
| :jane | :bill |
| :mary | :bill |
| :bob  | :jane |
-------------------
```

Result Graph:

```
:bill    :lovedBy   :jane ,
                    :mary .



:jane
         :loves     :bill .

:mary    :loves     :bill .
```

Query Forms: CONSTRUCT

SPARQL Query:

```
CONSTRUCT
  { ?a :loves ?b.
    ?b :lovedBy ?a. }
WHERE
  { ?a  a :Person;
        :loves ?b . }
```

Solution Sequence from Pattern Matching :

```
-------------------
| a        | b      |
===================
| :jane | :bill |
| :mary | :bill |
| :bob   | :jane |
-------------------
```

Result Graph:

```
:bill    :lovedBy   :jane ,
                    :mary .

:bob     :loves     :jane .

:jane    :lovedBy   :bob ;
         :loves     :bill .

:mary    :loves     :bill .
```

Query Forms: ASK

– **ASK**

Returns a boolean indicating whether a query pattern matches or not.

SPARQL Query:

```
ASK
  { ?a  a :Person;
        :loves ?b . }
```

Result:

...

Solution Sequence from Pattern Matching :

```
-----------------
| a     | b     |
=================
| :jane | :bill |
| :mary | :bill |
| :bob  | :jane |
-----------------
```

Query Forms: ASK

SPARQL Query:

```
ASK
  { ?a  a :Person;
        :loves ?b . }
```

Result:

```
YES
```

Solution Sequence
from Pattern Matching :

```
-----------------
| a      | b      |
=================
| :jane | :bill |
| :mary | :bill |
| :bob  | :jane |
-----------------
```

Query Forms: ASK

SPARQL Query:

```
ASK
  { ?a  :loves ?b.
    ?b  :loves ?a. }
```

Result:

```
...
```

Solution Sequence from Pattern Matching :

```
-----------------
| a       | b       |
=================
-----------------
```

Query Forms: ASK

SPARQL Query:

```
ASK
   { ?a   :loves ?b.
     ?b   :loves ?a. }
```

Result:

NO

Solution Sequence
from Pattern Matching :

```
-----------------
| a       | b       |
=================
-----------------
```

Query Forms: DESCRIBE

SPARQL Query:

```
DESCRIBE ?b
WHERE
  { ?a  a :Person;
        :loves ?b . }
```

– **DESCRIBE**
  Returns an RDF graph that describes the resources found.

Solution Sequence from Pattern Matching :

```
-----------------
| a      | b     |
=================
| :jane | :bill |
| :mary | :bill |
| :bob  | :jane |
-----------------
```

Result Graph:

```
...
```

Query Forms: DESCRIBE

SPARQL Query:

```
DESCRIBE ?b
WHERE
  { ?a  a :Person;
       :loves ?b . }
```

Solution Sequence from Pattern Matching :

```
-----------------
| a      | b      |
=================
| :jane | :bill |
| :mary | :bill |
| :bob  | :jane |
-----------------
```

Result Graph:

```
:bill   a        :Person ;
        :friend  :mary , :jane ;
        :gender  "male" .
```

Query Forms: DESCRIBE

SPARQL Query:

```
DESCRIBE ?b
WHERE
  { ?a  a :Person;
        :loves ?b . }
```

Solution Sequence from Pattern Matching :

```
-----------------
| a      | b      |
=================
| :jane  | :bill  |
| :mary  | :bill  |
| :bob   | :jane  |
-----------------
```

Result Graph:

```
:bill    a          :Person ;
         :friend    :mary , :jane ;
         :gender    "male" .
```

Query Forms: DESCRIBE

SPARQL Query:

```
DESCRIBE ?b
WHERE
  { ?a  a :Person;
        :loves ?b . }
```

Solution Sequence from Pattern Matching :

```
-------------------
| a      | b      |
===================
| :jane  | :bill  |
| :mary  | :bill  |
| :bob   | :jane  |
-------------------
```

Result Graph:

```
:bill   a        :Person ;
        :friend  :mary , :jane ;
        :gender  "male" .

:jane   a        :Person ;
        :age     22 ;
        :friend  :mary , :bob , :bill ;
        :gender  "female"@en ;
        :loves   :bill .
```

OPTIONAL

OPTIONAL GRAPH PATTERNS (OPTIONAL)

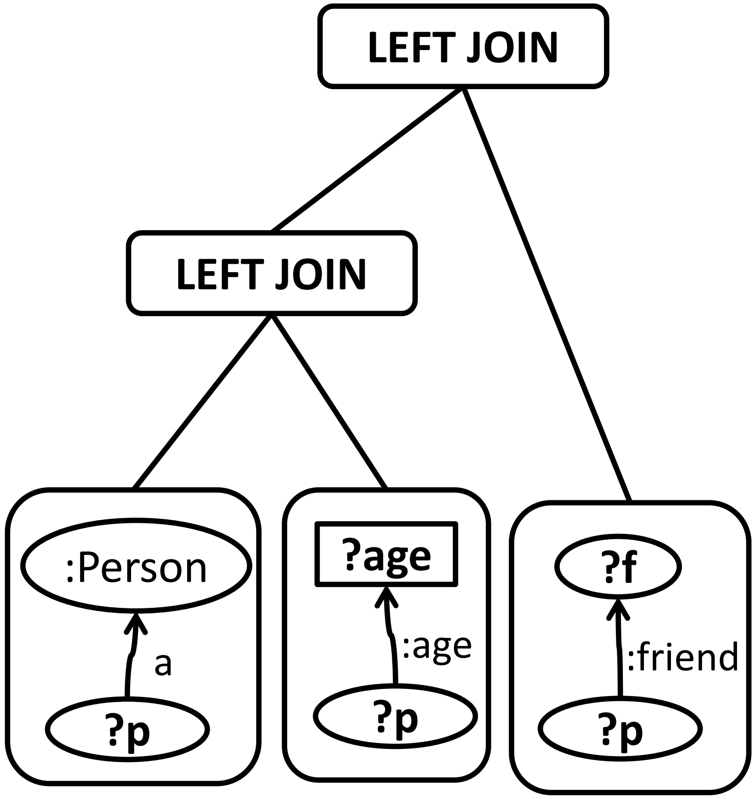
Optional graph patterns

- Complete structures cannot be assumed in all RDF graphs. It is useful to be able to have queries that allow information to be added to the solution where the information is available, but do not reject the solution because some part of the query pattern does not match.
- Optional parts of the graph pattern may be specified syntactically with the OPTIONAL keyword applied to a graph pattern.
- Graph patterns are defined recursively. A graph pattern may have zero or more optional graph patterns.
- An optional graph pattern translates to a **left join** in the **SPARQL algebra**.

Optional graph patterns

SPARQL Query:

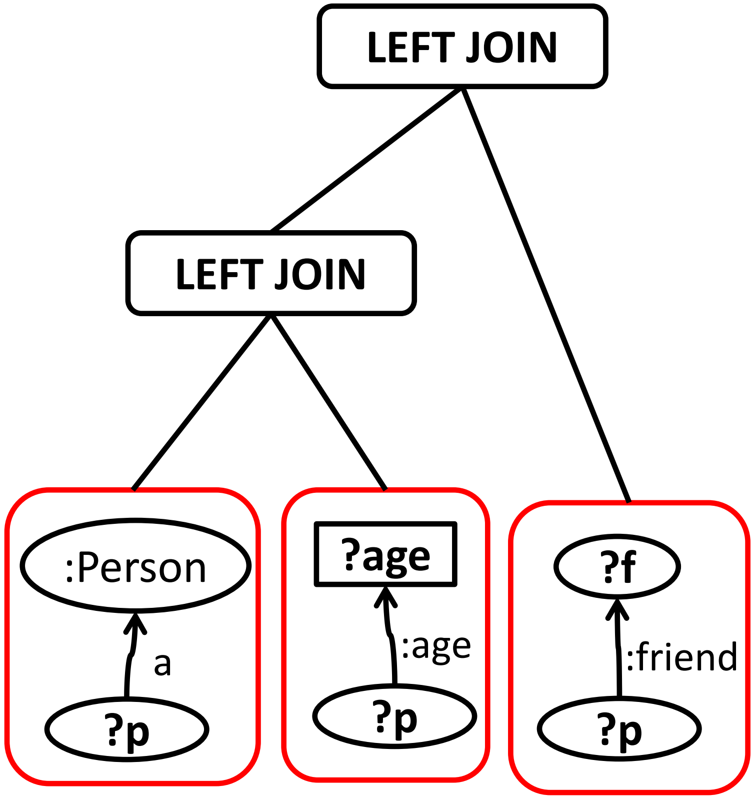
```
SELECT *
WHERE
  { ?p  a :Person.
    OPTIONAL { ?p :age ?age. }
    OPTIONAL { ?p :friend ?f .}
  }
```



Optional graph patterns

SPARQL Query:

```
SELECT *
WHERE
  { ?p  a :Person.
    OPTIONAL { ?p :age ?age. }
    OPTIONAL { ?p :friend ?f .}
  }
```



```
---------
| p     |
=========
| :bill |
| :bob  |
| :mary |
| :jane |
---------
```

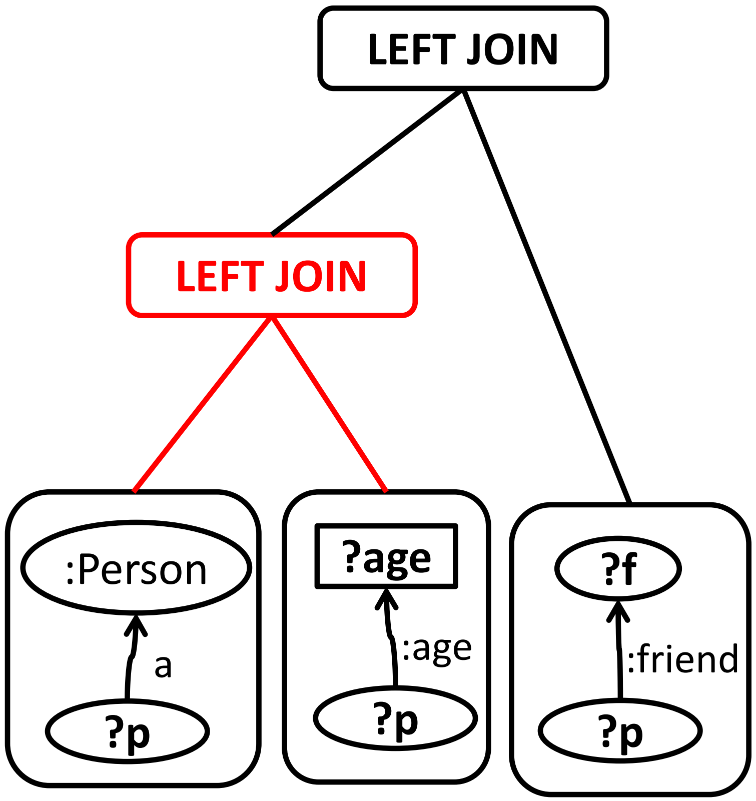
```
----------------
| p     | age |
================
| :bob  | 26  |
| :mary | 22  |
| :jane | 22  |
----------------
```

```
------------------
| p     | f      |
==================
| :bill | :jane  |
| :bill | :mary  |
| :mary | :bob   |
| :jane | :bill  |
| :jane | :bob   |
| :jane | :mary  |
------------------
```

Optional graph patterns

SPARQL Query:

```
SELECT *
WHERE
  { ?p  a :Person.
    OPTIONAL { ?p :age ?age. }
    OPTIONAL { ?p :friend ?f .}
  }
```



```
---------------
| p      | age |
===============
| :bill  |     |
| :bob   | 26  |
| :mary  | 22  |
| :jane  | 22  |
---------------
```

```
---------
| p      |
=========
| :bill  |
| :bob   |
| :mary  |
| :jane  |
---------
```

```
---------------
| p      | age |
===============
| :bob   | 26  |
| :mary  | 22  |
| :jane  | 22  |
---------------
```

```
------------------
| p      | f       |
==================
| :bill  | :jane   |
| :bill  | :mary   |
| :mary  | :bob    |
| :jane  | :bill   |
| :jane  | :bob    |
| :jane  | :mary   |
------------------
```

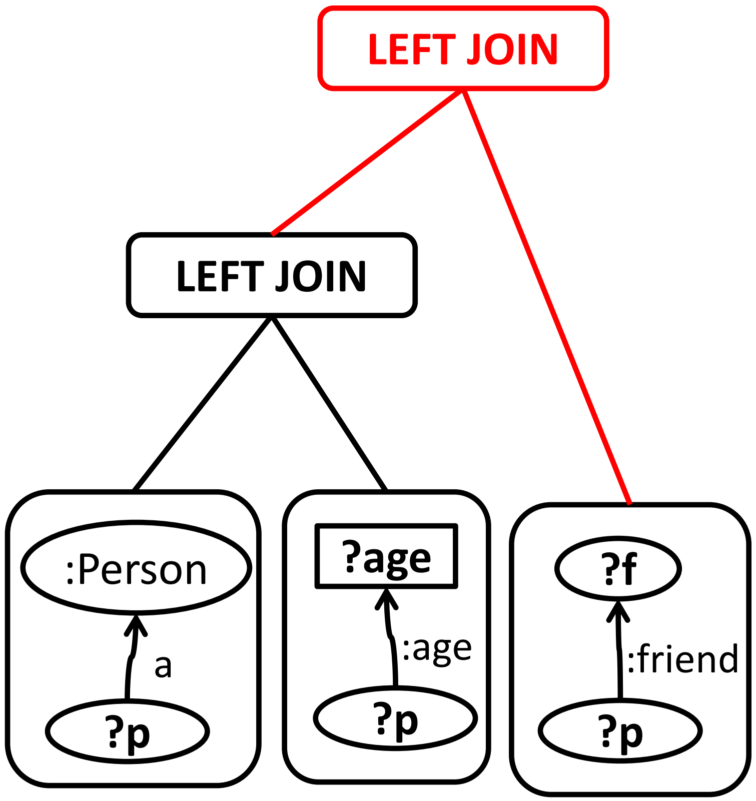
Optional graph patterns

SPARQL Query:

Optional Graph Patterns in SELECT Queries

```
SELECT *
WHERE
  { ?p  a :Person.
    OPTIONAL { ?p :age ?age. }
    OPTIONAL { ?p :friend ?f .}
  }
```

```
------------------------
| p     | age | f      |
========================
| :bill |     | :jane  |
| :bill |     | :mary  |
| :bob  | 26  |        |
| :mary | 22  | :bob   |
| :jane | 22  | :bill  |
| :jane | 22  | :bob   |
| :jane | 22  | :mary  |
------------------------
```



```
---------------
| p     | age |
===============
| :bill |     |
| :bob  | 26  |
| :mary | 22  |
| :jane | 22  |
---------------
```

```
---------
| p     |
=========
| :bill |
| :bob  |
| :mary |
| :jane |
---------
```

```
---------------
| p     | age |
===============
| :bob  | 26  |
| :mary | 22  |
| :jane | 22  |
---------------
```

```
------------------
| p     | friend |
==================
| :bill | :jane  |
| :bill | :mary  |
| :mary | :bob   |
| :jane | :bill  |
| :jane | :bob   |
| :jane | :mary  |
------------------
```

Optional graph patterns in CONSTRUCT queries

- The result of a construct query is an RDF graph formed by taking each query solution in the solution sequence, substituting for the variables in the graph template, and combining the triples into a single RDF graph by set union.
- If any such instantiation produces a triple **containing an unbound variable** or an illegal RDF construct, such as a literal in subject or predicate position, **then that triple is not included in the output RDF graph**. --> gibt es dazu ein bsp?*

Optional graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  { ?p a :Person.
    ?p :age ?age.
    ?p :friend ?f. }
WHERE
  { ?p  a :Person.
    OPTIONAL { ?p :age ?age. }
    OPTIONAL { ?p :friend ?f .}
  }
```

Result Graph:

. . .

Solution Sequence from Pattern Matching :

```
------------------------
| p     | age | f      |
========================
| :bill |     | :jane  |
| :bill |     | :mary  |
| :bob  | 26  |        |
| :mary | 22  | :bob   |
| :jane | 22  | :bill  |
| :jane | 22  | :bob   |
| :jane | 22  | :mary  |
------------------------
```

Optional graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  { ?p a :Person.
    ?p :age ?age.
    ?p :friend ?f. }
WHERE
  { ?p  a :Person.
    OPTIONAL { ?p :age ?age. }
    OPTIONAL { ?p :friend ?f .}
  }
```

Solution Sequence from Pattern Matching :

```
------------------------
| p      | age | f        |
========================
| :bill |      | :jane   |
| :bill |      | :mary   |
| :bob   | 26   |          |
| :mary | 22   | :bob     |
| :jane | 22   | :bill   |
| :jane | 22   | :bob     |
| :jane | 22   | :mary   |
------------------------
```

Result Graph:

```
:bill    a          :Person ;
         :friend            :jane .
```

Optional graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  { ?p a :Person.
    ?p :age ?age.
    ?p :friend ?f. }
WHERE
  { ?p  a :Person.
    OPTIONAL { ?p :age ?age. }
    OPTIONAL { ?p :friend ?f .}
  }
```

Result Graph:

```
:bill   a         :Person ;
        :friend   :mary , :jane .
```

Solution Sequence from Pattern Matching :

```
------------------------
| p     | age | f      |
========================
| :bill |     | :jane  |
| :bill |     | :mary  |
| :bob  | 26  |        |
| :mary | 22  | :bob   |
| :jane | 22  | :bill  |
| :jane | 22  | :bob   |
| :jane | 22  | :mary  |
------------------------
```

Optional graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  { ?p a :Person.
    ?p :age ?age.
    ?p :friend ?f. }
WHERE
  { ?p  a :Person.
    OPTIONAL { ?p :age ?age. }
    OPTIONAL { ?p :friend ?f .}
  }
```

Solution Sequence from Pattern Matching :

```
------------------------
| p      | age | f      |
========================
| :bill |     | :jane  |
| :bill |     | :mary  |
| :bob  | 26  |        |
| :mary | 22  | :bob   |
| :jane | 22  | :bill  |
| :jane | 22  | :bob   |
| :jane | 22  | :mary  |
------------------------
```

Result Graph:

```
:bill    a         :Person ;
         :friend   :mary , :jane .

:bob     a        :Person ;
         :age     26 .
```

Optional graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  { ?p a :Person.
    ?p :age ?age.
    ?p :friend ?f. }
WHERE
  { ?p  a :Person.
    OPTIONAL { ?p :age ?age. }
    OPTIONAL { ?p :friend ?f .}
  }
```

Solution Sequence from Pattern Matching :

```
------------------------
| p      | age | f      |
========================
| :bill |     | :jane  |
| :bill |     | :mary  |
| :bob  | 26  |        |
| :mary | 22  | :bob   |
| :jane | 22  | :bill  |
| :jane | 22  | :bob   |
| :jane | 22  | :mary  |
------------------------
```

Result Graph:

```
:bill    a         :Person ;
         :friend   :mary , :jane .

:bob     a        :Person ;
         :age     26 .

:mary    a         :Person ;
         :age      22 ;
         :friend   :bob .
```

Optional graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  { ?p a :Person.
    ?p :age ?age.
    ?p :friend ?f. }
WHERE
  { ?p  a :Person.
    OPTIONAL { ?p :age ?age. }
    OPTIONAL { ?p :friend ?f .}
  }
```

Solution Sequence from Pattern Matching :

```
------------------------
| p      | age | f      |
========================
| :bill |      | :jane  |
| :bill |      | :mary  |
| :bob   | 26  |        |
| :mary | 22  | :bob   |
| :jane | 22  | :bill  |
| :jane | 22  | :bob   |
| :jane | 22  | :mary  |
------------------------
```

Result Graph:

```
:bill    a         :Person ;
         :friend   :mary , :jane .

:bob     a        :Person ;
         :age     26 .

:jane    a         :Person ;
         :age      22 ;
         :friend

                   :bill .

:mary    a         :Person ;
         :age      22 ;
         :friend   :bob .
```

Optional graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  { ?p a :Person.
    ?p :age ?age.
    ?p :friend ?f. }
WHERE
  { ?p  a :Person.
    OPTIONAL { ?p :age ?age. }
    OPTIONAL { ?p :friend ?f .}
  }
```

Solution Sequence from Pattern Matching :

```
-------------------------
| p      | age | f       |
=========================
| :bill |      | :jane   |
| :bill |      | :mary   |
| :bob   | 26  |         |
| :mary | 22   | :bob    |
| :jane | 22   | :bill   |
| :jane | 22   | :bob    |
| :jane | 22   | :mary   |
-------------------------
```

Result Graph:

```
:bill    a         :Person ;
         :friend   :mary , :jane .

:bob     a         :Person ;
         :age     26 .

:jane    a         :Person ;
         :age      22 ;
         :friend            :bob ,
                   :bill .

:mary    a         :Person ;
         :age      22 ;
         :friend   :bob .
```

Optional graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  { ?p a :Person.
    ?p :age ?age.
    ?p :friend ?f. }
WHERE
  { ?p  a :Person.
    OPTIONAL { ?p :age ?age. }
    OPTIONAL { ?p :friend ?f .}
  }
```

Solution Sequence from Pattern Matching :

```
------------------------
| p     | age | f       |
========================
| :bill |     | :jane   |
| :bill |     | :mary   |
| :bob  | 26  |         |
| :mary | 22  | :bob    |
| :jane | 22  | :bill   |
| :jane | 22  | :bob    |
| :jane | 22  | :mary   |
------------------------
```

Result Graph:

```
:bill   a        :Person ;
        :friend  :mary , :jane .

:bob    a       :Person ;
        :age    26 .

:jane   a        :Person ;
        :age     22 ;
        :friend  :mary , :bob ,
                 :bill .

:mary   a        :Person ;
        :age     22 ;
        :friend  :bob .
```

UNION

ALTERNATIVE GRAPH PATTERNS (UNION)

Alternative Graph Patterns (UNION)

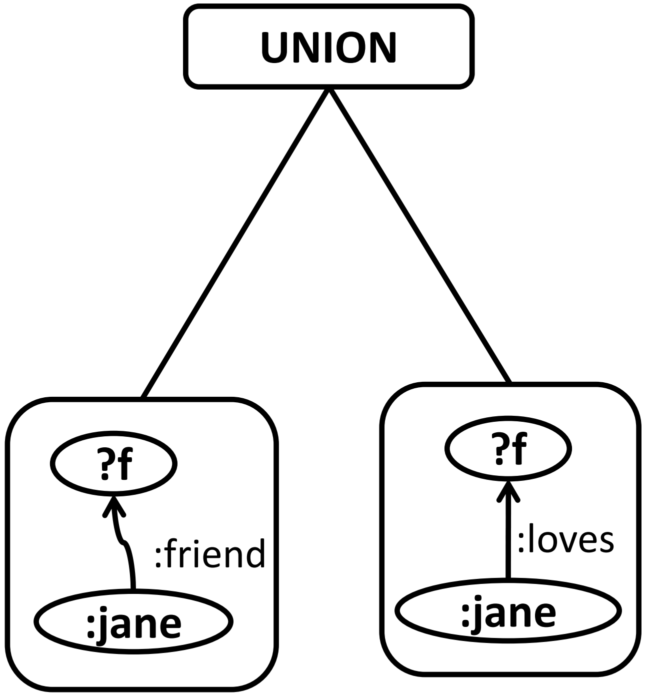
- SPARQL provides a means of combining graph patterns so that one of several alternative graph patterns may match. If more than one of the alternatives matches, all the possible pattern solutions are found.
- Pattern alternatives are syntactically specified with the UNION keyword.

- To determine exactly how the information was recorded, a query can use different variables for alternative patterns. This is especially useful in CONSTRUCT queries.

Alternative graph patterns

SPARQL Query:

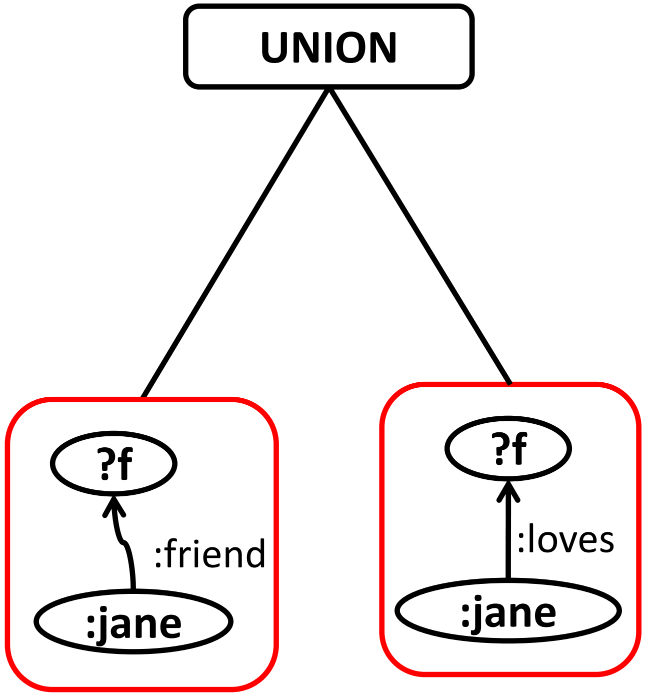
```
SELECT *
WHERE
  {   { :jane :friend ?f }
    UNION
      { :jane :loves ?f }
  }
```



Alternative graph patterns

SPARQL Query:

```
SELECT *
WHERE
  {   { :jane :friend ?f }
    UNION
      { :jane :loves ?f }
  }
```



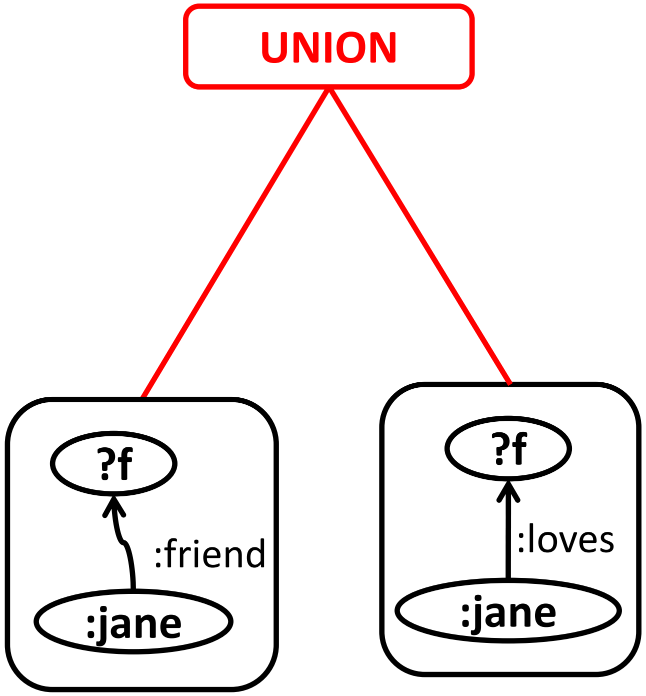
```
---------
| f     |
=========
| :bill |
| :bob  |
| :mary |
---------
```

```
---------
| f     |
=========
| :bill |
---------
```

Alternative graph patterns

SPARQL Query:

```
SELECT *
WHERE
  {   { :jane :friend ?f }
    UNION
      { :jane :loves ?f }
  }
```



```
---------
| f       |
=========
| :bill |
| :bob  |
| :mary |
| :bill |
---------
```

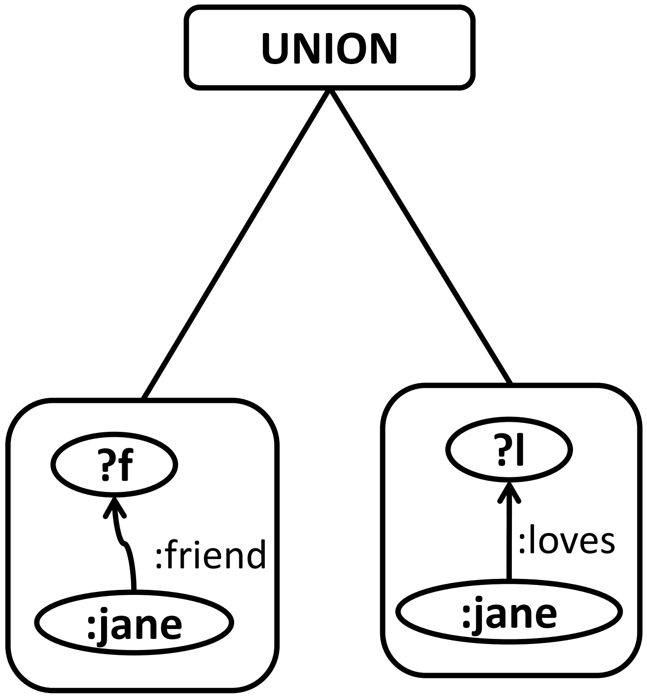
```
---------
| f       |
=========
| :bill |
| :bob  |
| :mary |
---------
```

```
---------
| f       |
=========
| :bill |
---------
```

Alternative graph patterns with different variables

SPARQL Query:

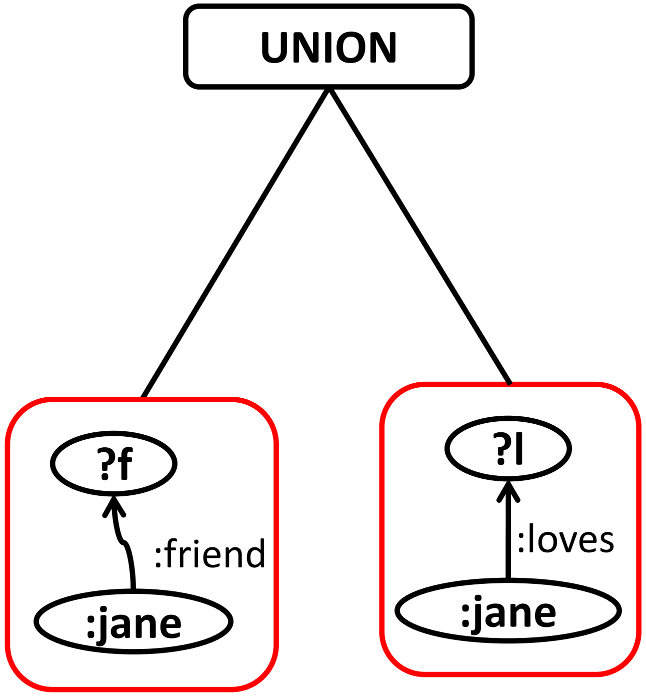
```
SELECT *
WHERE
  {   { :jane :friend ?f }
    UNION
      { :jane :loves ?l }
  }
```



Alternative graph patterns with different variables

SPARQL Query:

```
SELECT *
WHERE
  {    { :jane :friend ?f }
    UNION
       { :jane :loves ?l }
  }
```



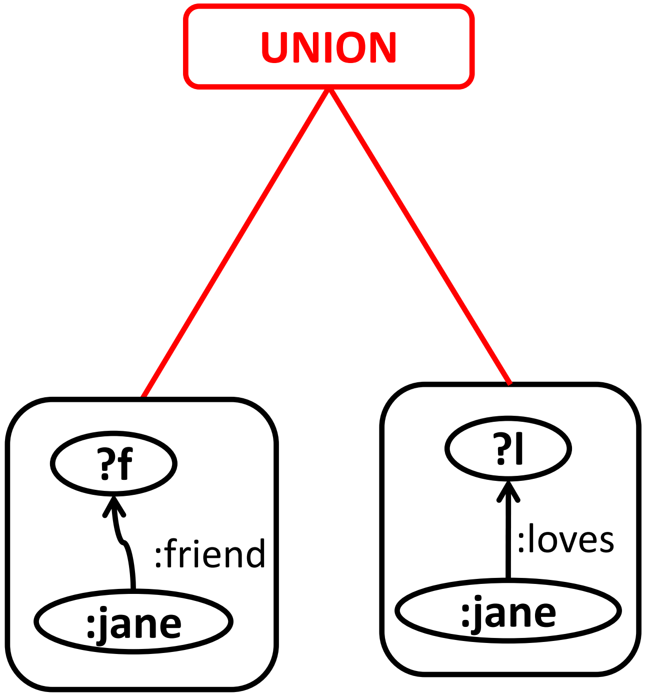
```
---------
| f     |
=========
| :bill |
| :bob  |
| :mary |
---------
```

```
---------
| l     |
=========
| :bill |
---------
```

Alternative graph patterns with different variables

SPARQL Query:

```
SELECT *
WHERE
  {    { :jane :friend ?f }
     UNION
       { :jane :loves ?l }
  }
```



```
-----------------
| f       | l       |
=================
| :bill |         |
| :bob  |         |
| :mary |         |
|       | :bill |
-----------------
```

```
---------
| f     |
=========
| :bill |
| :bob  |
| :mary |
---------
```

```
---------
| l     |
=========
| :bill |
---------
```

Alternative graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  {   ?f :likedBy :jane.
      ?l :lovedBy :jane. }
WHERE
  {    { :jane :friend ?f }
    UNION
       { :jane :loves ?l }
  }
```

Solution Sequence from Pattern Matching :

```
-----------------
| f       | l       |
=================
| :bill   |         |
| :bob    |         |
| :mary   |         |
|         | :bill   |
-----------------
```

Result Graph:

...

Alternative graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  {  ?f :likedBy :jane.
     ?l :lovedBy :jane. }
WHERE
  {   { :jane :friend ?f }
    UNION
      { :jane :loves ?l }
  }
```

Solution Sequence from Pattern Matching :

```
-----------------
| f       | l       |
=================
| :bill   |         |
| :bob    |         |
| :mary   |         |
|         | :bill   |
-----------------
```

Result Graph:

```
:bill   :likedBy   :jane
                          .
```

Alternative graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  {   ?f :likedBy :jane.
      ?l :lovedBy :jane. }
WHERE
  {    { :jane :friend ?f }
     UNION
       { :jane :loves ?l }
  }
```

Solution Sequence from Pattern Matching :

```
-----------------
| f       | l       |
=================
| :bill |         |
| :bob  |         |
| :mary |         |
|         | :bill |
-----------------
```

Result Graph:

```
:bill    :likedBy   :jane
                           .
:bob     :likedBy   :jane .
```

Alternative graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  {   ?f :likedBy :jane.
      ?l :lovedBy :jane. }
WHERE
  {    { :jane :friend ?f }
     UNION
       { :jane :loves ?l }
  }
```

Solution Sequence from Pattern Matching :

```
-----------------
| f       | l       |
=================
| :bill |         |
| :bob  |         |
| :mary |         |
|       | :bill |
-----------------
```

Result Graph:

```
:bill    :likedBy   :jane
                                .

:bob     :likedBy   :jane .

:mary    :likedBy   :jane .
```

Alternative graph patterns in CONSTRUCT queries

SPARQL Query:

```
CONSTRUCT
  {  ?f :likedBy :jane.
     ?l :lovedBy :jane. }
WHERE
  {   { :jane :friend ?f }
    UNION
      { :jane :loves ?l }
  }
```

Solution Sequence from Pattern Matching :

```
-----------------
| f       | l      |
=================
| :bill |          |
| :bob  |          |
| :mary |          |
|         | :bill |
-----------------
```

Result Graph:

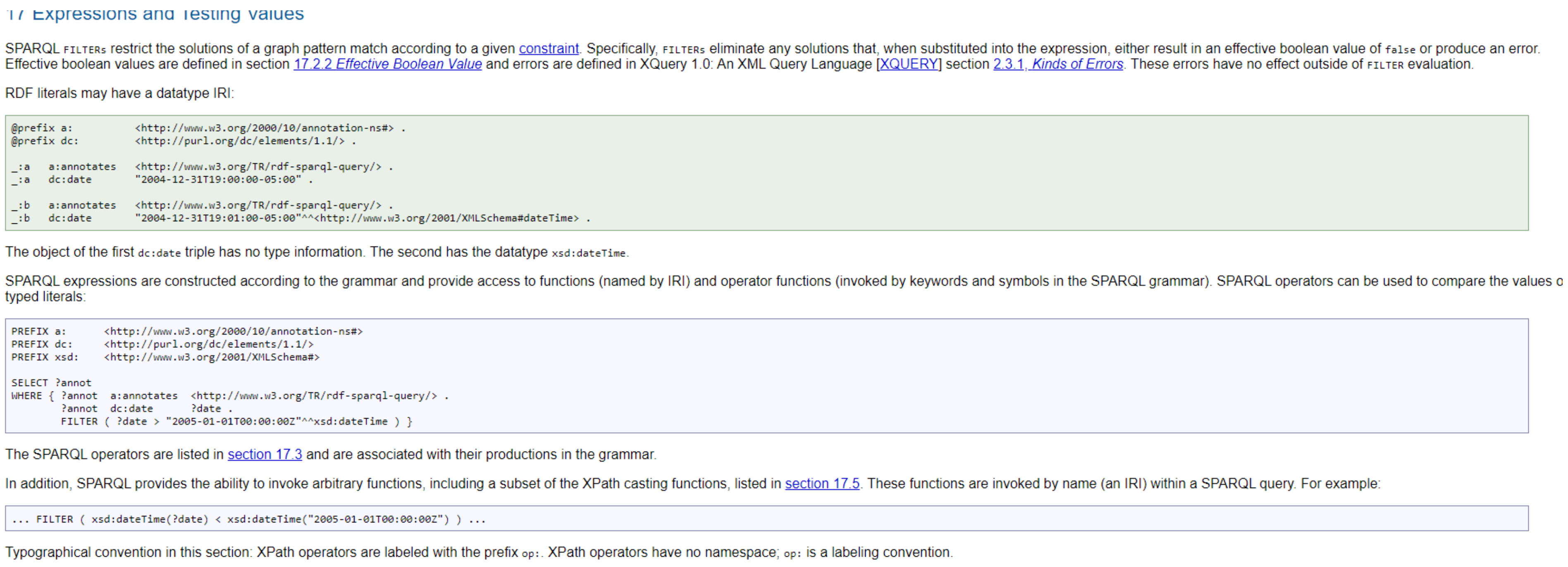
```
:bill    :likedBy  :jane ;
         :lovedBy   :jane .

:bob     :likedBy  :jane .

:mary    :likedBy  :jane .
```

FILTER

SELECTION CRITERIA (FILTER), SCOPE OF FILTERS

Specifying selection criteria (FILTER)

- Graph pattern matching produces a solution sequence, where each solution has a set of bindings of variables to RDF terms.
- SPARQL **FILTER**s restrict solutions to those for which the **filter expression** evaluates to TRUE.
- For functions and operators that can be used in FILTER expressions see: http://www.w3.org/TR/sparql11-query/#expressions



17 Expressions and Testing values

SPARQL FILTERs restrict the solutions of a graph pattern match according to a given constraint. Specifically, FILTERs eliminate any solutions that, when substituted into the expression, either result in an effective boolean value of `false` or produce an error. Effective boolean values are defined in section *17.2.2 Effective Boolean Value* and errors are defined in XQuery 1.0: An XML Query Language [XQUERY] section 2.3.1, *Kinds of Errors*. These errors have no effect outside of FILTER evaluation.

RDF literals may have a datatype IRI:

```
@prefix a:          <http://www.w3.org/2000/10/annotation-ns#> .
@prefix dc:         <http://purl.org/dc/elements/1.1/> .

_:a   a:annotates   <http://www.w3.org/TR/rdf-sparql-query/> .
_:a   dc:date       "2004-12-31T19:00:00-05:00" .

_:b   a:annotates   <http://www.w3.org/TR/rdf-sparql-query/> .
_:b   dc:date       "2004-12-31T19:01:00-05:00"^^<http://www.w3.org/2001/XMLSchema#dateTime> .
```

The object of the first `dc:date` triple has no type information. The second has the datatype `xsd:dateTime`.

SPARQL expressions are constructed according to the grammar and provide access to functions (named by IRI) and operator functions (invoked by keywords and symbols in the SPARQL grammar). SPARQL operators can be used to compare the values o
typed literals:

```
PREFIX a:      <http://www.w3.org/2000/10/annotation-ns#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>

SELECT ?annot
WHERE { ?annot  a:annotates  <http://www.w3.org/TR/rdf-sparql-query/> .
        ?annot  dc:date      ?date .
        FILTER ( ?date > "2005-01-01T00:00:00Z"^^xsd:dateTime ) }
```

The SPARQL operators are listed in section 17.3 and are associated with their productions in the grammar.

In addition, SPARQL provides the ability to invoke arbitrary functions, including a subset of the XPath casting functions, listed in section 17.5. These functions are invoked by name (an IRI) within a SPARQL query. For example:

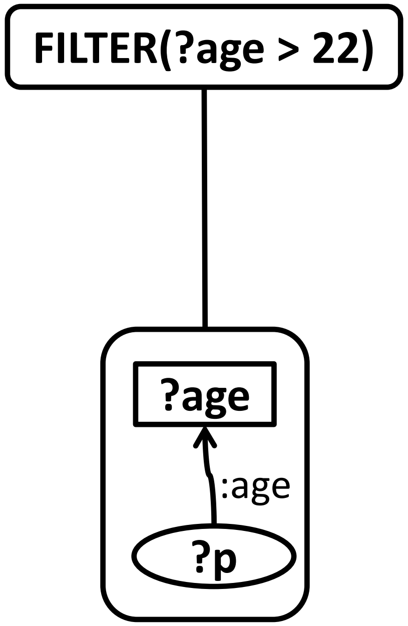
```
... FILTER ( xsd:dateTime(?date) < xsd:dateTime("2005-01-01T00:00:00Z") ) ...
```

Typographical convention in this section: XPath operators are labeled with the prefix `op:`. XPath operators have no namespace; `op:` is a labeling convention.

Specifying selection criteria (FILTER)

SPARQL Query:

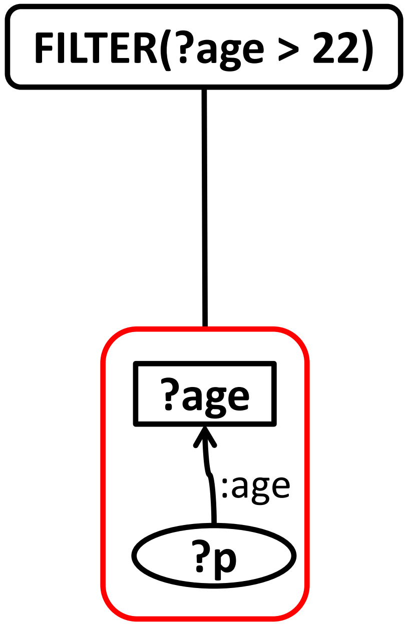
```
SELECT *
WHERE
  { ?p :age ?age
    FILTER(?age > 22)
  }
```



Specifying selection criteria (FILTER)

SPARQL Query:

```
SELECT *
WHERE
  { ?p :age ?age
    FILTER(?age > 22)
  }
```

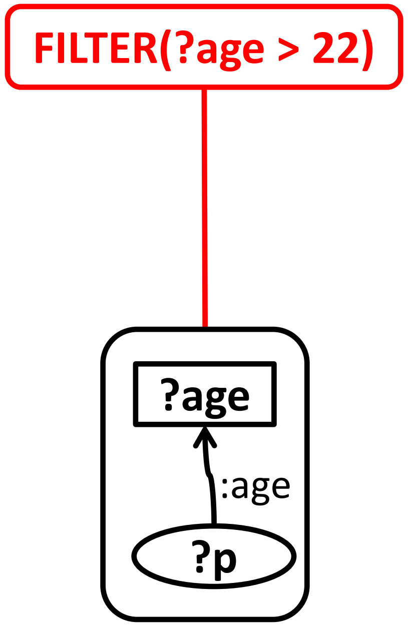


```
---------------
| p      | age |
===============
| :bob   | 26  |
| :mary  | 22  |
| :jane  | 22  |
---------------
```

Specifying selection criteria (FILTER)

SPARQL Query:

```
SELECT *
WHERE
  { ?p :age ?age
    FILTER(?age > 22)
  }
```



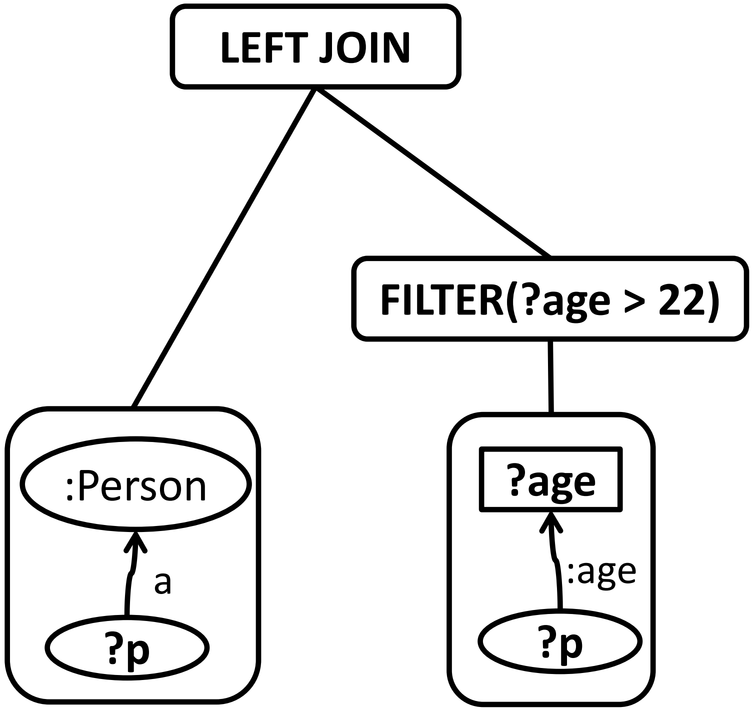
```
---------------
| p      | age |
===============
| :bob   | 26  |
---------------
```

```
---------------
| p      | age |
===============
| :bob   | 26  |
| :mary  | 22  |
| :jane  | 22  |
---------------
```

Scope of FILTERs (example 1)

SPARQL Query:

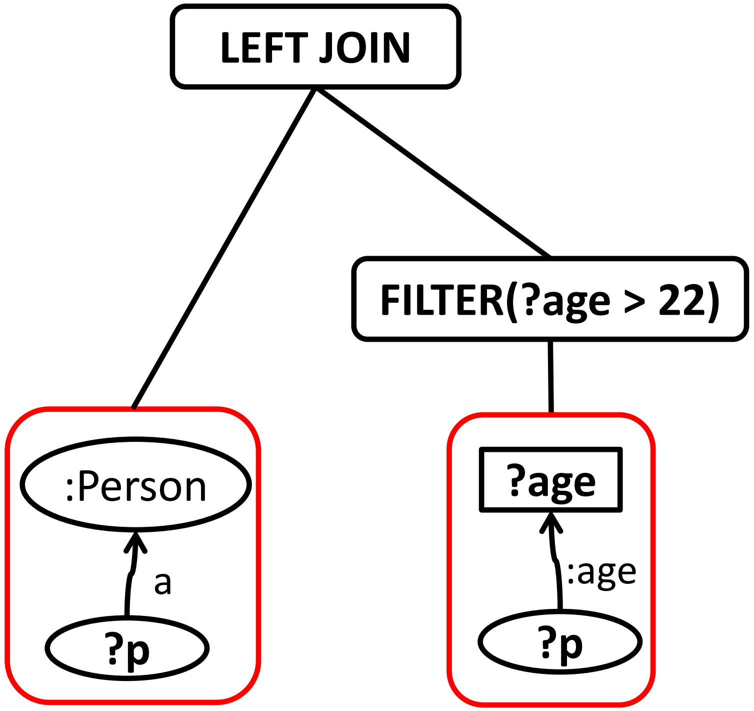
```
SELECT *
WHERE
  { ?p a :Person.
    OPTIONAL
      { ?p :age ?age
        FILTER(?age > 22) }
  }
```



Scope of FILTERs (example 1)

SPARQL Query:

```
SELECT *
WHERE
  { ?p a :Person.
    OPTIONAL
      { ?p :age ?age
        FILTER(?age > 22) }
  }
```



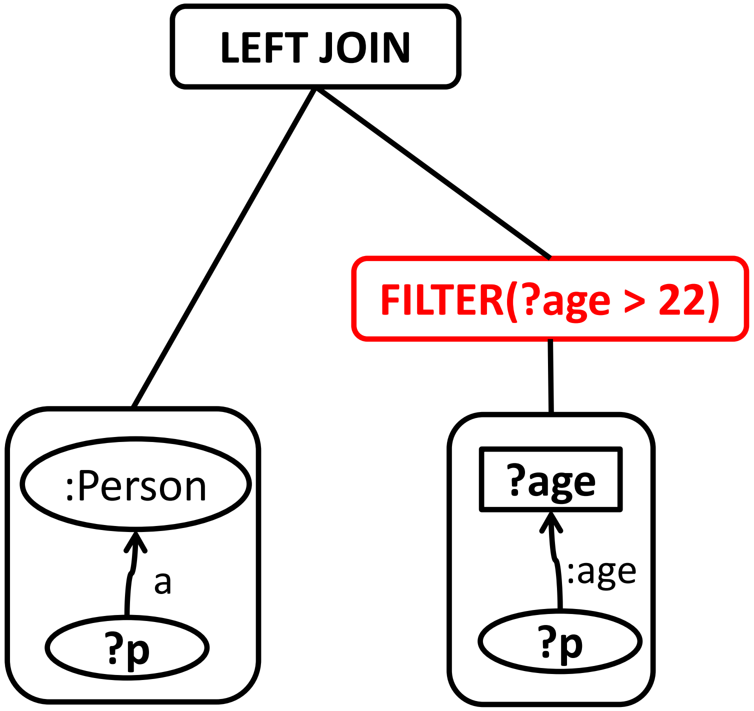
```
----------
| p      |
==========
| :bill  |
| :bob   |
| :mary  |
| :jane  |
----------
```

```
------------------
| p     | age  |
================
| :bob  | 26   |
| :mary | 22   |
| :jane | 22   |
------------------
```

Scope of FILTERs (example 1)

SPARQL Query:

```
SELECT *
WHERE
  { ?p a :Person.
    OPTIONAL
      { ?p :age ?age
        FILTER(?age > 22) }
  }
```



```
---------
| p     |
=========
| :bill |
| :bob  |
| :mary |
| :jane |
---------
```

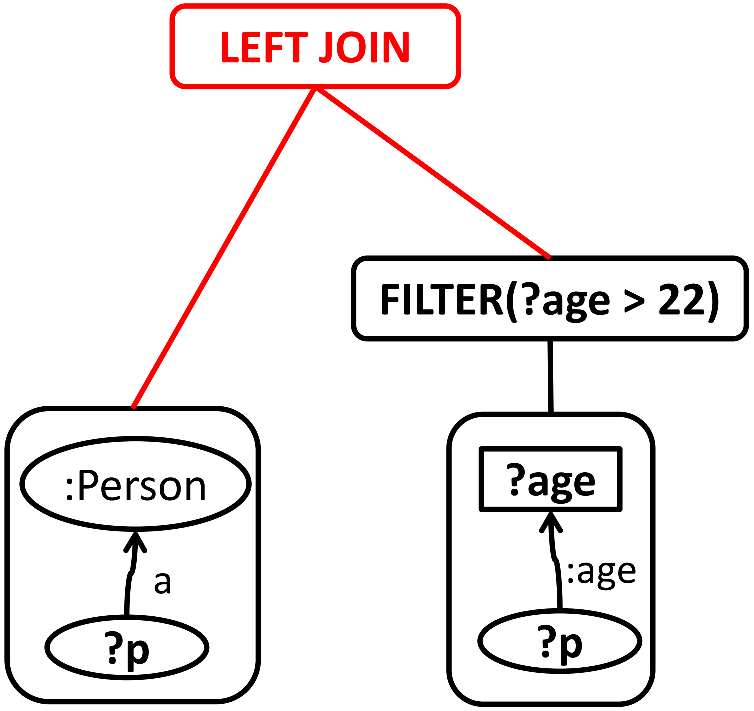
```
---------------
| p     | age |
===============
| :bob  | 26  |
---------------
```

```
---------------
| p     | age |
===============
| :bob  | 26  |
| :mary | 22  |
| :jane | 22  |
---------------
```

Scope of FILTERs (example 1)

SPARQL Query:

```
SELECT *
WHERE
  { ?p a :Person.
    OPTIONAL
      { ?p :age ?age
        FILTER(?age > 22) }
  }
```



```
---------------
| p      | age |
===============
| :bill  |     |
| :bob   | 26  |
| :mary  |     |
| :jane  |     |
---------------
```

```
---------
| p      |
=========
| :bill  |
| :bob   |
| :mary  |
| :jane  |
---------
```

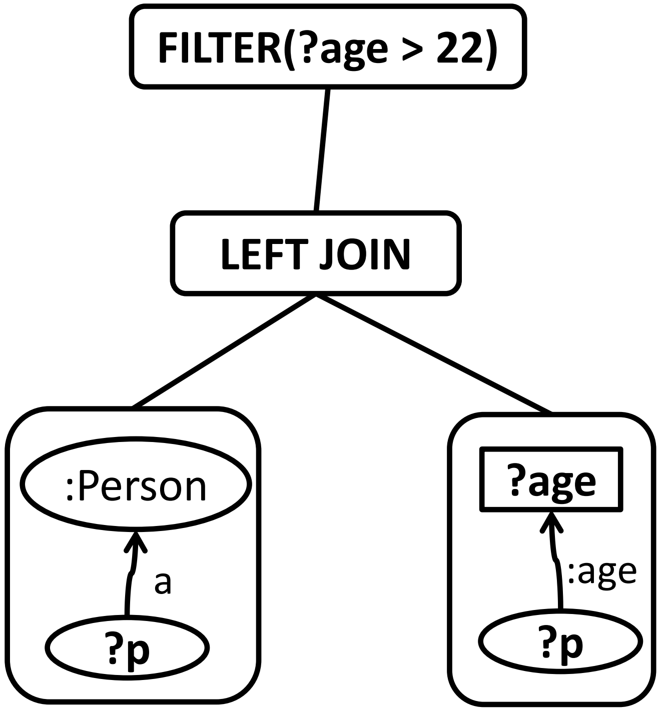
```
---------------
| p      | age |
===============
| :bob   | 26  |
---------------
```

```
---------------
| p      | age |
===============
| :bob   | 26  |
| :mary  | 22  |
| :jane  | 22  |
---------------
```

Scope of FILTERs (example 2)

SPARQL Query:

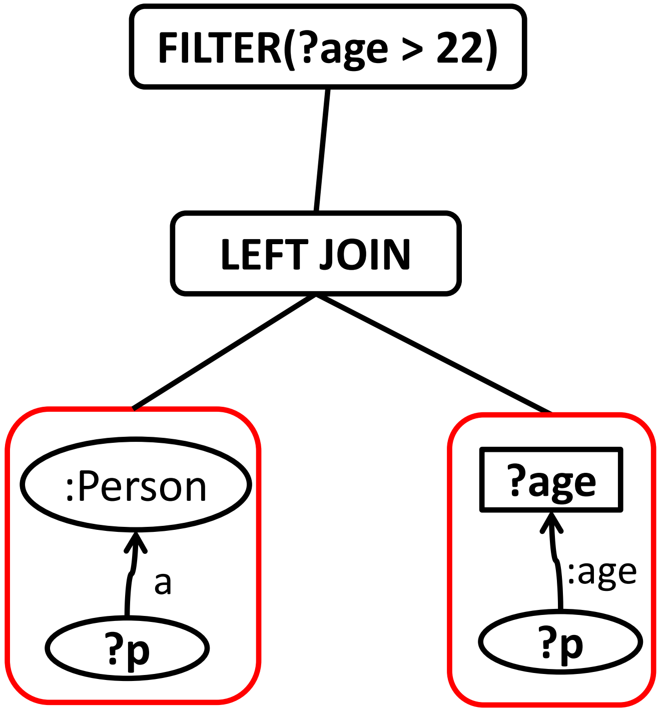
```
SELECT *
WHERE
  { ?p a :Person.
    OPTIONAL
      { ?p :age ?age }
    FILTER(?age > 22)
  }
```



Scope of FILTERs (example 2)

SPARQL Query:

```
SELECT *
WHERE
  { ?p a :Person.
    OPTIONAL
      { ?p :age ?age }
    FILTER(?age > 22)
  }
```



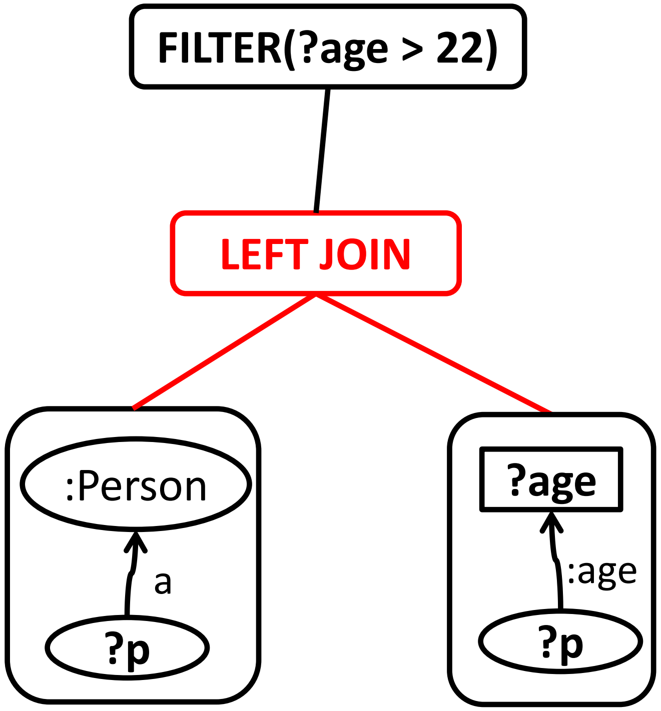
```
---------
| p     |
=========
| :bill |
| :bob  |
| :mary |
| :jane |
---------
```

```
---------------
| p     | age |
===============
| :bob  | 26  |
| :mary | 22  |
| :jane | 22  |
---------------
```

Scope of FILTERs (example 2)

SPARQL Query:

```
SELECT *
WHERE
  { ?p a :Person.
    OPTIONAL
      { ?p :age ?age }
    FILTER(?age > 22)
  }
```



```
---------------
| p     | age |
===============
| :bill |     |
| :bob  | 26  |
| :mary | 22  |
| :jane | 22  |
---------------
```

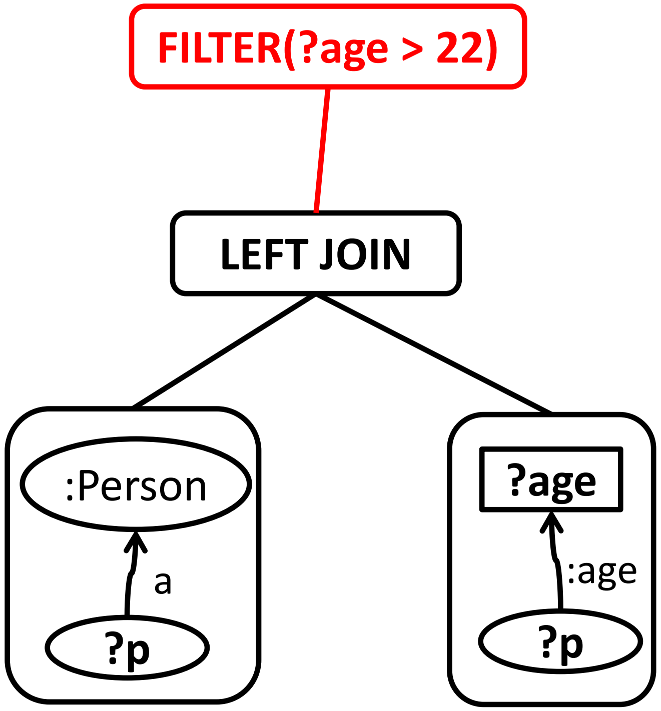
```
---------
| p     |
=========
| :bill |
| :bob  |
| :mary |
| :jane |
---------
```

```
---------------
| p     | age |
===============
| :bob  | 26  |
| :mary | 22  |
| :jane | 22  |
---------------
```

Scope of FILTERs (example 2)

SPARQL Query:

```
SELECT *
WHERE
  { ?p a :Person.
    OPTIONAL
      { ?p :age ?age }
    FILTER(?age > 22)
  }
```



```
---------------
| p     | age |
===============
| :bob  | 26  |
---------------
```

```
---------------
| p     | age |
===============
| :bill |     |
| :bob  | 26  |
| :mary | 22  |
| :jane | 22  |
---------------
```

```
---------
| p     |
=========
| :bill |
| :bob  |
| :mary |
| :jane |
---------
```

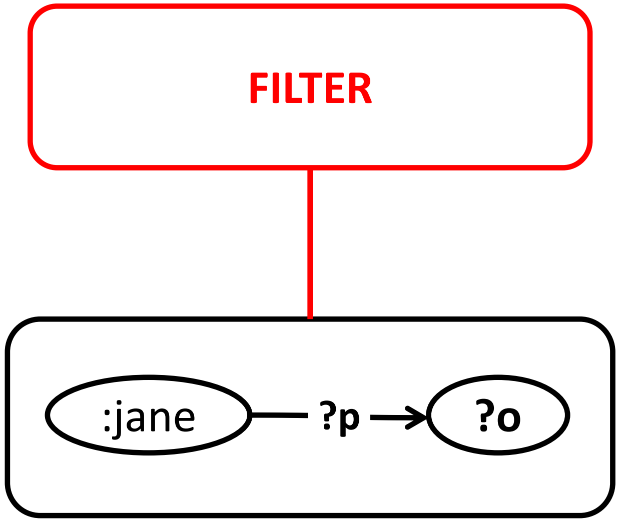
```
---------------
| p     | age |
===============
| :bob  | 26  |
| :mary | 22  |
| :jane | 22  |
---------------
```

Some more FILTERs

SPARQL Query:

```
SELECT *
WHERE
  { :jane ?p ?o.
    FILTER(?o IN (:mary, :bill))
  }
```

```
--------------------
| p        | o     |
====================
| :friend  | :mary |
| :loves   | :bill |
| :friend  | :bill |
--------------------
```



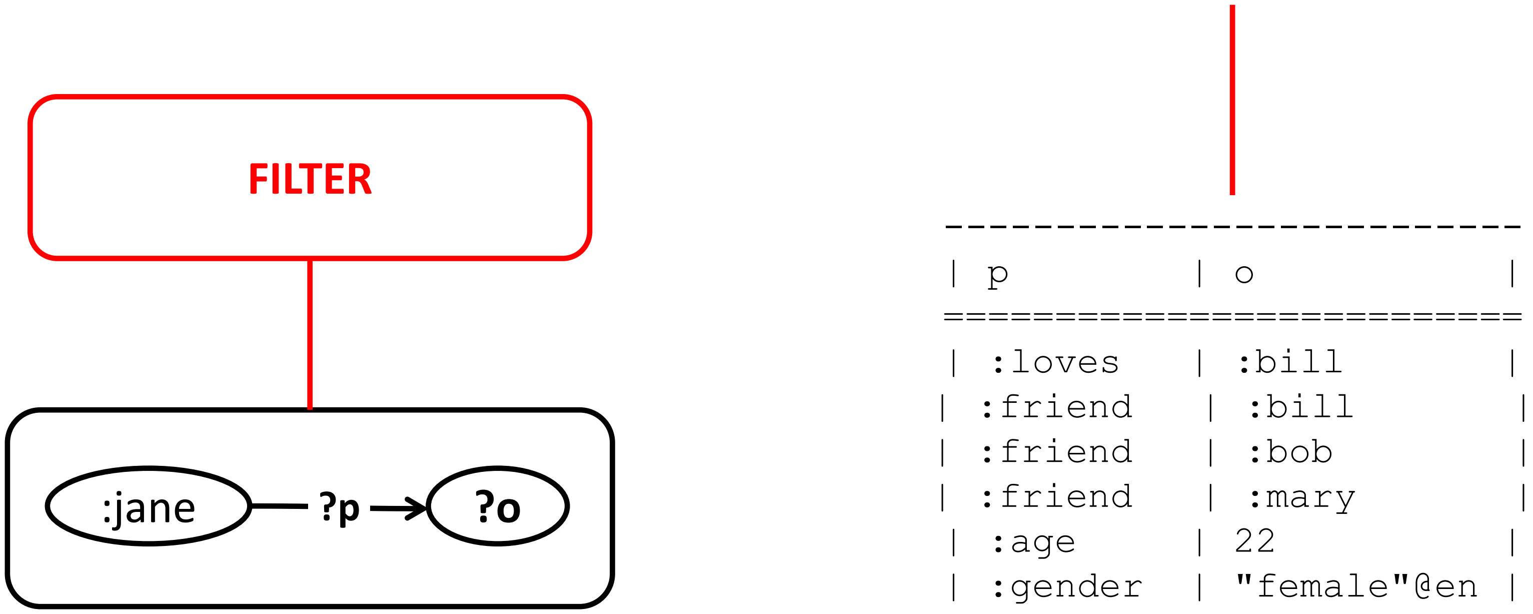
```
---------------------------
| p         | o           |
===========================
| :loves    | :bill       |
| :friend   | :bill       |
| :friend   | :bob        |
| :friend   | :mary       |
| :age      | 22          |
| :gender   | "female"@en |
| rdf:type  | :Person     |
---------------------------
```

Some more FILTERs

SPARQL Query:

```
SELECT *
WHERE
  { :jane ?p ?o.
    FILTER(?p NOT IN (:loves, :friend))
  }
```

```
---------------------------
| p          | o          |
===========================
| :age       | 22          |
| :gender    | "female"@en |
| rdf:type   | :Person     |
---------------------------
```



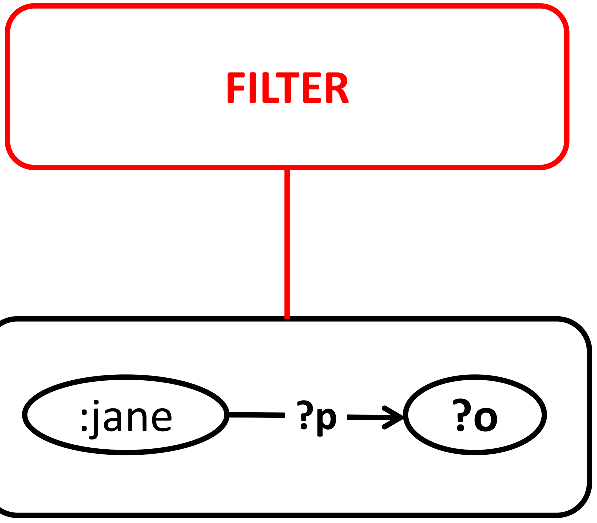
```
---------------------------
| p          | o           |
===========================
| :loves     | :bill       |
| :friend    | :bill       |
| :friend    | :bob        |
| :friend    | :mary       |
| :age       | 22          |
| :gender    | "female"@en |
| rdf:type   | :Person     |
---------------------------
```

Some more FILTERs

SPARQL Query:

```
SELECT *
WHERE
  { :jane ?p ?o.
    FILTER (isIRI(?o))
  }
```

```
-----------------------
| p          | o         |
=======================
| :loves     | :bill     |
| :friend    | :bill     |
| :friend    | :bob      |
| :friend    | :mary     |
| rdf:type   | :Person   |
-----------------------
```



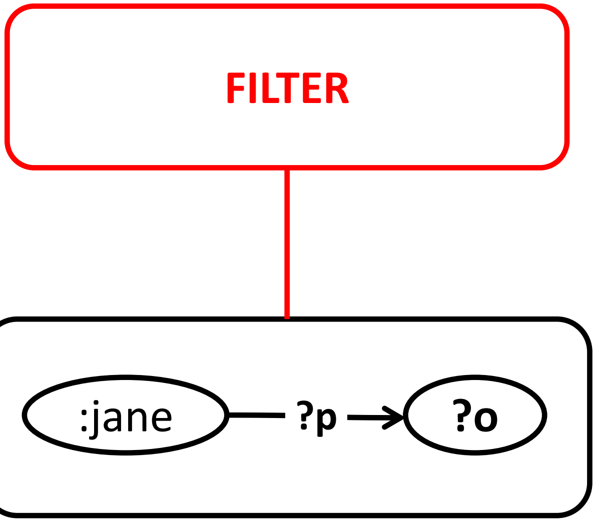
```
---------------------------
| p          | o            |
===========================
| :loves     | :bill        |
| :friend    | :bill        |
| :friend    | :bob         |
| :friend    | :mary        |
| :age       | 22           |
| :gender    | "female"@en  |
| rdf:type   | :Person      |
---------------------------
```

Some more FILTERs

SPARQL Query:

```
SELECT *
WHERE
  { :jane ?p ?o.
    FILTER (lang(?o) = "en")
  }
```

```
-------------------------
| p        | o            |
=========================
| :gender  | "female"@en  |
-------------------------
```



```
--------------------------
| p        | o            |
==========================
| :loves   | :bill        |
| :friend  | :bill        |
| :friend  | :bob         |
| :friend  | :mary        |
| :age     | 22           |
| :gender  | "female"@en  |
| rdf:type | :Person      |
--------------------------
```

Negation

NEGATION: FILTER NOT EXISTS, MINUS

Negation (FILTER NOT EXISTS, MINUS)

- The NOT EXISTS filter expression tests whether a graph pattern does not match the dataset, given the values of variables in the group graph pattern in which the filter occurs. It does not generate any additional bindings.
- The other style of negation provided in SPARQL is MINUS which evaluates both its arguments, then calculates solutions in the left-hand side that are not compatible with the solutions on the right-hand side.
- NOT EXISTS and MINUS represent two ways of thinking about negation, one based on testing whether a pattern exists in the data, given the bindings already determined by the query pattern, and one based on removing matches based on the evaluation of two patterns. In some cases they can produce different answers (see: http://www.w3.org/TR/sparql11-query/#neg-notexists-minus)

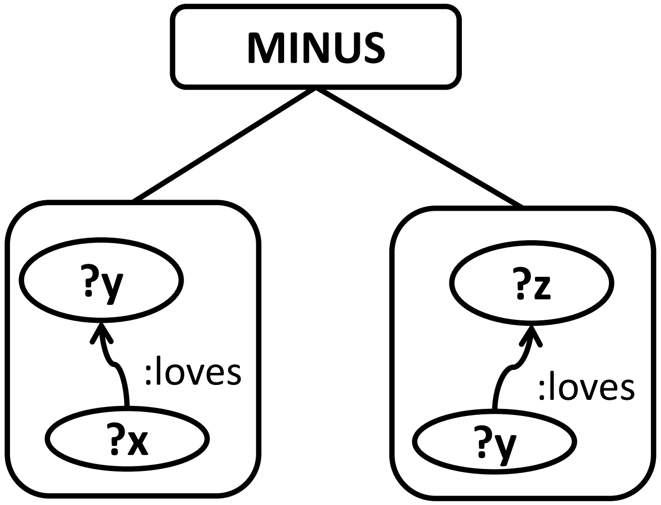
Negation (FILTER NOT EXISTS, MINUS)

SPARQL Query:

```
SELECT *
WHERE
  {   { ?x :loves ?y. }
    MINUS
      { ?y :loves ?z.}
  }
```

same result as

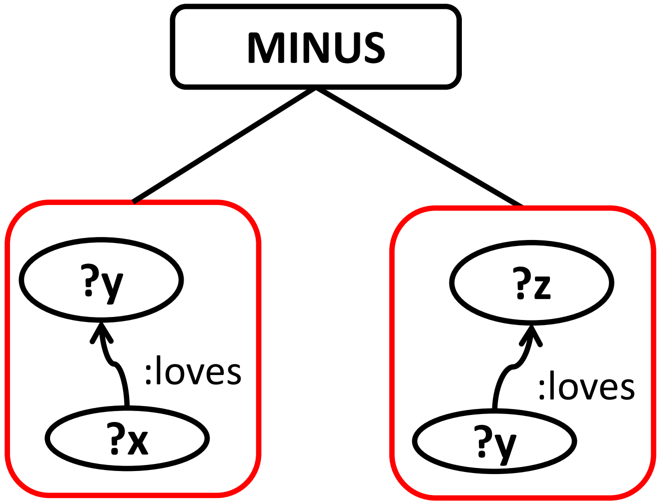
```
SELECT *
WHERE
  { ?x :loves ?y.
    FILTER NOT EXISTS {?y :loves ?z.}
  }
```



Negation (FILTER NOT EXISTS, MINUS)

SPARQL Query:

```
SELECT *
WHERE
  {    { ?x :loves ?y. }
    MINUS
       { ?y :loves ?z.}
  }
```



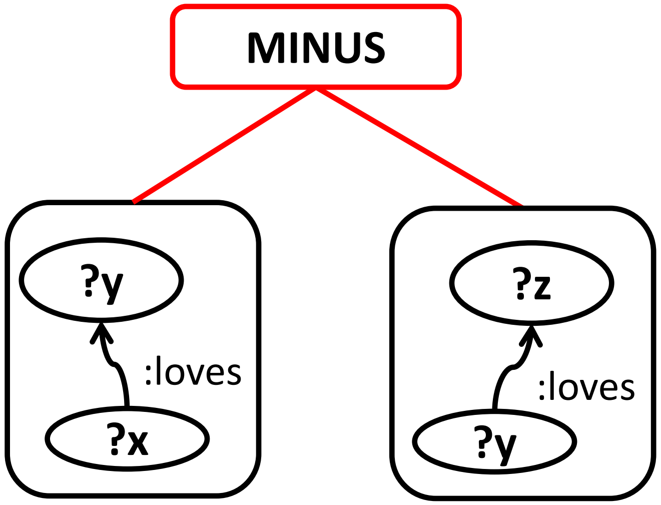
```
-----------------
| x      | y      |
=================
| :bob   | :jane |
| :mary | :bill |
| :jane | :bill |
-----------------
```

```
-----------------
| y      | z      |
=================
| :bob   | :jane |
| :mary | :bill |
| :jane | :bill |
-----------------
```

Negation (FILTER NOT EXISTS, MINUS)

SPARQL Query:

```
SELECT *
WHERE
  {    { ?x :loves ?y. }
     MINUS
       { ?y :loves ?z.}
  }
```



```
-----------------
| x       | y       |
=================
| :mary | :bill |
| :jane | :bill |
-----------------
```

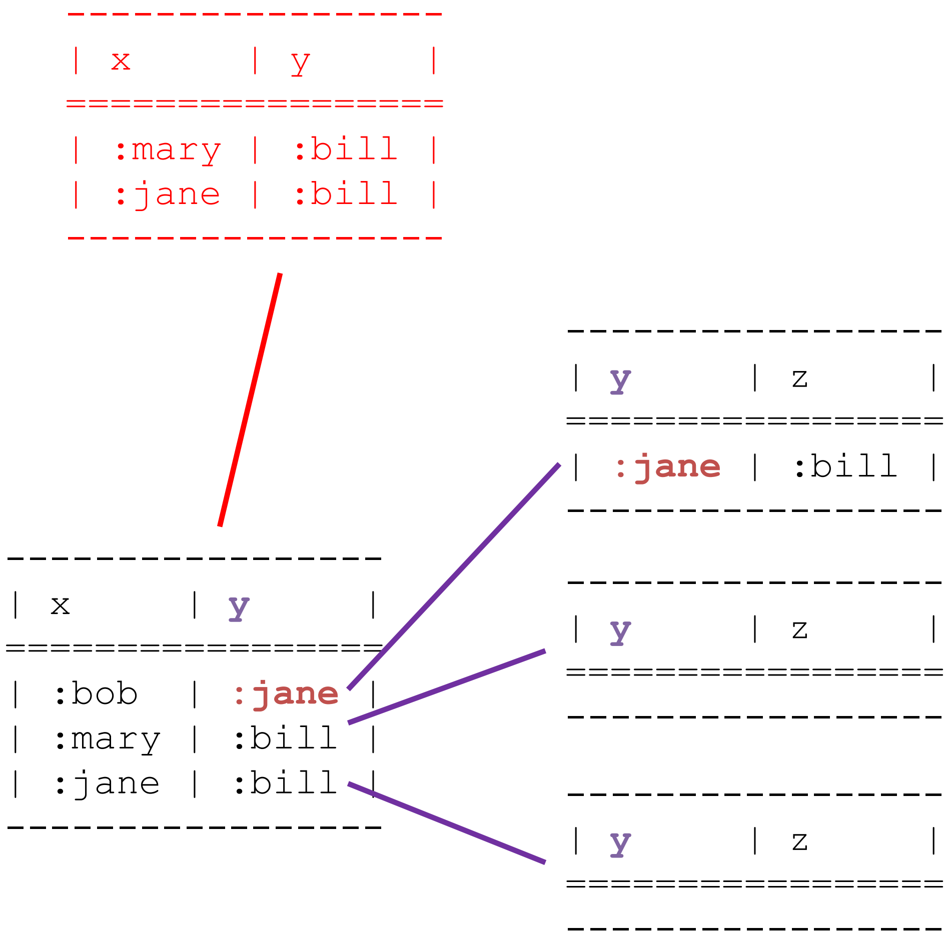
```
-----------------
| x       | y       |
=================
| :bob  | :jane |
| :mary | :bill |
| :jane | :bill |
-----------------
```

```
-----------------
| y       | z       |
=================
| :bob  | :jane |
| :mary | :bill |
| :jane | :bill |
-----------------
```

Negation (FILTER NOT EXISTS, MINUS)

same result as

```
SELECT *
WHERE
  { ?x :loves ?y.
    FILTER NOT EXISTS {?y :loves ?z.}
  }
```



```
-----------------
| x      | y     |
=================
| :mary | :bill |
| :jane | :bill |
-----------------
```

```
-----------------
| x      | y     |
=================
| :bob   | :jane |
| :mary | :bill |
| :jane | :bill |
-----------------
```

```
-----------------
| y      | z     |
=================
| :jane | :bill |
-----------------
```

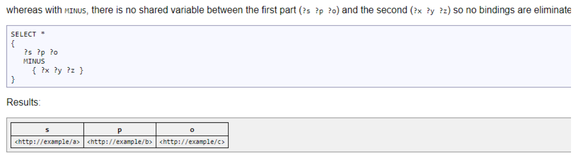
```
-----------------
| y      | z     |
=================
-----------------
```

```
-----------------
| y      | z     |
=================
-----------------
```

Differences between NOT EXISTS and MINUS

SPARQL Query:

```
SELECT *
WHERE
  {   { ?x :loves ?y. }
    MINUS
      { ?z :loves ?a.}
  }
```



whereas with MINUS, there is no shared variable between the first part (?s ?p ?o) and the second (?x ?y ?z) so no bindings are eliminate

```
SELECT *
{
   ?s ?p ?o
   MINUS
     { ?x ?y ?z }
}
```

Results:

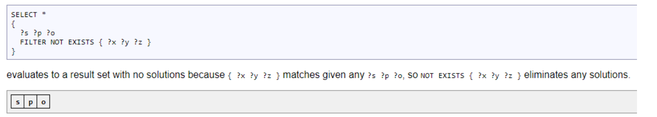
s	p	o
<http://example/a>	<http://example/b>	<http://example/c>

```
-----------------
| x     | y     |
=================
| :bob  | :jane |
| :mary | :bill |
| :jane | :bill |
-----------------
```

```
-----------------
| x     | y     |
=================
| :bob  | :jane |
| :mary | :bill |
| :jane | :bill |
-----------------
```

```
-----------------
| z     | a     |
=================
| :bob  | :jane |
| :mary | :bill |
| :jane | :bill |
-----------------
```

different result as



```
SELECT *
{
  ?s ?p ?o
  FILTER NOT EXISTS { ?x ?y ?z }
}
```

evaluates to a result set with no solutions because { ?x ?y ?z } matches given any ?s ?p ?o, so NOT EXISTS { ?x ?y ?z } eliminates any solutions.

s	p	o

```
SELECT *
WHERE
  { ?x :loves ?y.
    FILTER NOT EXISTS {?z :loves ?a.}
  }
```

```
---------
| x | y |
=========
---------
```

```
-----------------
| x     | y     |
=================
| :bob  | :jane |
| :mary | :bill |
| :jane | :bill |
-----------------
```

```
-----------------
| z     | a     |
=================
| :bob  | :jane |
| :mary | :bill |
| :jane | :bill |
-----------------
```

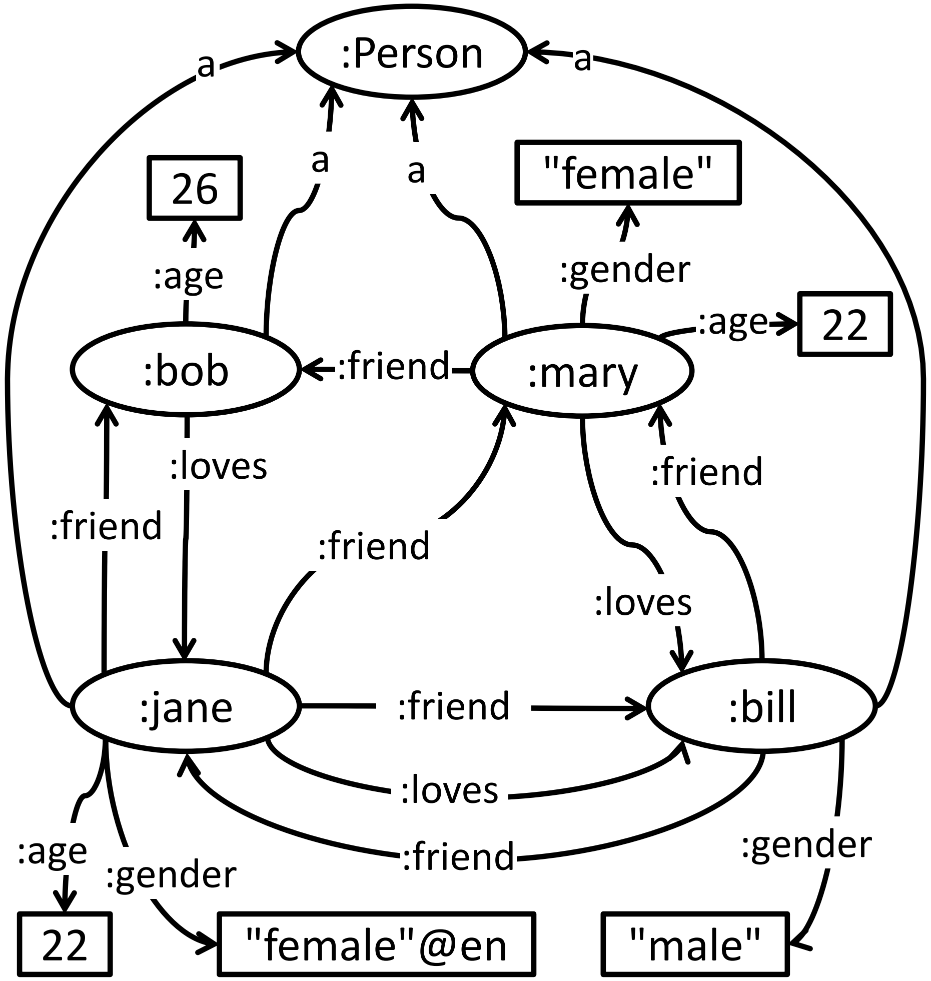
```
-----------------
| z     | a     |
=================
| :bob  | :jane |
| :mary | :bill |
| :jane | :bill |
-----------------
```

```
-----------------
| z     | a     |
=================
| :bob  | :jane |
| :mary | :bill |
| :jane | :bill |
-----------------
```

Running example: rather complicated social relationships in RDF

"Who has every person (except herself) as friend?"

```
@prefix : <http://example.org/> .

:jane  a :Person;
  :gender "female"@en; :age 22;
  :friend :mary, :bob, :bill;
  :loves :bill.

:mary  a :Person;
  :gender "female"; :age 22;
  :friend :bob;
  :loves :bill.

:bob  a :Person;
  :age 26;
  :loves :jane.

:bill  a :Person;
  :gender "male";
  :friend :mary, :jane.
```



"Relational Division" in SPARQL

"Who has every person (except herself) as friend?"

```
SELECT  ?p
WHERE
  { ?p rdf:type :Person
    FILTER NOT EXISTS {?person rdf:type :Person
      FILTER ( ?p != ?person )
      FILTER NOT EXISTS {?p :friend ?person}
    }
  }
```

"Relational Division" in SPARQL

"Who has every person (except herself) as friend?"

```
SELECT  ?p
WHERE
  { ?p rdf:type :Person
    FILTER NOT EXISTS {?person rdf:type :Person
      FILTER ( ?p != ?person )
      FILTER NOT EXISTS {?p :friend ?person}
    }
  }
```

```
---------
| p     |
=========
| :bill |
| :bob  |
| :mary |
| :jane |
---------
```

"Relational Division" in SPARQL

"Who has every person (except herself) as friend?"

```
SELECT  ?p
WHERE
  { ?p rdf:type :Person
    FILTER NOT EXISTS {?person rdf:type :Person
      FILTER ( ?p != ?person )
      FILTER NOT EXISTS {?p :friend ?person}
    }
  }
```

```
---------
| p     |
=========
---------
```

?p = :bill

```
----------
| p      |
==========
| :bill  |
| :bob   |
| :mary  |
| :jane  |
----------
```

```
-----------
| person  |
===========
| :bob    |
-----------
```

```
-----------
| person  |
===========
| :bill   |
| :mary   |
| :jane   |
-----------
```

```
-----------
| person  |
===========
| :bill   |
| :jane   |
-----------
```

```
-----------
| person  |
===========
-----------
```

"Relational Division" in SPARQL

"Who has every person (except herself) as friend?"

```
SELECT  ?p
WHERE
  { ?p rdf:type :Person
    FILTER NOT EXISTS {?person rdf:type :Person
      FILTER ( ?p != ?person )
      FILTER NOT EXISTS {?p :friend ?person}
    }
  }
```

```
---------
| p     |
=========
---------
```

```
----------
| p      |
==========
| :bill  |
| :bob   |
| :mary  |
| :jane  |
----------
```

?p = :bob

```
-----------
| person  |
===========
| :bob    |
-----------
```

```
-----------
| person  |
===========
| :bill   |
| :mary   |
| :jane   |
-----------
```

```
-----------
| person  |
===========
| :bill   |
| :jane   |
-----------
```

```
-----------
| person  |
===========
-----------
```

"Relational Division" in SPARQL

"Who has every person (except herself) as friend?"

```
SELECT  ?p
WHERE
  { ?p rdf:type :Person
    FILTER NOT EXISTS {?person rdf:type :Person
      FILTER ( ?p != ?person )
      FILTER NOT EXISTS {?p :friend ?person}
    }
  }
```

```
---------
| p     |
=========
---------
```

```
----------
| p      |
==========
| :bill  |
| :bob   |
| :mary  |
| :jane  |
----------
```

?p = :mary

```
-----------
| person  |
===========
| :bob    |
-----------
```

```
-----------
| person  |
===========
| :bill   |
| :mary   |
| :jane   |
-----------
```

```
-----------
| person  |
===========
| :bill   |
| :jane   |
-----------
```

```
-----------
| person  |
===========
-----------
```

"Relational Division" in SPARQL

I don't understand

"Who has every person (except herself) as friend?"

```
SELECT  ?p
WHERE
  { ?p rdf:type :Person
    FILTER NOT EXISTS {?person rdf:type :Person
      FILTER ( ?p != ?person )
      FILTER NOT EXISTS {?p :friend ?person}
    }
  }
```

```
---------
| p     |
=========
| :jane |
---------
```

```
----------
| p      |
==========
| :bill  |
| :bob   |
| :mary  |
| :jane  |
----------
```

```
-----------
| person  |
===========
| :bob    |
-----------
```

```
-----------
| person  |
===========
| :bill   |
| :mary   |
| :jane   |
-----------
```

```
-----------
| person  |
===========
| :bill   |
| :jane   |
-----------
```

?p = :jane

```
-----------
| person  |
===========
-----------
```

Assignment

ASSIGNMENT USING BIND, VALUES, AND IN SELECT CLAUSE

Assignment

- The **value of an expression** can be **added to a solution** mapping by binding a new variable to the value of the expression, which is an RDF term. The variable can then be used in the query and also can be returned in results.
- Three syntax forms allow this: the **BIND** keyword, expressions in the **SELECT** clause, and expressions in the **GROUP BY** clause. The assignment form is `(expression AS ?var)`
- If the evaluation of the expression produces an **error**, the variable remains **unbound** for that solution but the query evaluation **continues**.
- Data can also be directly included in a query using **VALUES** for inline data.

- The BIND form allows a value to be assigned to a variable from a basic graph pattern or property path expression. Use of BIND ends the preceding basic graph pattern.

Assignment in SELECT clause

Assignment in SELECT clause:

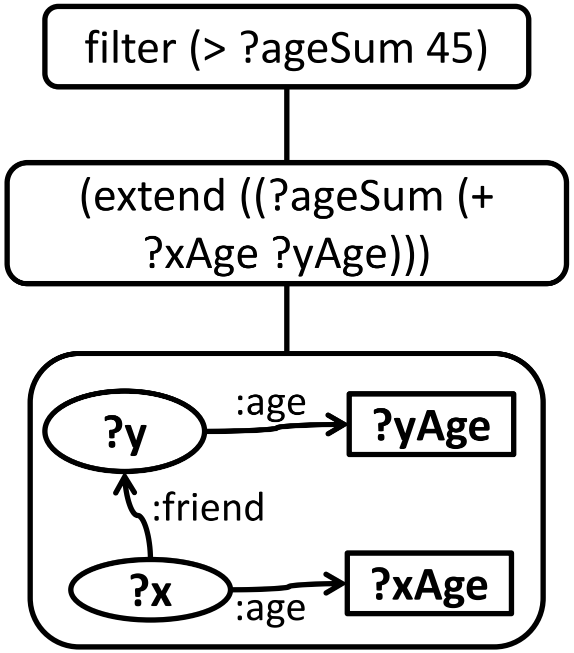
```
SELECT  ?x ?y ?xAge ?yAge (( ?xAge + ?yAge ) AS ?ageSum)
WHERE
  { ?x :friend ?y .
    ?x :age ?xAge .
    ?y :age ?yAge
    FILTER ( ( ?xAge + ?yAge ) > 45 )
  }
```

Result set:

```
---------------------------------------
| x       | y      | xAge | yAge | ageSum |
=======================================
| :mary | :bob | 22   | 26   | 48     |
| :jane | :bob | 22   | 26   | 48     |
---------------------------------------
```

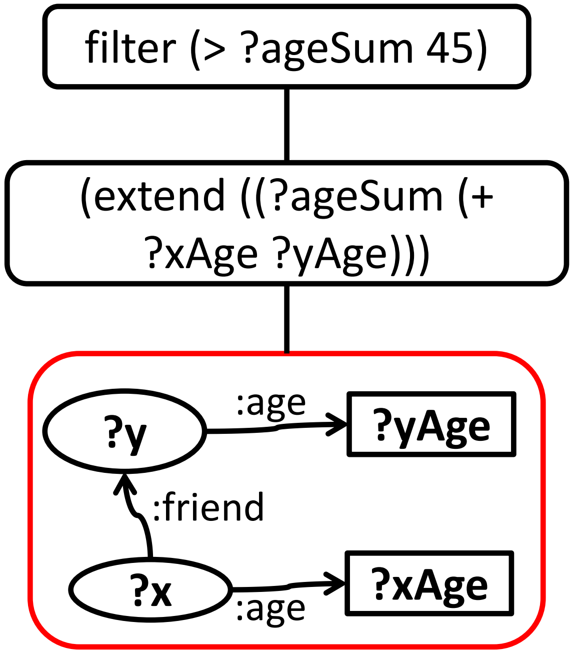
Assignment with BIND

```
SELECT  *
WHERE
  { ?x :friend ?y .
    ?x :age ?xAge .
    ?y :age ?yAge
    BIND(( ?xAge + ?yAge ) AS ?ageSum)
    FILTER ( ?ageSum > 45 )
  }
```



Assignment with BIND

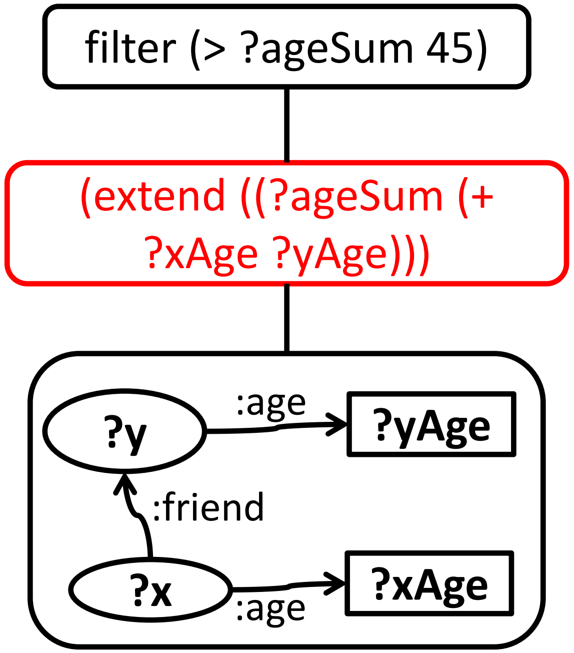
```
SELECT  *
WHERE
  { ?x :friend ?y .
    ?x :age ?xAge .
    ?y :age ?yAge
    BIND(( ?xAge + ?yAge ) AS ?ageSum)
    FILTER ( ?ageSum > 45 )
  }
```



```
--------------------------------
| x     | y     | xAge | yAge |
================================
| :mary | :bob  | 22   | 26   |
| :jane | :bob  | 22   | 26   |
| :jane | :mary | 22   | 22   |
--------------------------------
```

Assignment with BIND

```
SELECT  *
WHERE
  { ?x :friend ?y .
    ?x :age ?xAge .
    ?y :age ?yAge
    BIND(( ?xAge + ?yAge ) AS ?ageSum)
    FILTER ( ?ageSum > 45 )
  }
```

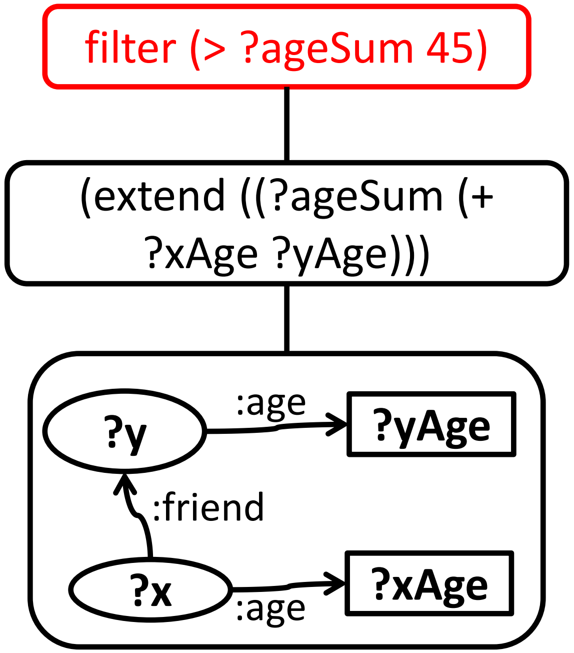


```
-------------------------------------------
| x     | y     | xAge | yAge | ageSum |
===========================================
| :mary | :bob  | 22   | 26   | 48     |
| :jane | :bob  | 22   | 26   | 48     |
| :jane | :mary | 22   | 22   | 44     |
-------------------------------------------
```

```
----------------------------------
| x     | y     | xAge | yAge |
==================================
| :mary | :bob  | 22   | 26   |
| :jane | :bob  | 22   | 26   |
| :jane | :mary | 22   | 22   |
----------------------------------
```

Assignment with BIND

```
SELECT  *
WHERE
   { ?x :friend ?y .
     ?x :age ?xAge .
     ?y :age ?yAge
     BIND(( ?xAge + ?yAge ) AS ?ageSum)
     FILTER ( ?ageSum > 45 )
   }
```



```
---------------------------------------------
| x      | y     | xAge | yAge | ageSum |
=============================================
| :mary | :bob | 22    | 26    | 48      |
| :jane | :bob | 22    | 26    | 48      |
---------------------------------------------
```

```
---------------------------------------------
| x      | y      | xAge | yAge | ageSum |
=============================================
| :mary | :bob  | 22    | 26    | 48      |
| :jane | :bob  | 22    | 26    | 48      |
| :jane | :mary | 22    | 22    | 44      |
---------------------------------------------
```

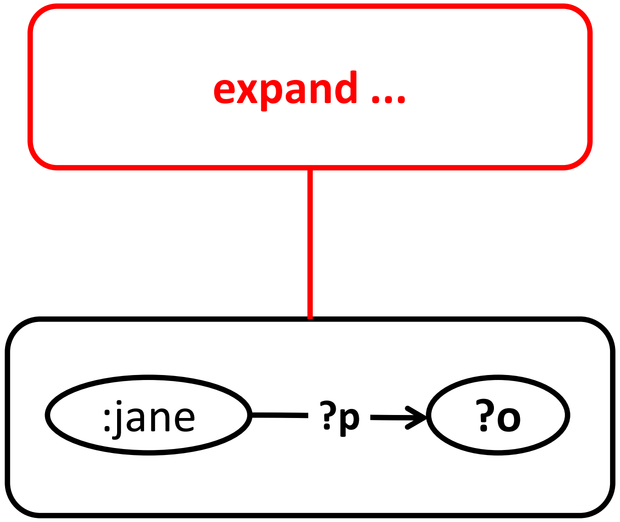
```
------------------------------------
| x      | y      | xAge | yAge |
====================================
| :mary | :bob  | 22    | 26    |
| :jane | :bob  | 22    | 26    |
| :jane | :mary | 22    | 22    |
------------------------------------
```

Some more Assignments

SPARQL Query:

```
SELECT *
WHERE
  { :jane ?p ?o.
    BIND(?o*2 AS ?x )
  }
```

```
-------------------------------------
| p          | o              | x  |
=====================================
| :loves     | :bill          |    |
| :friend    | :bill          |    |
| :friend    | :bob           |    |
| :friend    | :mary          |    |
| :age       | 22             | 44 |
| :gender    | "female"@en    |    |
| rdf:type   | :Person        |    |
-------------------------------------
```



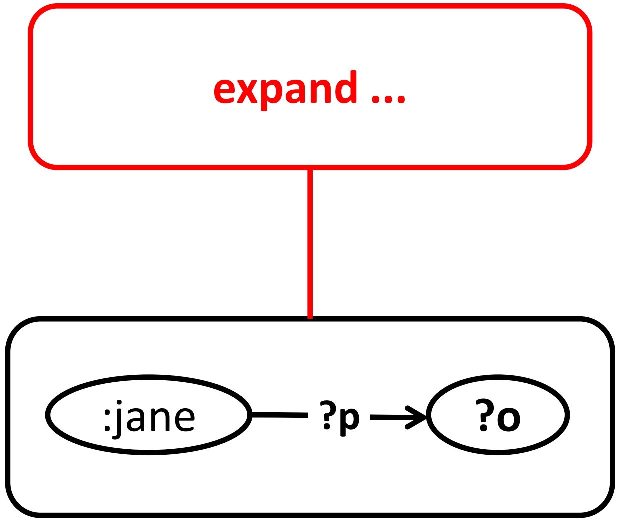
```
------------------------------
| p          | o              |
==============================
| :loves     | :bill          |
| :friend    | :bill          |
| :friend    | :bob           |
| :friend    | :mary          |
| :age       | 22             |
| :gender    | "female"@en    |
| rdf:type   | :Person        |
------------------------------
```

Some more Assignments

SPARQL Query:

```
SELECT *
WHERE
  { :jane ?p ?o.
    BIND(lang(?o) AS ?x )
  }
```

```
-------------------------------------
| p           | o             | x    |
=====================================
| :loves      | :bill         |      |
| :friend     | :bill         |      |
| :friend     | :bob          |      |
| :friend     | :mary         |      |
| :age        | 22            | ""   |
| :gender     | "female"@en   | "en" |
| rdf:type    | :Person       |      |
-------------------------------------
```



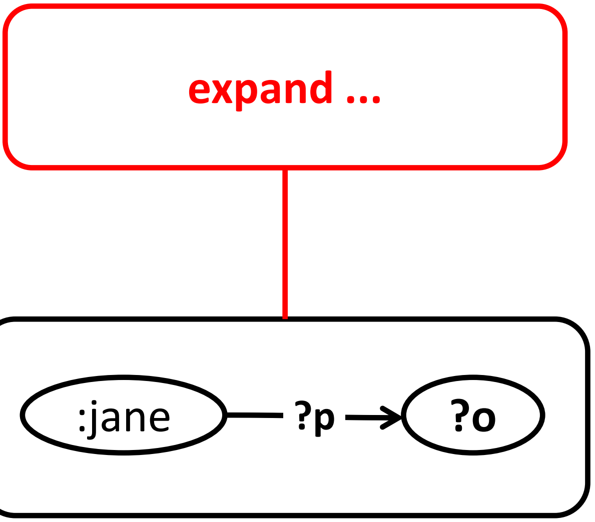
```
----------------------------
| p           | o             |
============================
| :loves      | :bill         |
| :friend     | :bill         |
| :friend     | :bob          |
| :friend     | :mary         |
| :age        | 22            |
| :gender     | "female"@en   |
| rdf:type    | :Person       |
----------------------------
```

Some more Assignments

SPARQL Query:

```
SELECT *
WHERE
  { :jane ?p ?o.
    BIND(isIRI(?o) AS ?x )
  }
```

```
-----------------------------------
| p          | o             | x     |
===================================
| :loves     | :bill         | true  |
| :friend    | :bill         | true  |
| :friend    | :bob          | true  |
| :friend    | :mary         | true  |
| :age       | 22            | false |
| :gender    | "female"@en   | false |
| rdf:type   | :Person       | true  |
-----------------------------------
```



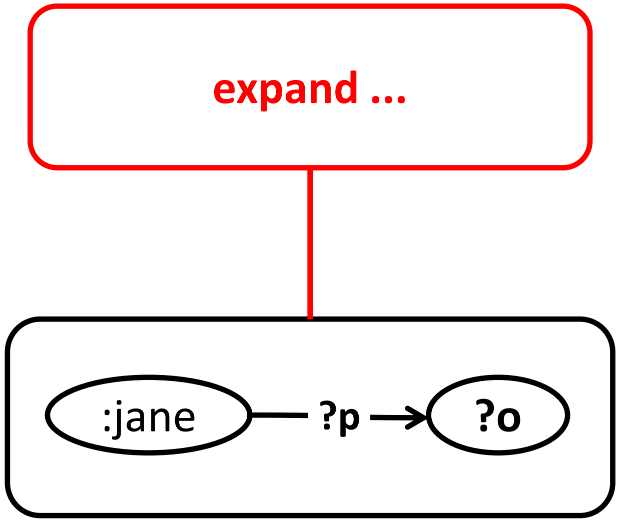
```
---------------------------
| p          | o             |
===========================
| :loves     | :bill         |
| :friend    | :bill         |
| :friend    | :bob          |
| :friend    | :mary         |
| :age       | 22            |
| :gender    | "female"@en   |
| rdf:type   | :Person       |
---------------------------
```

Some more Assignments

SPARQL Query:

```
SELECT *
WHERE
  { :jane ?p ?o.
    BIND(if(isNumeric(?o),
     "Numeric", "NonNumeric") AS ?x)
  }
```

```
-----------------------------------------
| p          | o             | x            |
=========================================
| :loves     | :bill         | "NonNumeric" |
| :friend    | :bill         | "NonNumeric" |
| :friend    | :bob          | "NonNumeric" |
| :friend    | :mary         | "NonNumeric" |
| :age       | 22            | "Numeric"    |
| :gender    | "female"@en   | "NonNumeric" |
| rdf:type   | :Person       | "NonNumeric" |
-----------------------------------------
```

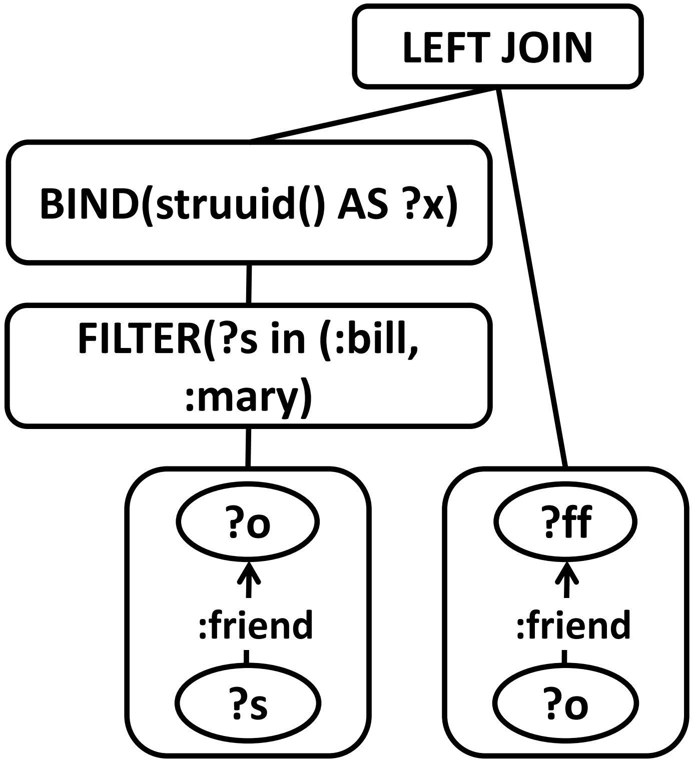


```
--------------------------
| p          | o             |
==========================
| :loves     | :bill         |
| :friend    | :bill         |
| :friend    | :bob          |
| :friend    | :mary         |
| :age       | 22            |
| :gender    | "female"@en   |
| rdf:type   | :Person       |
--------------------------
```

Reification of matched subgraphs

SPARQL Query:

```
SELECT  *
WHERE
  { { ?s :friend ?o
      FILTER(?s in (:bill, :mary))
    }
    BIND(struuid() AS ?x)
    OPTIONAL { ?o :friend ?ff}
  }
```



```
-----------------------------------------------------
| s     | o     | x                     | ff    |
=====================================================
| :bill | :jane | "9cd45c93-2179-..."  | :bill |
| :bill | :jane | "9cd45c93-2179-..."  | :bob  |
| :bill | :jane | "9cd45c93-2179-..."  | :mary |
| :bill | :mary | "bb1e45a4-fdfd-..."  | :bob  |
| :mary | :bob  | "734e6789-e980-..."  |       |
-----------------------------------------------------
```

```
--------------------------------------------
| s     | o     | x                      |
============================================
| :bill | :jane | "9cd45c93-2179-..."  |
| :bill | :mary | "bb1e45a4-fdfd-..."  |
| :mary | :bob  | "734e6789-e980-..."  |
--------------------------------------------
```

```
-------------------
| s     | o     |
===================
| :bill | :jane |
| :bill | :mary |
| :mary | :bob  |
-------------------
```

```
-------------------
| s     | o     |
===================
| :bill | :jane |
| :bill | :mary |
| :mary | :bob  |
| :jane | :bill |
| :jane | :bob  |
| :jane | :mary |
-------------------
```

```
-------------------
| o     | ff    |
===================
| :bill | :jane |
| :bill | :mary |
| :mary | :bob  |
| :jane | :bill |
| :jane | :bob  |
| :jane | :mary |
-------------------
```

Reifying matched subgraphs and using them as nodes in RDF graphs

```
CONSTRUCT
  { ?x a :friendship;
        :from ?s; :to ?o;
        :ffriend ?ff. }
WHERE
  { { ?s :friend ?o
      FILTER(?s
             in (:bill, :mary))
    }
    BIND(uuid() AS ?x)
    OPTIONAL { ?o :friend ?ff}
  }
```

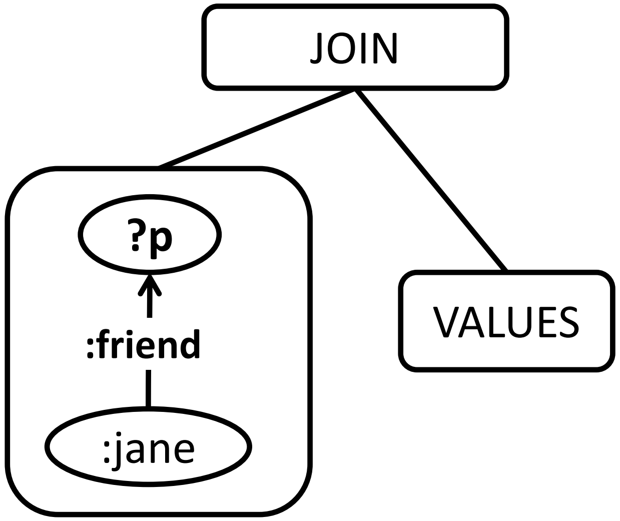
```
<urn:uuid:734e6...> a :friendship ;
        :from   :mary ;
        :to     :bob .

<urn:uuid:9cd45...>  a :friendship ;
        :ffriend  :mary , :bob , :bill ;
        :from     :bill ;
        :to       :jane .

<urn:uuid:3bb1e...>  a :friendship ;
        :ffriend  :bob ;
        :from     :bill ;
        :to       :mary .
```

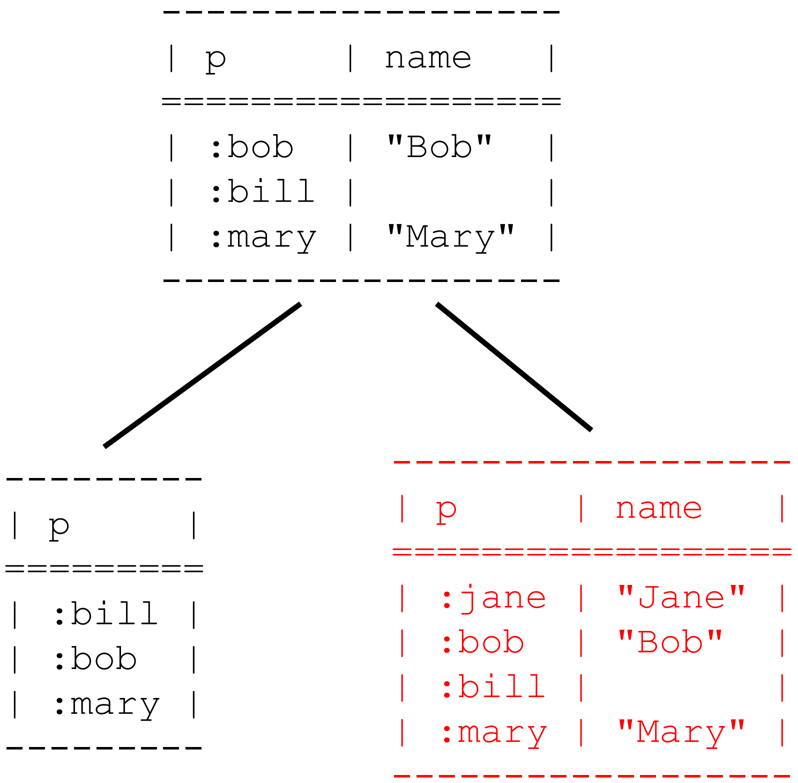
```
-----------------------------------------------
| s     | o     | x                     | ff    |
===============================================
| :bill | :jane | <urn:uuid:9cd45...> | :bill |
| :bill | :jane | <urn:uuid:9cd45...> | :bob  |
| :bill | :jane | <urn:uuid:9cd45...> | :mary |
| :bill | :mary | <urn:uuid:3bb1e...> | :bob  |
| :mary | :bob  | <urn:uuid:734e6...> |       |
-----------------------------------------------
```

VALUES: Providing inline Data

SPARQL Query:

```
SELECT *
WHERE
  { :jane :friend ?p .
    VALUES ( ?p ?name ) {
      ( :jane "Jane" )
      ( :bob   "Bob" )
      ( :bill UNDEF )
      ( :mary "Mary" )
    }
  }
```





```
-------------------
| p      | name   |
===================
| :bob   | "Bob"  |
| :bill  |        |
| :mary  | "Mary" |
-------------------
```

```
---------
| p     |
=========
| :bill |
| :bob  |
| :mary |
---------
```

```
-------------------
| p      | name   |
===================
| :jane  | "Jane" |
| :bob   | "Bob"  |
| :bill  |        |
| :mary  | "Mary" |
-------------------
```

Aggregates

AGGREGATES

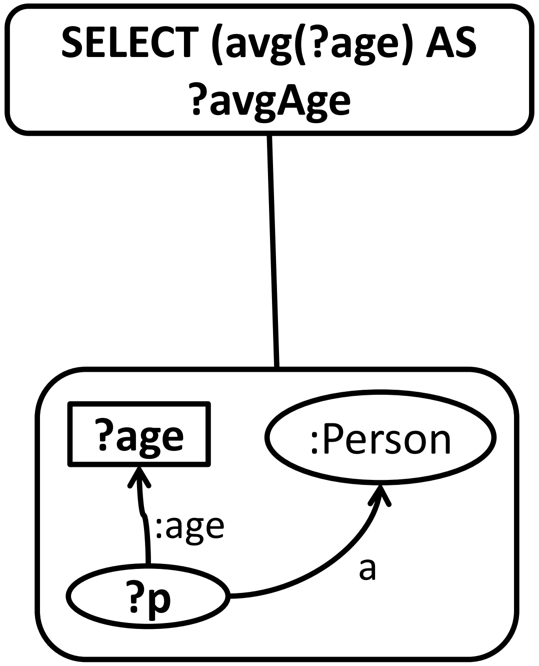
Aggregates

- Aggregates apply expressions over groups of solutions. By default a solution set consists of a single group, containing all solutions.
- Grouping may be specified using the GROUP BY syntax.
- Aggregates defined in version 1.1 of SPARQL are COUNT, SUM, MIN, MAX, AVG, GROUP_CONCAT, and SAMPLE.
- Aggregates are used where the querier wishes to see a result that is computed over a group of solutions, rather than a single solution. For example the maximum value that a particular variable takes, rather than each value individually.

- In order to only count distinct variable bindings use: `COUNT(DISTINCT ?var)`

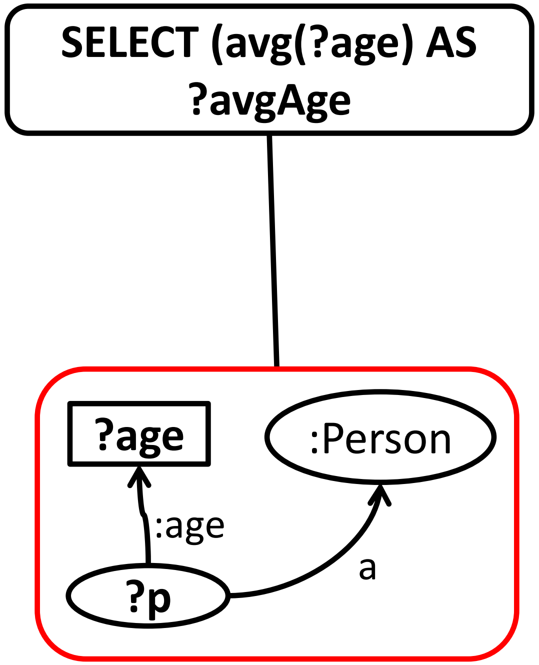
Aggregates (without GROUP BY)

```
SELECT  (avg(?age) AS ?avgAge)
WHERE
  { ?p rdf:type :Person .
    ?p :age ?age
  }
```



Aggregates (without GROUP BY)

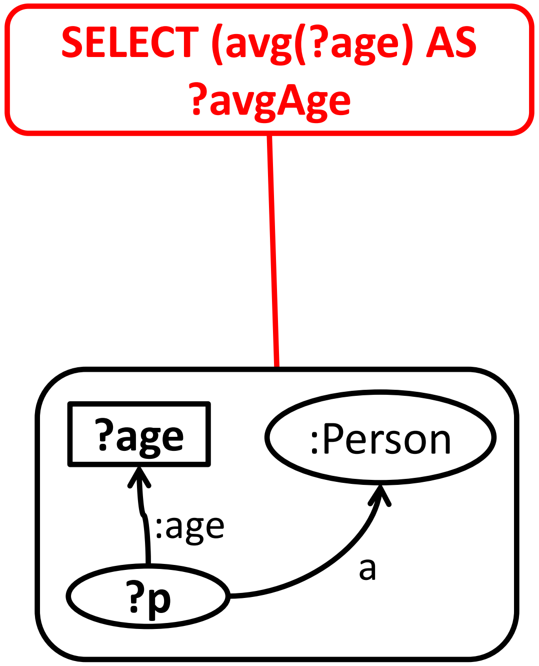
```
SELECT  (avg(?age) AS ?avgAge)
WHERE
  { ?p rdf:type :Person .
    ?p :age ?age
  }
```



```
---------------
| p     | age |
===============
| :bob  | 26  |
| :mary | 22  |
| :jane | 22  |
---------------
```

Aggregates (without GROUP BY)

```
SELECT  (avg(?age) AS ?avgAge)
WHERE
  { ?p rdf:type :Person .
    ?p :age ?age
  }
```



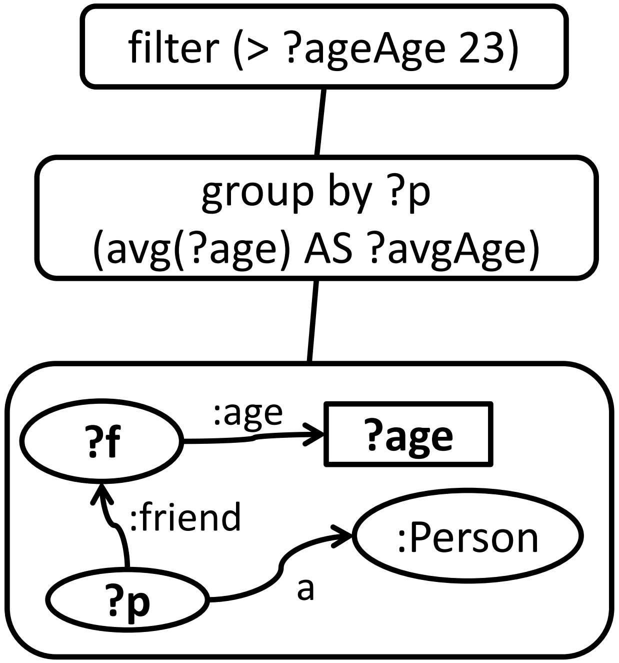
```
----------
| avgAge |
==========
| 23.333 |
----------
```

```
---------------
| p     | age |
===============
| :bob  | 26  |
| :mary | 22  |
| :jane | 22  |
---------------
```

Aggregates (with GROUP BY and HAVING)

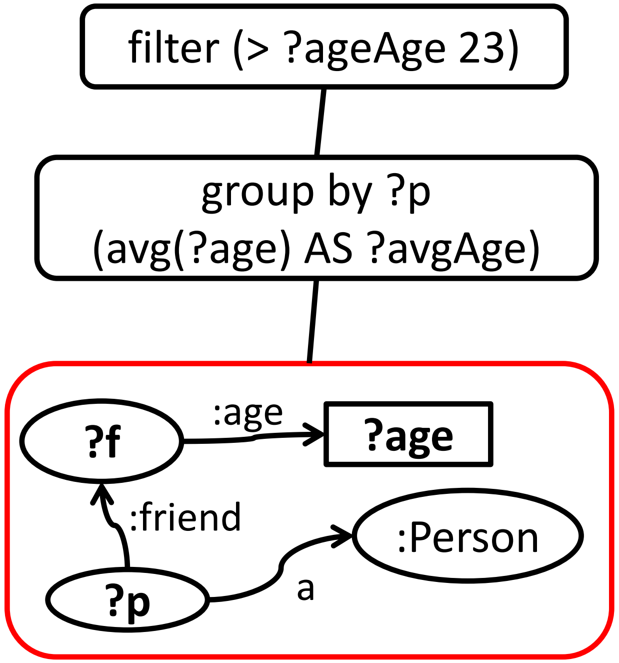
```
SELECT  ?p (avg(?age) AS ?avgAge)
WHERE
  { ?p rdf:type :Person .
    ?p :friend ?f .
    ?f :age ?age
  }
GROUP BY ?p
HAVING ( avg(?age) > 23 )
```

Finde Personen, deren Freunde im Durchschnitt älter als 23 sind und gib die jeweilige Person und das Durchschnittsalter ihrer Freunde aus.



Aggregates (with GROUP BY and HAVING)

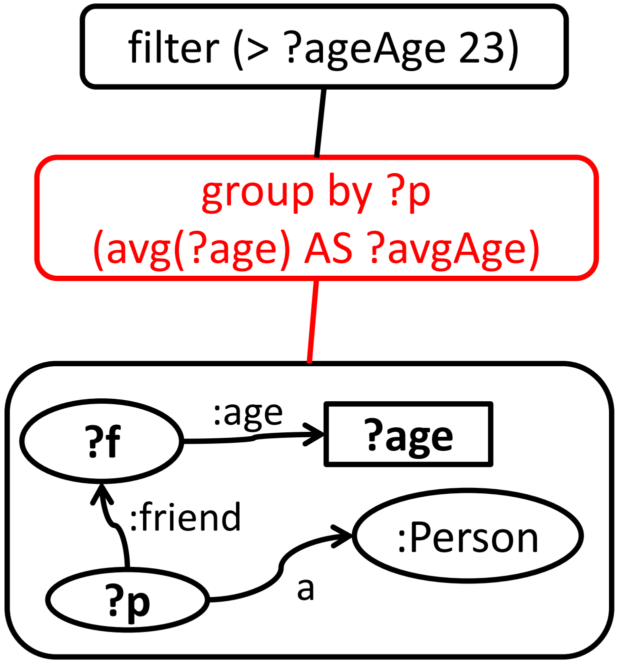
```
SELECT  ?p (avg(?age) AS ?avgAge)
WHERE
  { ?p rdf:type :Person .
    ?p :friend ?f .
    ?f :age ?age
  }
GROUP BY ?p
HAVING ( avg(?age) > 23 )
```



```
-----------------------
| p     | f     | age |
=======================
| :bill | :jane | 22  |
| :bill | :mary | 22  |
| :mary | :bob  | 26  |
| :jane | :bob  | 26  |
| :jane | :mary | 22  |
-----------------------
```

Aggregates (with GROUP BY and HAVING)

```
SELECT  ?p (avg(?age) AS ?avgAge)
WHERE
  { ?p rdf:type :Person .
    ?p :friend ?f .
    ?f :age ?age
  }
GROUP BY ?p
HAVING ( avg(?age) > 23 )
```

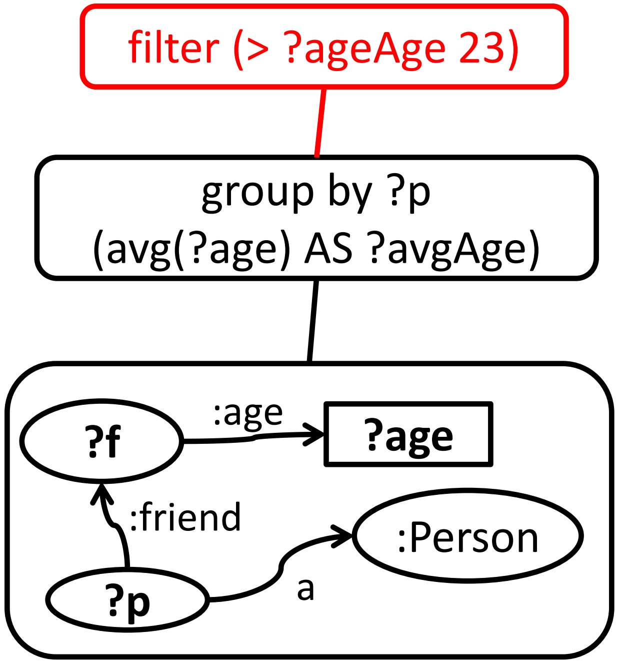


```
------------------
| p      | avgAge |
==================
| :jane  | 24.0   |
| :bill  | 22.0   |
| :mary  | 26.0   |
------------------
```

```
-----------------------
| p      | f      | age |
=======================
| :bill  | :jane  | 22  |
| :bill  | :mary  | 22  |
| :mary  | :bob   | 26  |
| :jane  | :bob   | 26  |
| :jane  | :mary  | 22  |
-----------------------
```

Aggregates (with GROUP BY and HAVING)

```
SELECT  ?p (avg(?age) AS ?avgAge)
WHERE
   { ?p rdf:type :Person .
     ?p :friend ?f .
     ?f :age ?age
   }
GROUP BY ?p
HAVING ( avg(?age) > 23 )
```



```
------------------
| p      | avgAge |
==================
| :jane | 24.0    |
| :mary | 26.0    |
------------------
```

```
------------------
| p      | avgAge |
==================
| :jane | 24.0    |
| :bill | 22.0    |
| :mary | 26.0    |
------------------
```

```
-----------------------
| p      | f      | age |
=======================
| :bill | :jane | 22   |
| :bill | :mary | 22   |
| :mary | :bob  | 26   |
| :jane | :bob  | 26   |
| :jane | :mary | 22   |
-----------------------
```

Subqueries

SUBQUERIES AND MULTISTEP AGGREGATION

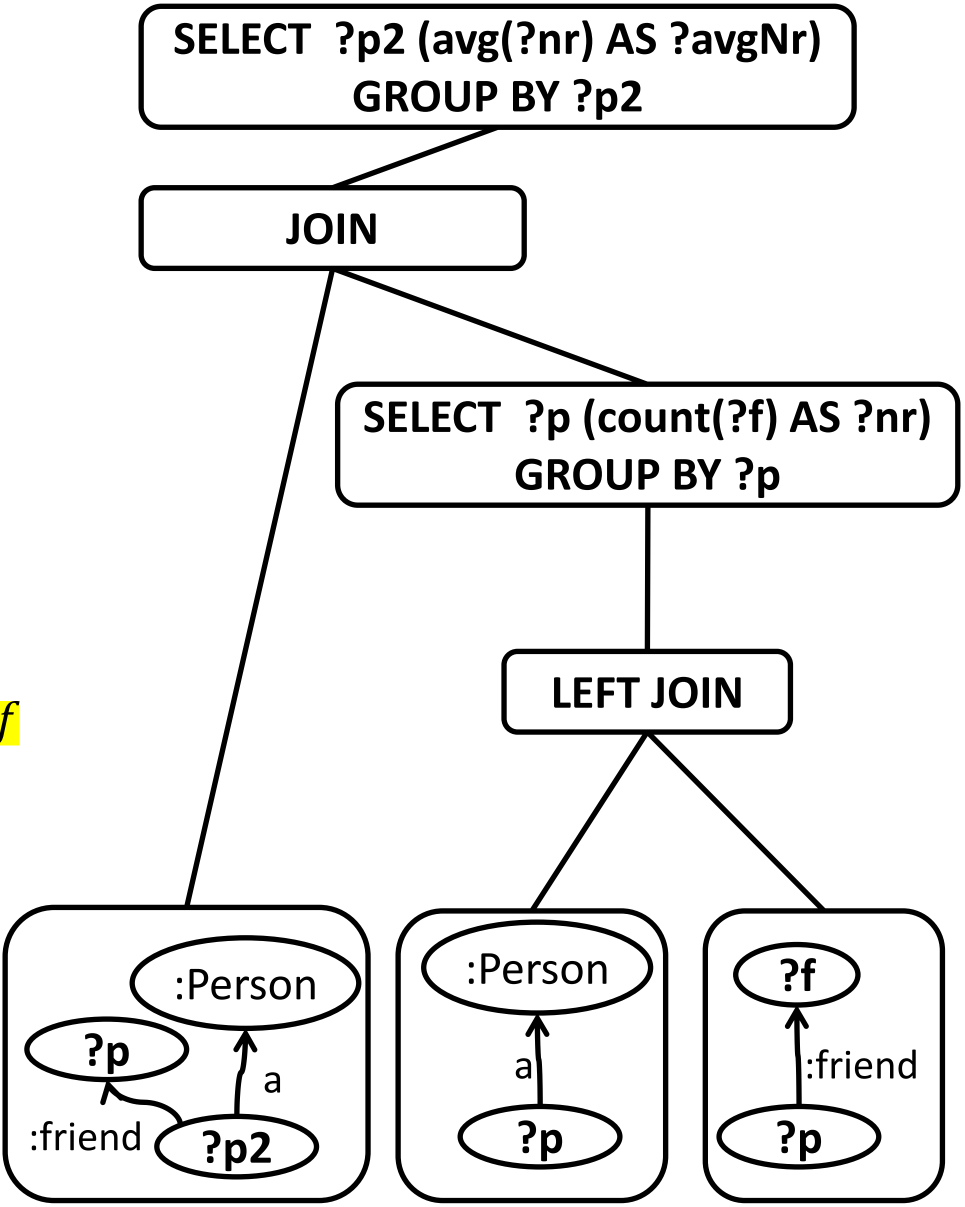
Subqueries

- Subqueries are a way to embed SPARQL queries within other queries, normally to achieve results which cannot otherwise be achieved, such as limiting the number of results from some sub-expression within the query.
- Due to the bottom-up nature of SPARQL query evaluation, the subqueries are evaluated logically first, and the results are projected up to the outer query.
- Note that only variables projected out of the subquery will be visible, or in scope, to the outer query.

Multistep Aggregation with Subqueries

```
SELECT  ?p2 (avg(?nr) AS ?avgNr)
WHERE
  { ?p2 rdf:type :Person .
    ?p2 :friend ?p
    { SELECT  ?p (count(?f) AS ?nr)
      WHERE
        { ?p rdf:type :Person
          OPTIONAL
            { ?p :friend ?f}
        }
      GROUP BY ?p
    }
  }
GROUP BY ?p2
```

How many friends do the friends of some person P2 have on average?



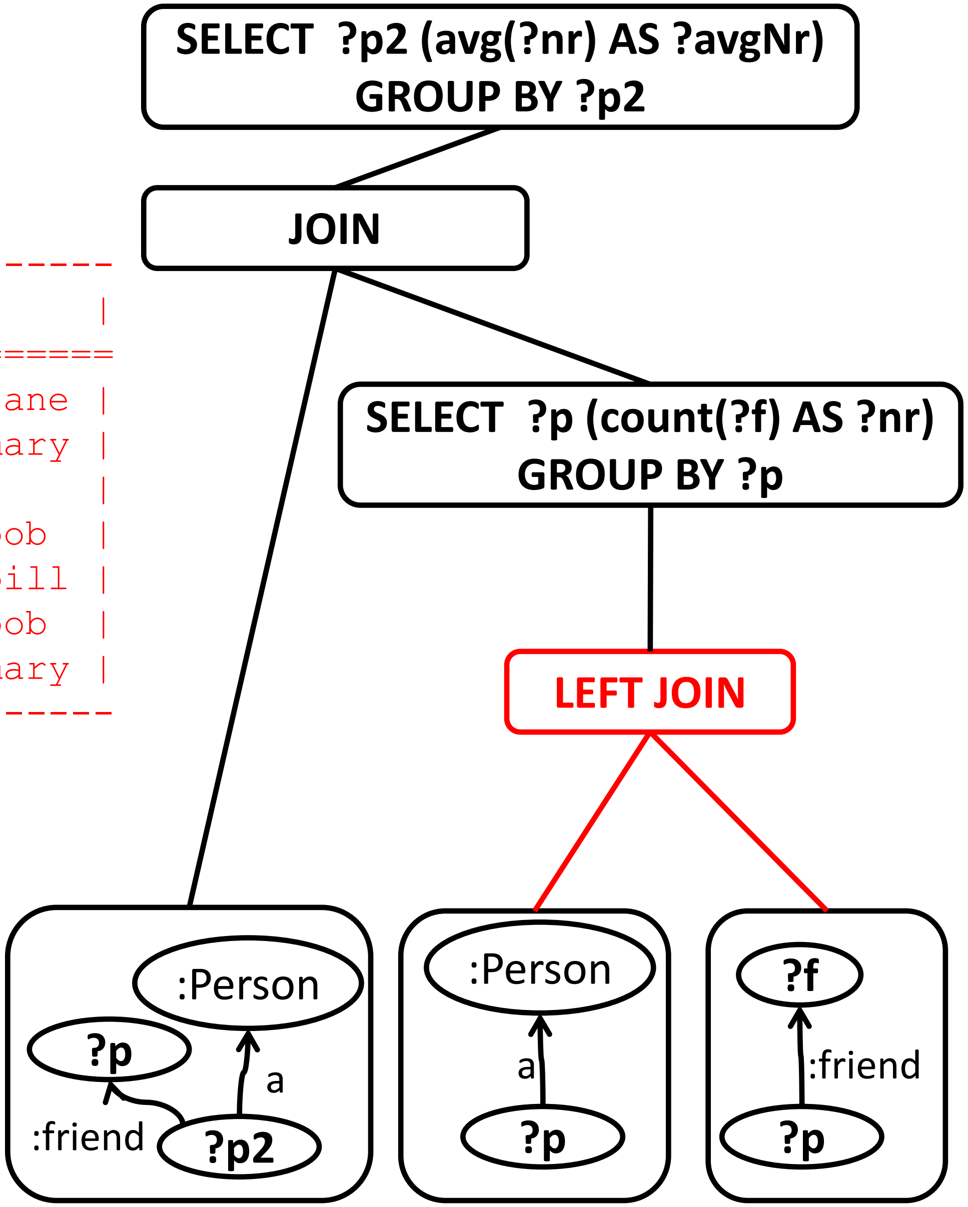
Multistep Aggregation with Subqueries

```
SELECT  ?p2 (avg(?nr) AS ?avgNr)
WHERE
  { ?p2 rdf:type :Person .
    ?p2 :friend ?p
    { SELECT  ?p (count(?f) AS ?nr)
      WHERE
        { ?p rdf:type :Person
          OPTIONAL
            { ?p :friend ?f}
        }
      GROUP BY ?p
    }
  }
GROUP BY ?p2
```

```
-----------------
| p      | f     |
=================
| :bill  | :jane |
| :bill  | :mary |
| :bob   |       |
| :mary  | :bob  |
| :jane  | :bill |
| :jane  | :bob  |
| :jane  | :mary |
-----------------
```

```
---------
| p      |
=========
| :bill  |
| :bob   |
| :mary  |
| :jane  |
---------
```

```
-----------------
| p      | f     |
=================
| :bill  | :jane |
| :bill  | :mary |
| :mary  | :bob  |
| :jane  | :bill |
| :jane  | :bob  |
| :jane  | :mary |
-----------------
```

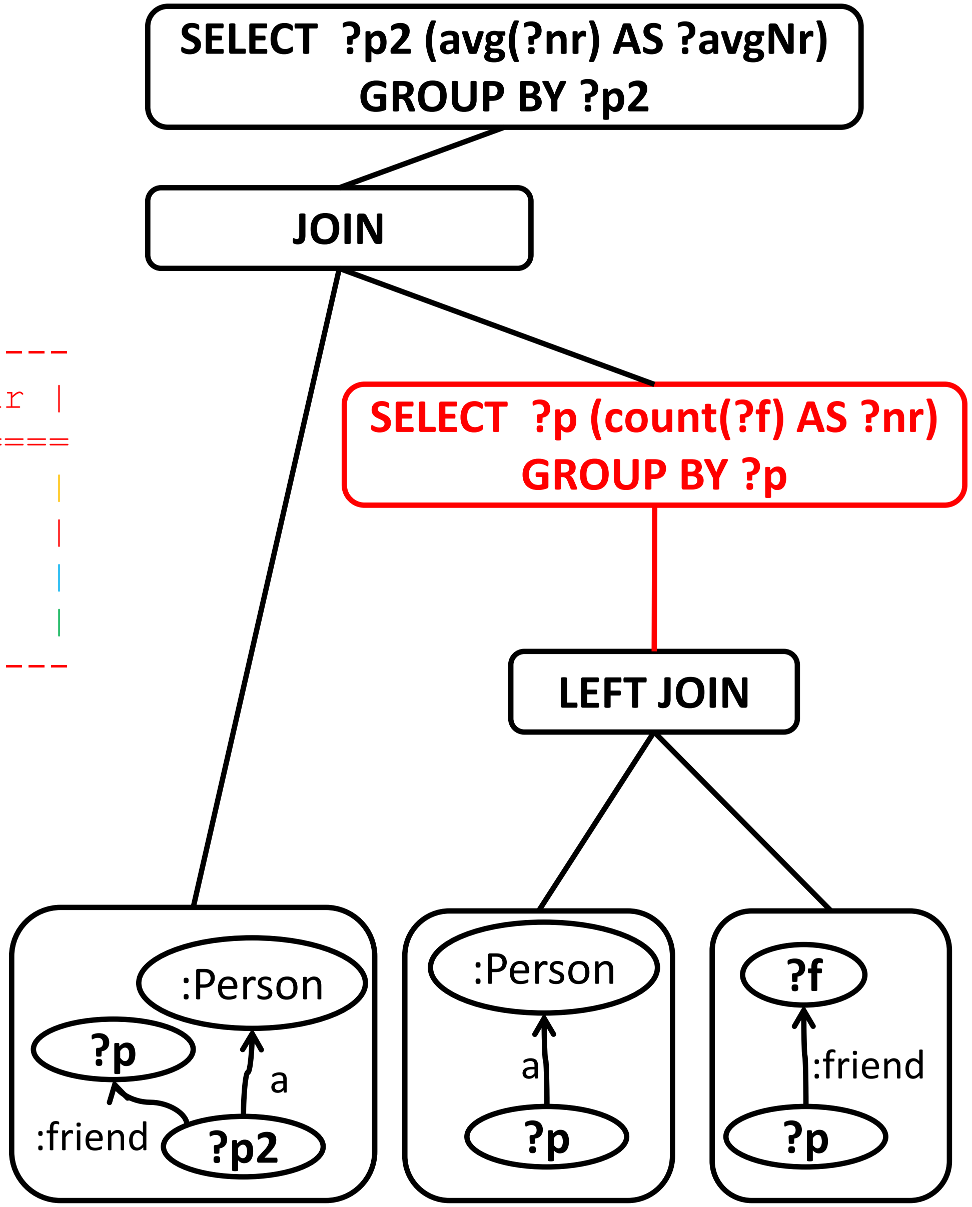


Multistep Aggregation with Subqueries

```
SELECT  ?p2 (avg(?nr) AS ?avgNr)
WHERE
  { ?p2 rdf:type :Person .
    ?p2 :friend ?p
    { SELECT  ?p (count(?f) AS ?nr)
      WHERE
        { ?p rdf:type :Person
          OPTIONAL
            { ?p :friend ?f}
        }
      GROUP BY ?p
    }
  }
GROUP BY ?p2
```

```
--------------
| p      | nr |
==============
| :jane  | 3  |
| :bill  | 2  |
| :mary  | 1  |
| :bob   | 0  |
--------------
```

```
-----------------
| p      | f      |
=================
| :bill  | :jane  |
| :bill  | :mary  |
| :bob   |        |
| :mary  | :bob   |
| :jane  | :bill  |
| :jane  | :bob   |
| :jane  | :mary  |
-----------------
```



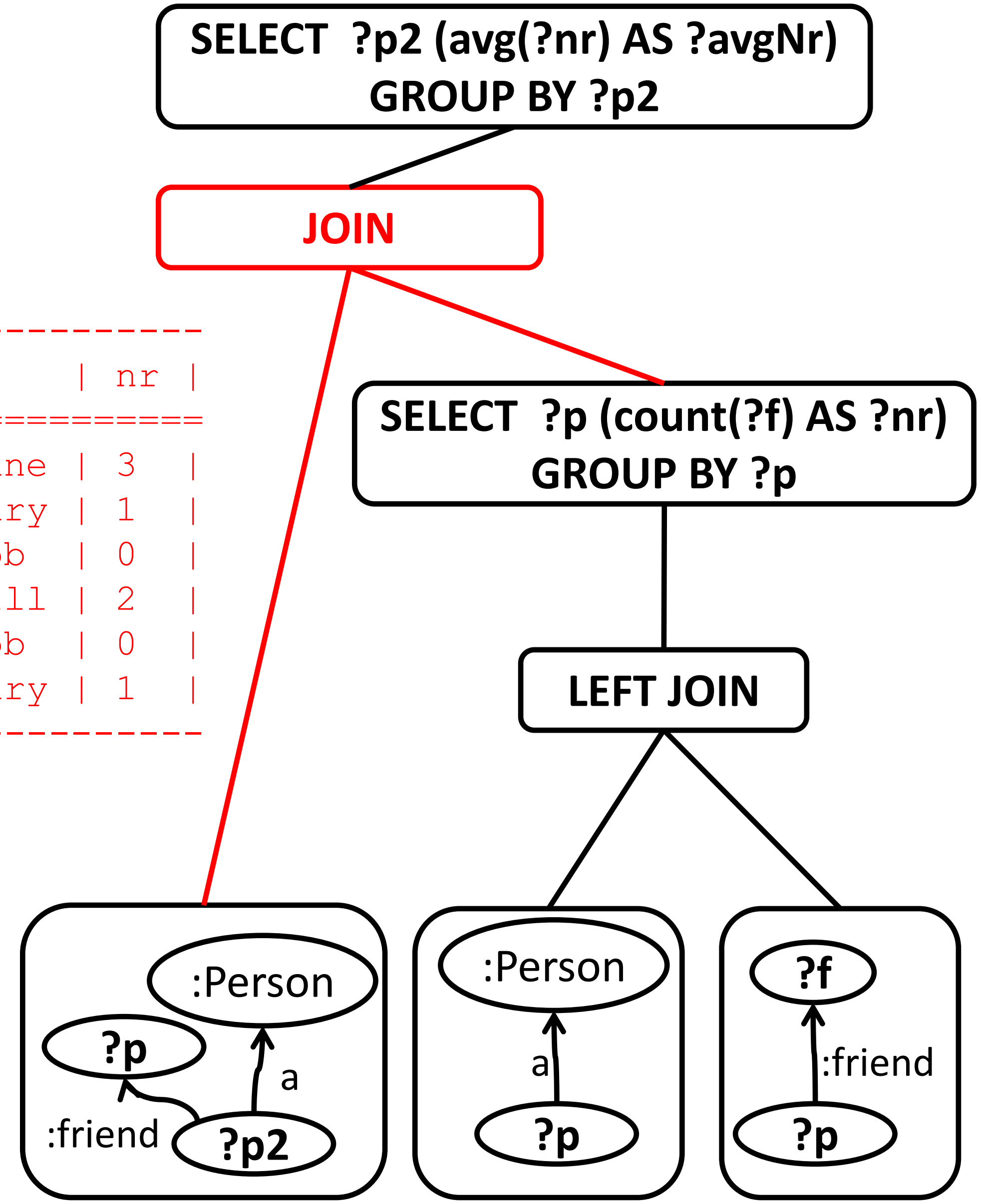
Multistep Aggregation with Subqueries

```
SELECT  ?p2 (avg(?nr) AS ?avgNr)
WHERE
  { ?p2 rdf:type :Person .
    ?p2 :friend ?p
    { SELECT  ?p (count(?f) AS ?nr)
      WHERE
        { ?p rdf:type :Person
          OPTIONAL
            { ?p :friend ?f}
        }
      GROUP BY ?p
    }
  }
GROUP BY ?p2
```

```
----------------------
| p2    | p     | nr |
======================
| :bill | :jane | 3  |
| :bill | :mary | 1  |
| :mary | :bob  | 0  |
| :jane | :bill | 2  |
| :jane | :bob  | 0  |
| :jane | :mary | 1  |
----------------------
```

```
-----------------
| p2    | p     |
=================
| :bill | :jane |
| :bill | :mary |
| :mary | :bob  |
| :jane | :bill |
| :jane | :bob  |
| :jane | :mary |
-----------------
```

```
--------------
| p     | nr |
==============
| :jane | 3  |
| :bill | 2  |
| :mary | 1  |
| :bob  | 0  |
--------------
```

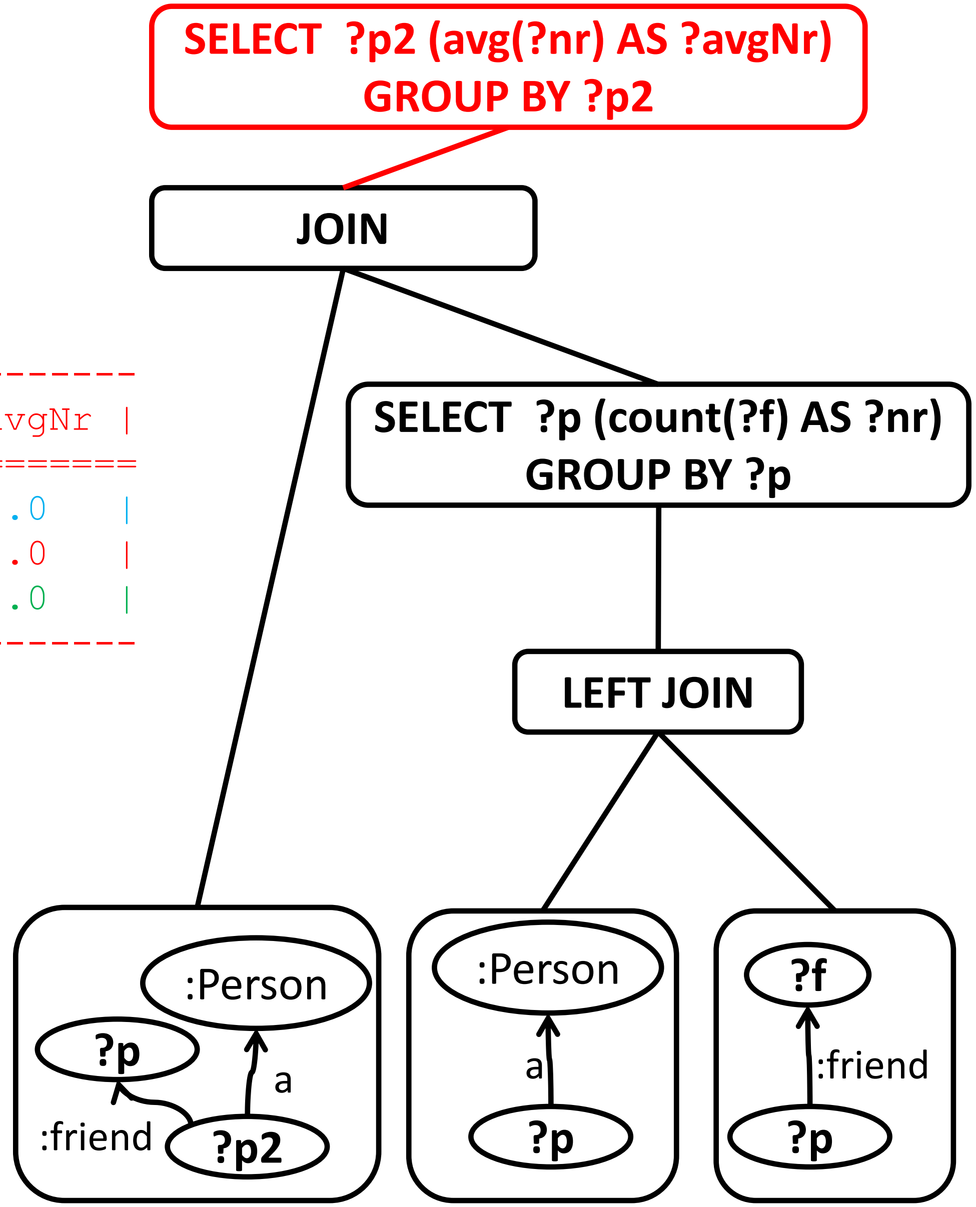


Multistep Aggregation with Subqueries

```
SELECT  ?p2 (avg(?nr) AS ?avgNr)
WHERE
  { ?p2 rdf:type :Person .
    ?p2 :friend ?p
    { SELECT  ?p (count(?f) AS ?nr)
      WHERE
        { ?p rdf:type :Person
          OPTIONAL
            { ?p :friend ?f}
        }
      GROUP BY ?p
    }
  }
GROUP BY ?p2
```

```
-----------------
| p2    | avgNr |
=================
| :jane | 1.0   |
| :bill | 2.0   |
| :mary | 0.0   |
-----------------
```

```
----------------------
| p2    | p     | nr |
======================
| :bill | :jane | 3  |
| :bill | :mary | 1  |
| :mary | :bob  | 0  |
| :jane | :bill | 2  |
| :jane | :bob  | 0  |
| :jane | :mary | 1  |
----------------------
```



Property Paths

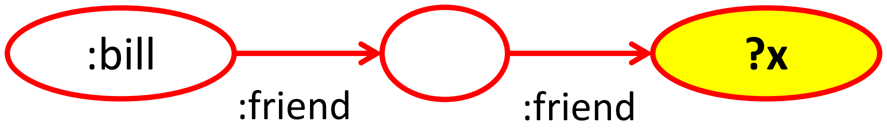
PROPERTY PATHS

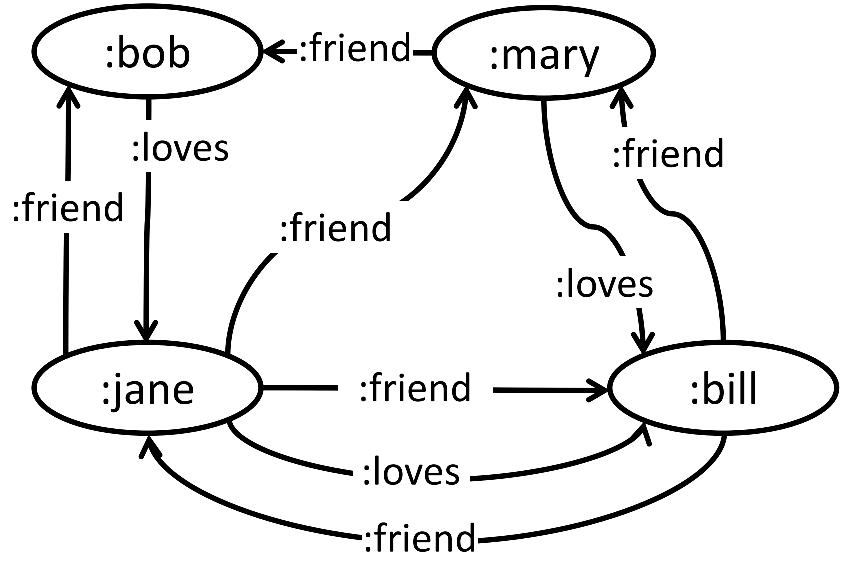
Property Paths

- A property path is **a possible route through a graph between two graph nodes**. A trivial case is a property path of length exactly 1, which is a triple pattern. The ends of the path may be RDF terms or variables. Variables cannot be used as part of the path itself, only the ends.
- Property paths allow for more concise expressions for some SPARQL basic graph patterns.
- SPARQL property paths treat the RDF triples as a directed, possibly cyclic, graph with named edges. Some property paths are equivalent to a translation into triple patterns and SPARQL UNION graph patterns. Evaluation of a property path expression can lead to **duplicates** because any variables introduced in the equivalent pattern are not part of the result if they are not already used elsewhere.

Property Paths: Sequence /

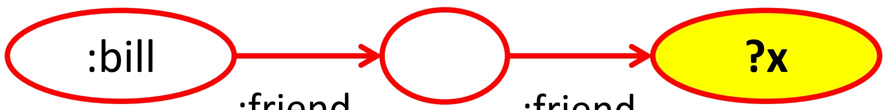
```
SELECT  *
WHERE
  { :bill :friend/:friend ?x}
```

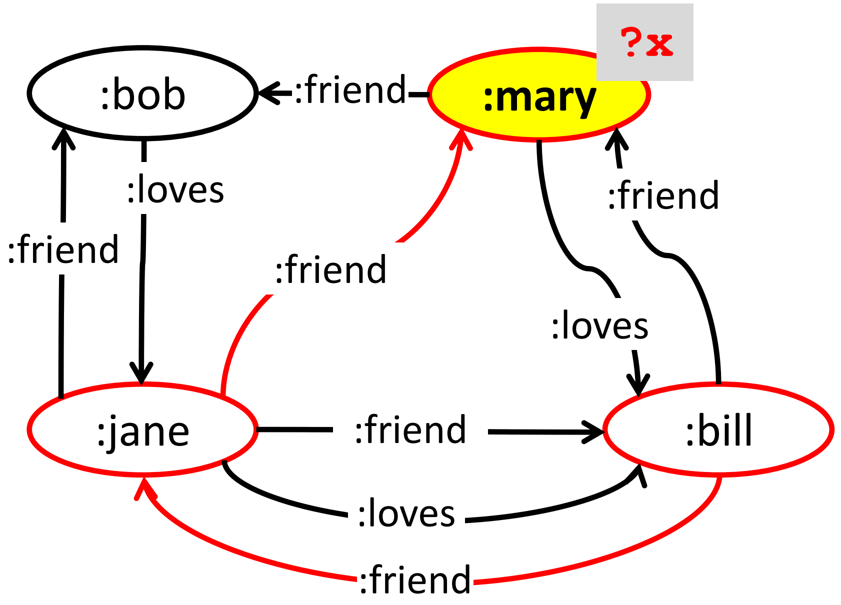




Property Paths: Sequence /

```
SELECT  *
WHERE
  { :bill :friend/:friend ?x}

---------
| x     |
=========
| :mary |
| :bill |
| :bob  |
| :bob  |
---------
```

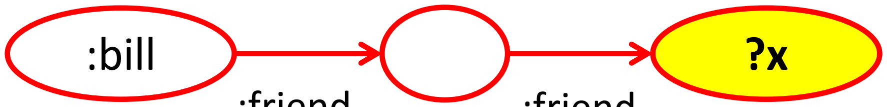


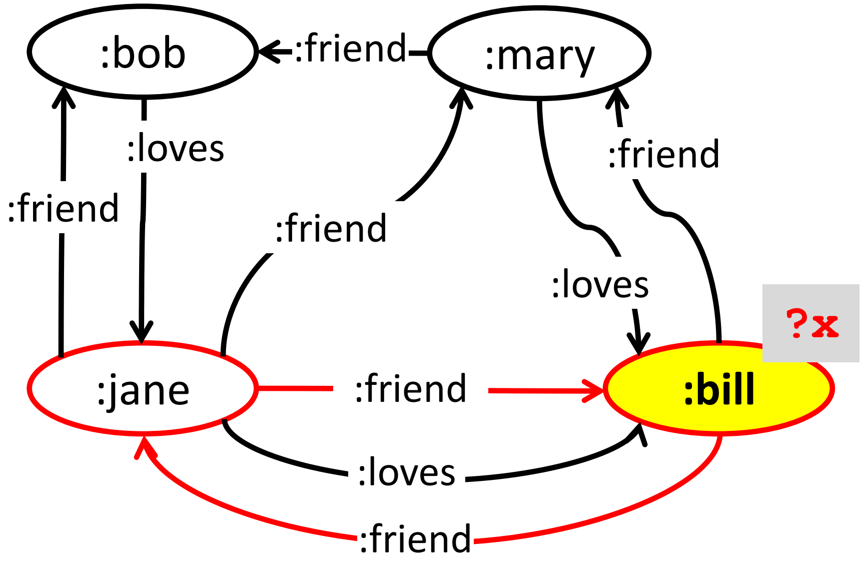


Property Paths: Sequence /

```
SELECT  *
WHERE
  { :bill :friend/:friend ?x}

---------
| x     |
=========
| :mary |
| :bill |
| :bob  |
| :bob  |
---------
```

A match may contain cycles

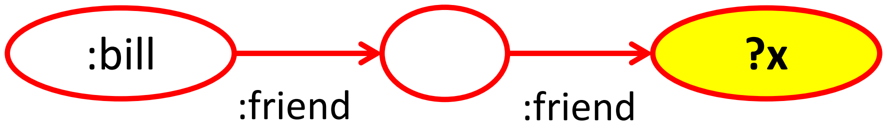


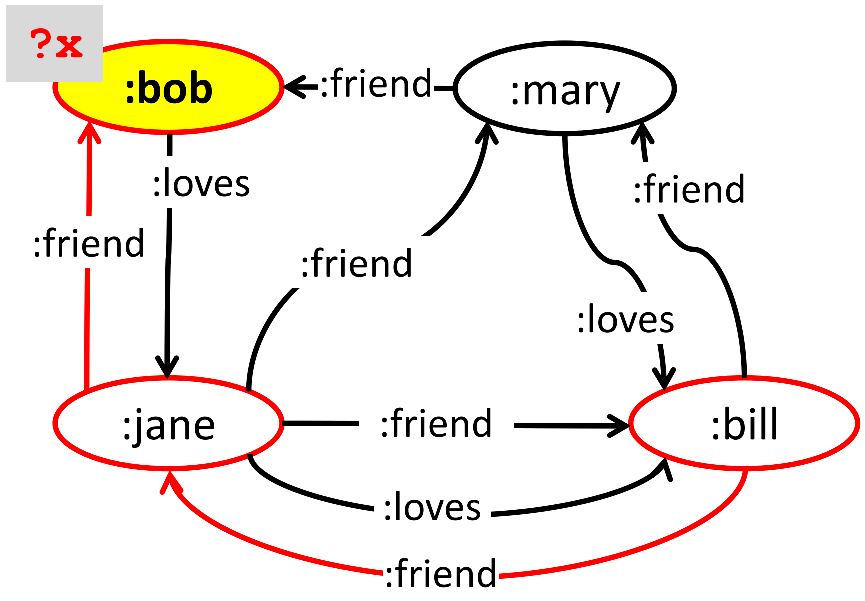


Property Paths: Sequence /

```
SELECT  *
WHERE
  { :bill :friend/:friend ?x}
```

```
---------
| x     |
=========
| :mary |
| :bill |
| :bob  |
| :bob  |
---------
```



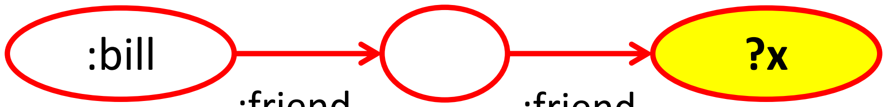


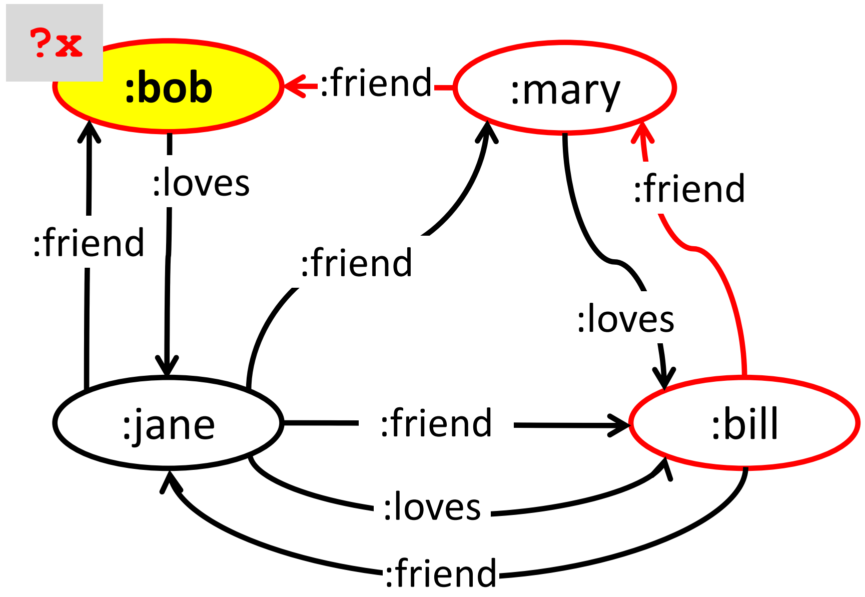
Property Paths: Sequence /

```
SELECT  *
WHERE
  { :bill :friend/:friend ?x}
```

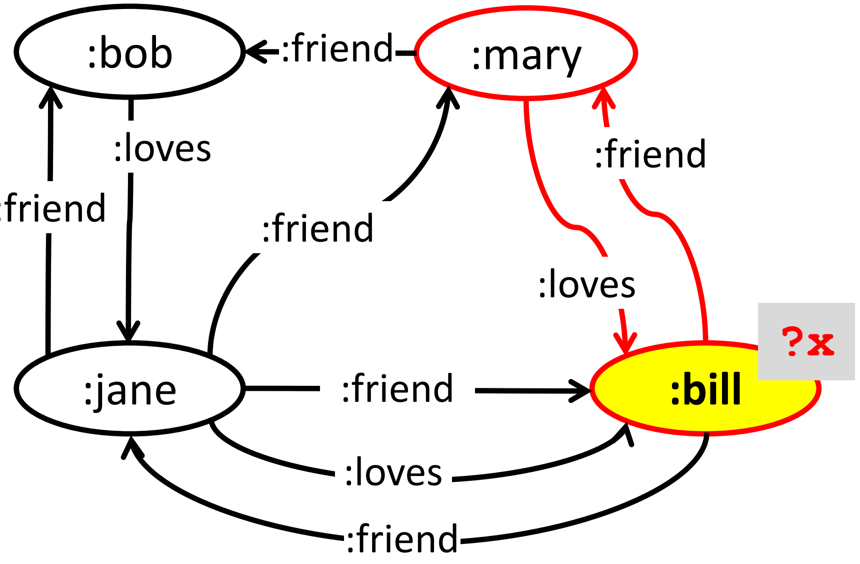
```
---------
| x     |
=========
| :mary |
| :bill |
| :bob  |
| :bob  |
---------
```

Multiple matches produce duplicate results

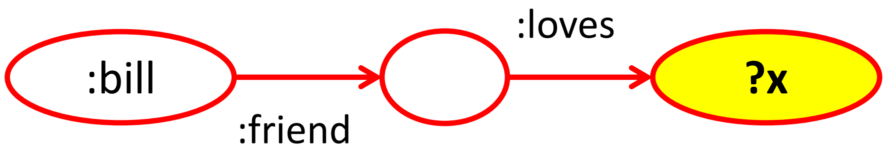




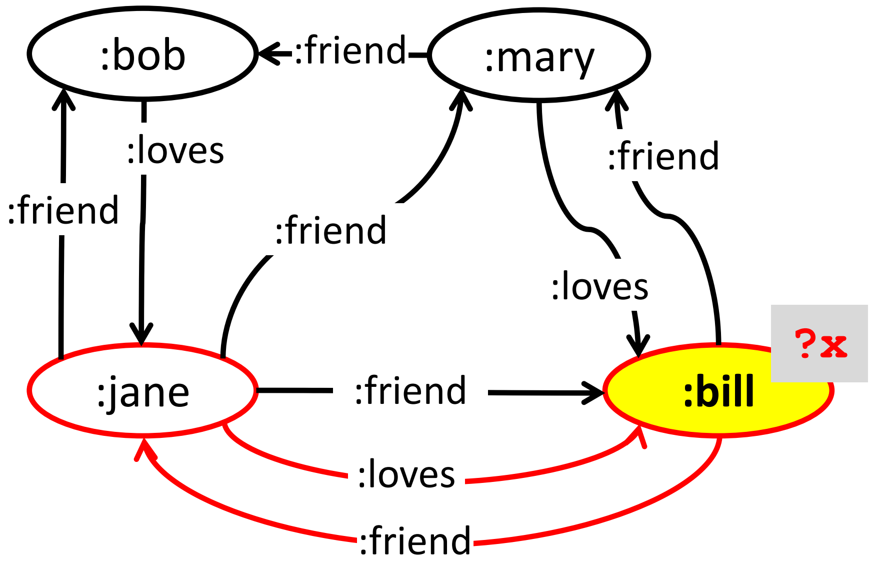
Property Paths: Sequence /



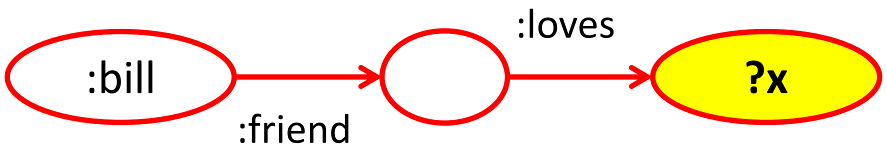
```
SELECT  *
WHERE
  { :bill :friend/:loves ?x}

---------
| x     |
=========
| :bill |
| :bill |
---------
```



Property Paths: Sequence /

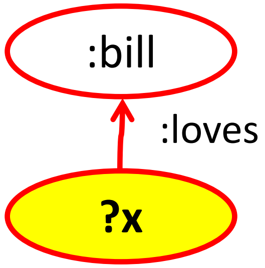


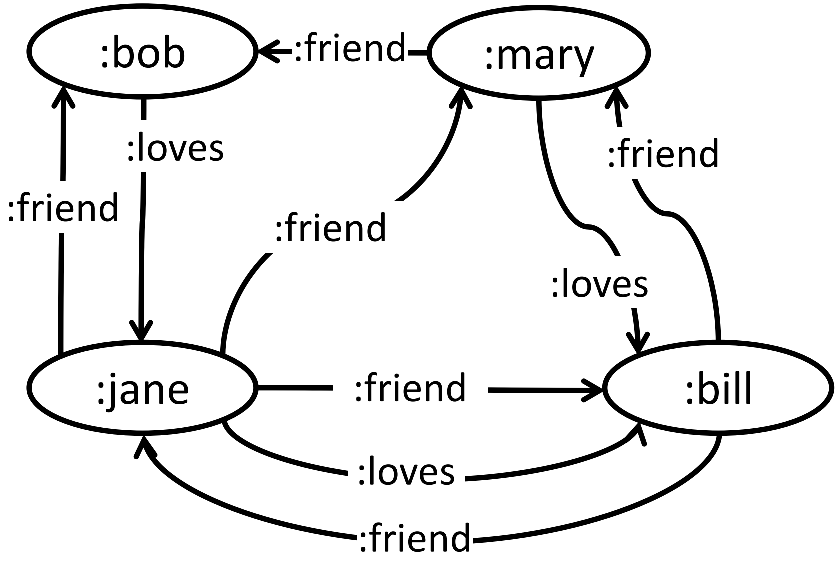
```
SELECT  *
WHERE
  { :bill :friend/:loves ?x}

---------
| x     |
=========
| :bill |
| :bill |
---------
```



Property Paths: Reverse Paths ^

```
SELECT  *
WHERE
  { :bill ^:loves ?x}
```

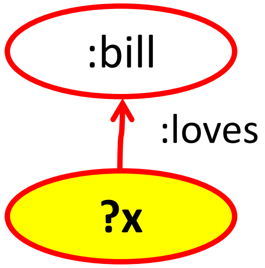


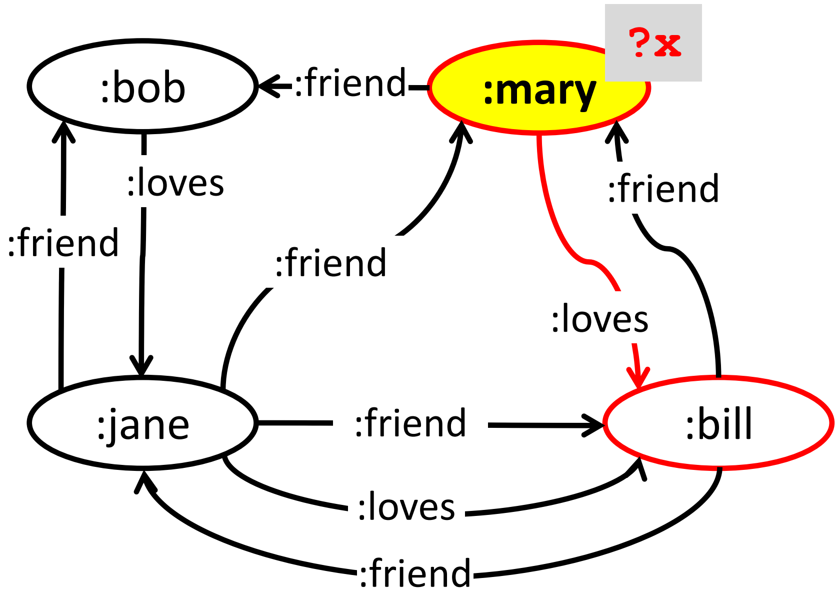


Property Paths: Reverse Paths ^

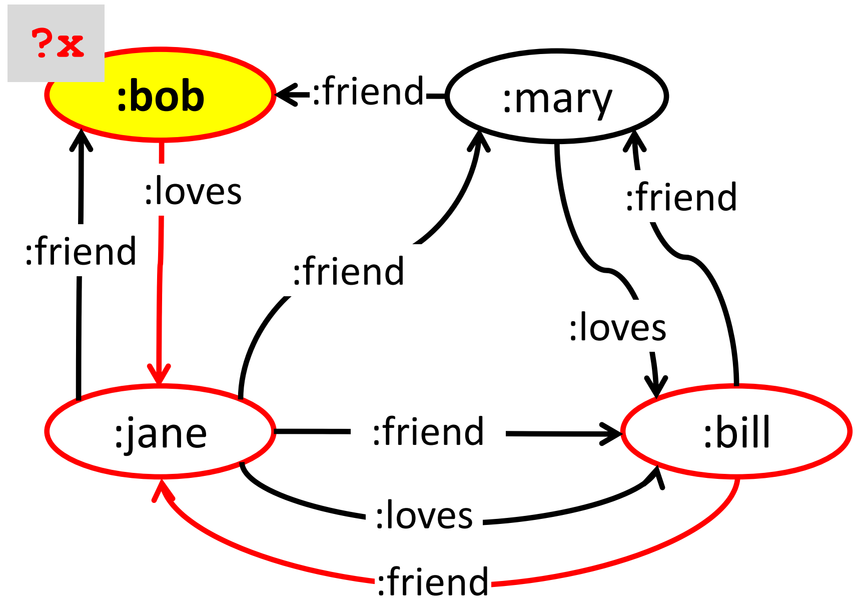
```
SELECT  *
WHERE
  { :bill ^:loves ?x}
```

```
---------
| x     |
=========
| :mary |
| :jane |
---------
```



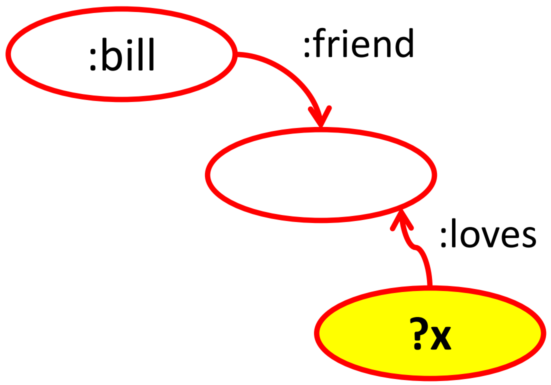


Property Paths: Reverse Paths ^



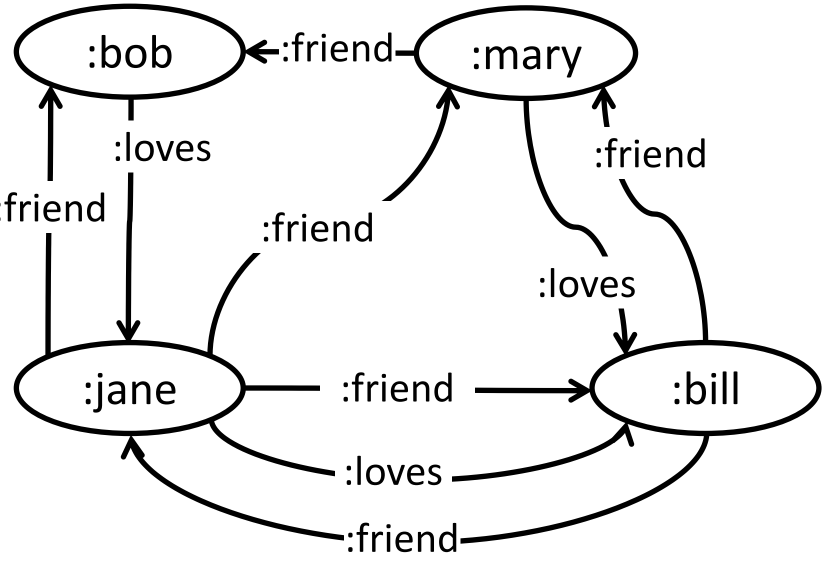
```
SELECT  *
WHERE
  { :bill :friend/^:loves ?x}
```

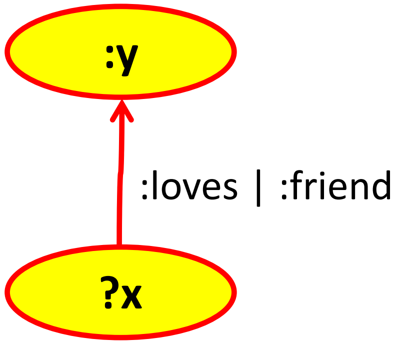
```
---------
| x      |
=========
| :bob   |
---------
```



Property Paths: Alternatives |

```
SELECT  *
WHERE
  { ?x (:loves | :friend) ?y}
```

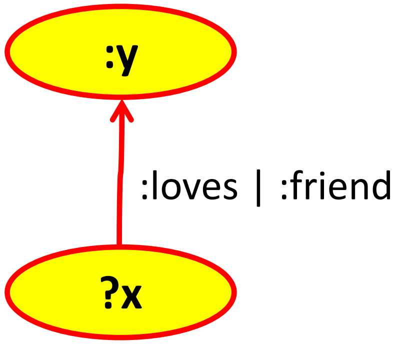


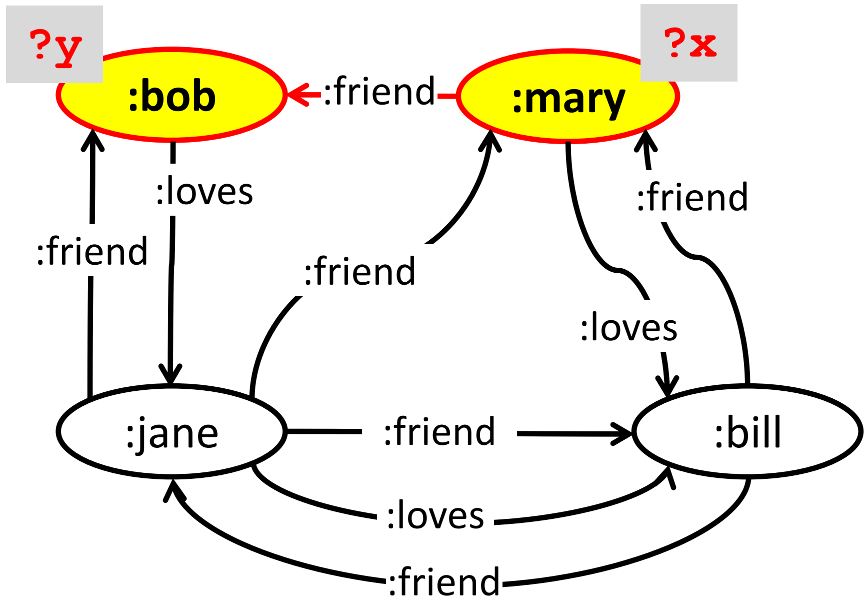


Property Paths: Alternatives |

```
SELECT  *
WHERE
  { ?x (:loves | :friend) ?y}
```

```
-----------------
| x      | y     |
=================
| :mary | :bob   |
| :mary | :bill |
| :bill | :jane |
| :bill | :mary |
| :jane | :bill |
| :jane | :bob   |
| :jane | :mary |
| :jane | :bill |
| :bob   | :jane |
-----------------
```

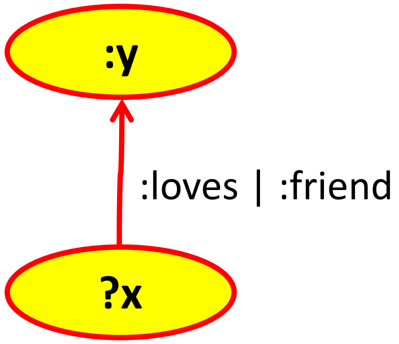


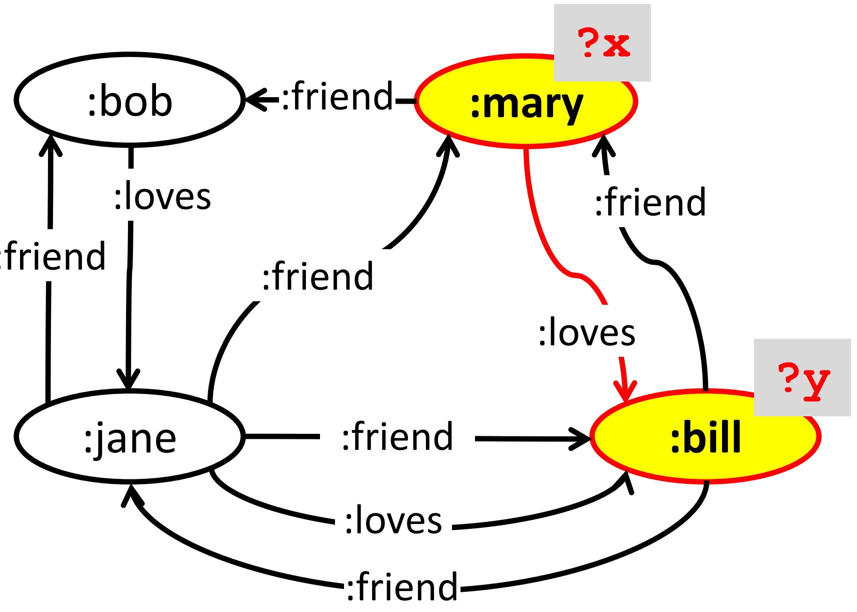


Property Paths: Alternatives |

```
SELECT  *
WHERE
  { ?x (:loves | :friend) ?y}
```

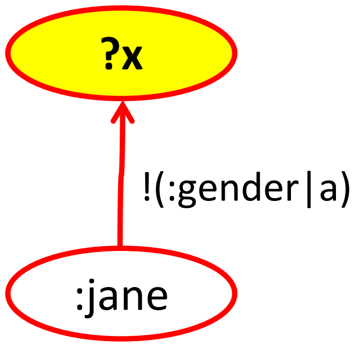
```
-----------------
| x     | y     |
=================
| :mary | :bob  |
| :mary | :bill |
| :bill | :jane |
| :bill | :mary |
| :jane | :bill |
| :jane | :bob  |
| :jane | :mary |
| :jane | :bill |
| :bob  | :jane |
-----------------
```

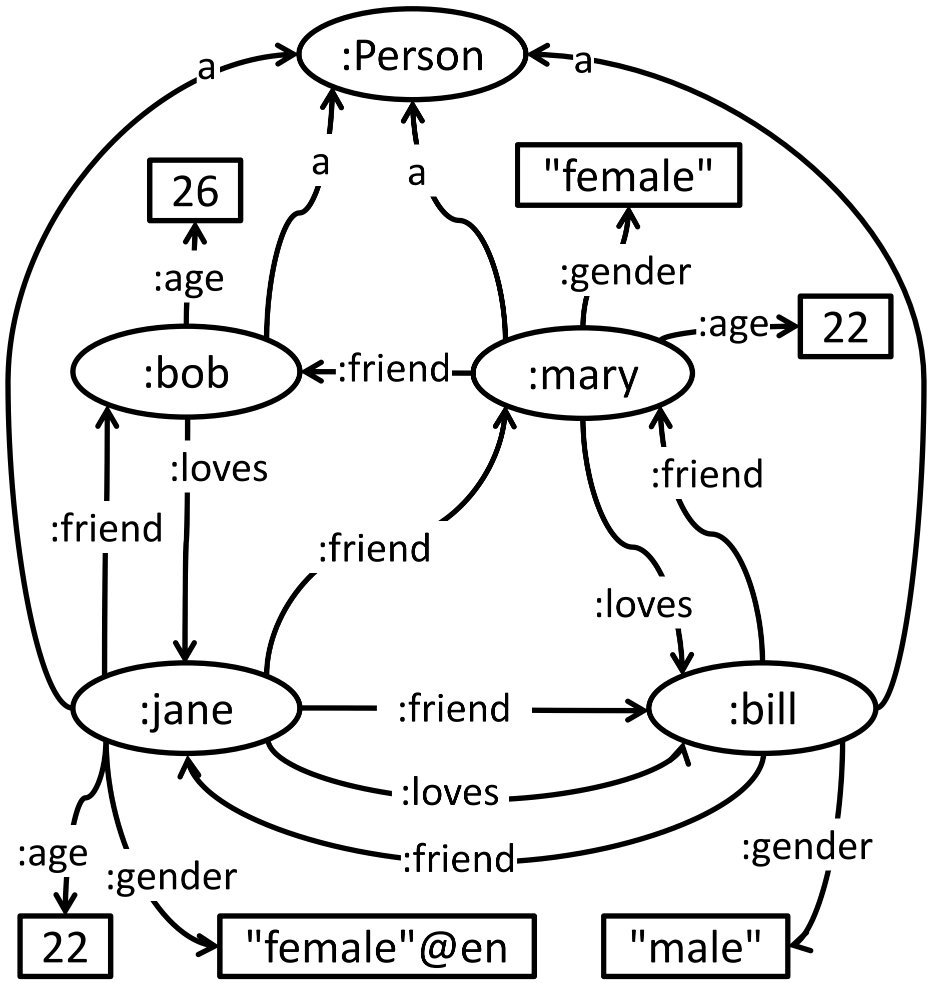




Property Paths: Negated Property Sets !

```
SELECT  *
WHERE
  { :jane !( :gender | a ) ?x}
```

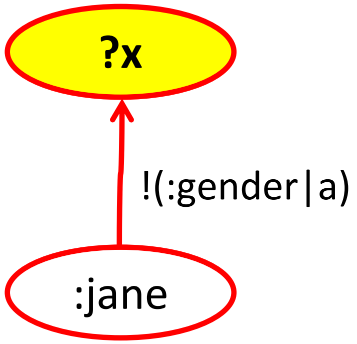


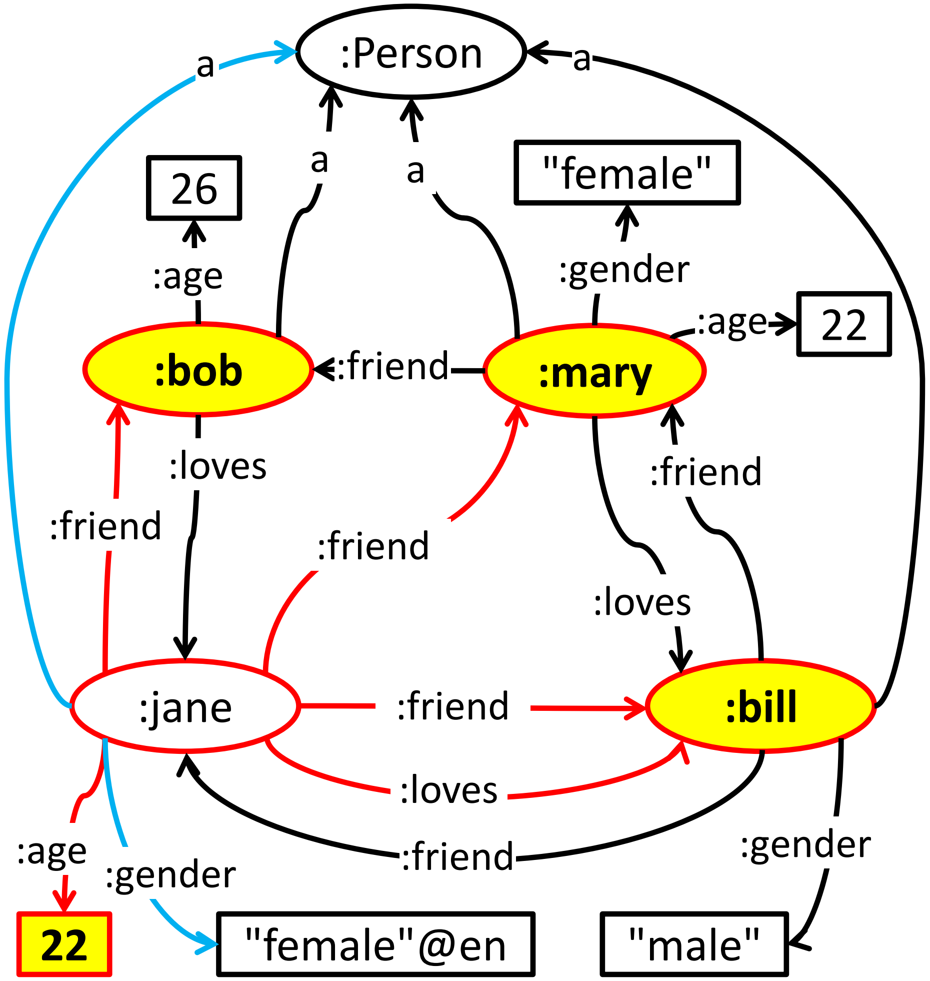


Property Paths: Negated Property Sets !

```
SELECT  *
WHERE
  { :jane !( :gender | a ) ?x}
```

```
---------
| x     |
=========
| :bill |
| :bill |
| :bob  |
| :mary |
| 22    |
---------
```



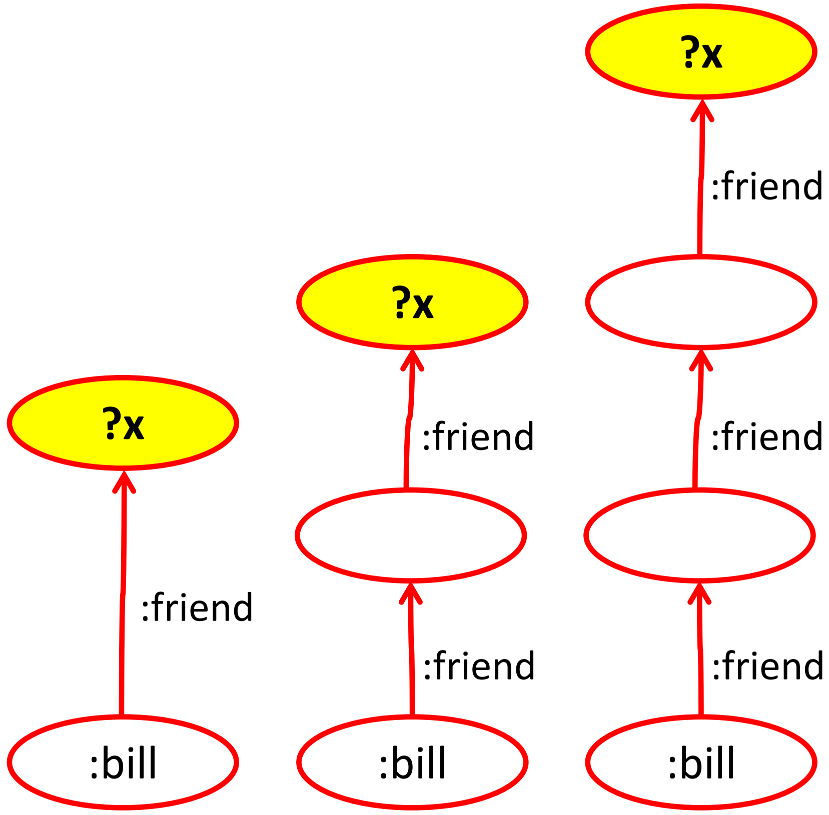


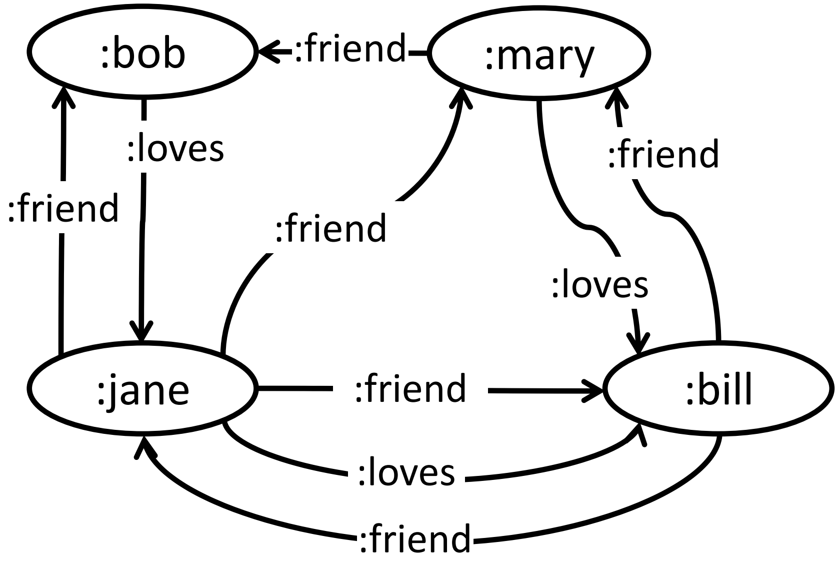
Property Paths: Arbitrary Length Path Matching

- Connectivity between the subject and object by **a property path of arbitrary length** can be found using the
  - "**zero or more**" property path operator: *****
  - "**one or more**" property path operator: **+**
  - "**zero or one**" property path operator: **?**
- Each of these operators uses the property path expression to try to find a connection between subject and object, using the path step a number of times, as restricted by the operator.
- Such connectivity matching **does not introduce duplicates** (it does not incorporate any count of the number of ways the connection can be made) **even if the repeated path itself would otherwise result in duplicates**.
- The graph matched may include cycles.

Property Paths: Transitive Closure (one or more) +

```
SELECT  *
WHERE
  { :bill (:friend)+ ?x}
```

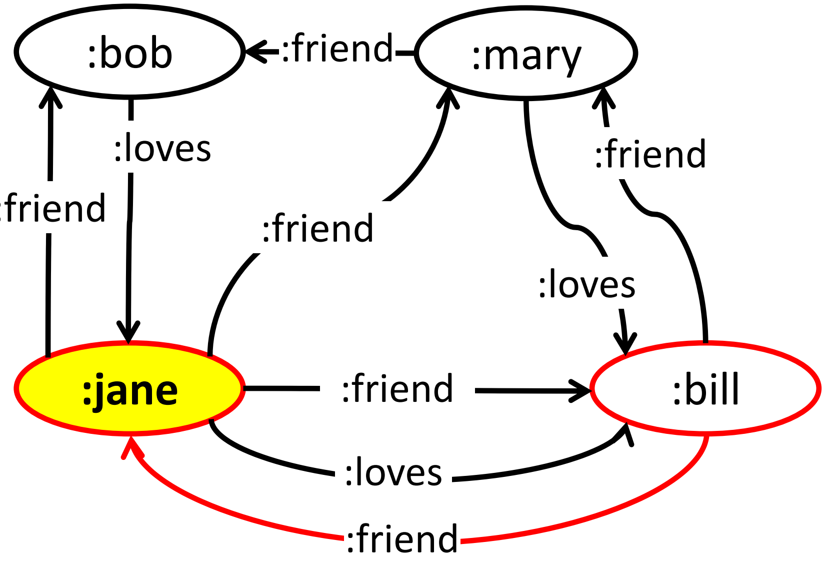




and so forth ...

Property Paths: Transitive Closure (one or more) **+**

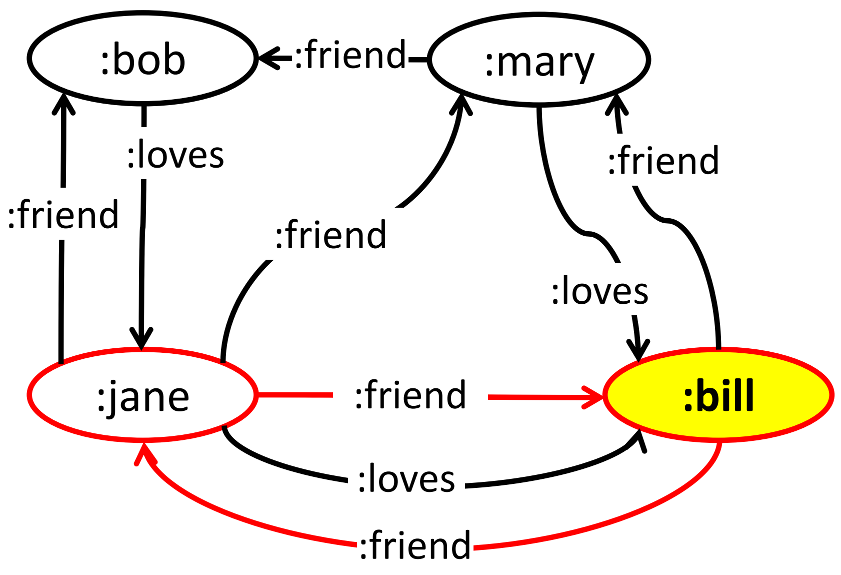
```
SELECT  *
WHERE
  { :bill (:friend)+ ?x}

---------
| x       |
=========
| :jane |
| :bill |
| :mary |
| :bob   |
---------
```



Property Paths: Transitive Closure (one or more) **+**

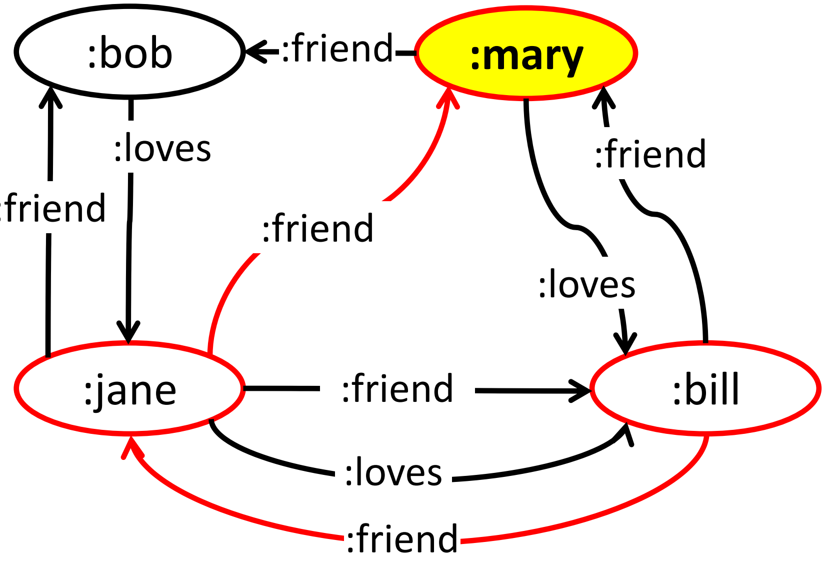
```
SELECT  *
WHERE
  { :bill (:friend)+ ?x}

---------
| x     |
=========
| :jane |
| :bill |
| :mary |
| :bob  |
---------
```

path may contain cycles



Property Paths: Transitive Closure (one or more) +

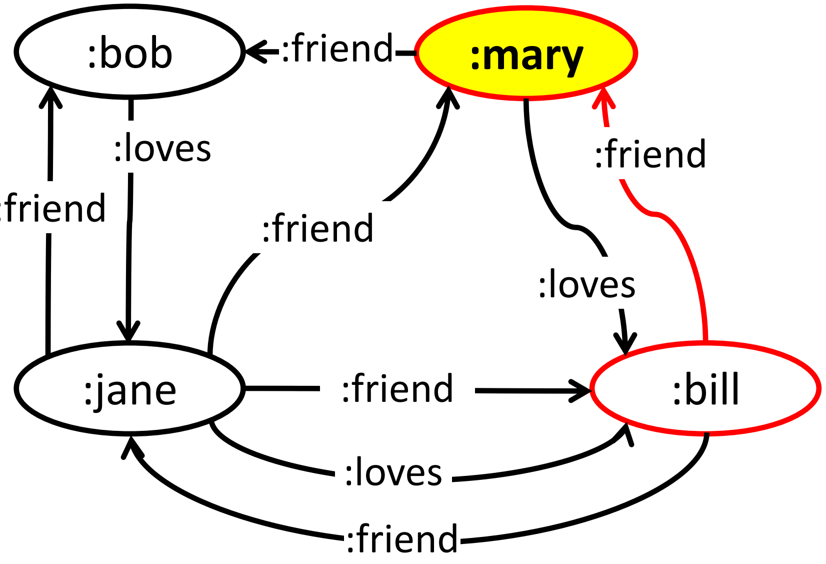
```
SELECT  *
WHERE
  { :bill (:friend)+ ?x}

---------
| x     |
=========
| :jane |
| :bill |
| :mary |
| :bob  |
---------
```



Property Paths: Transitive Closure (one or more) **+**

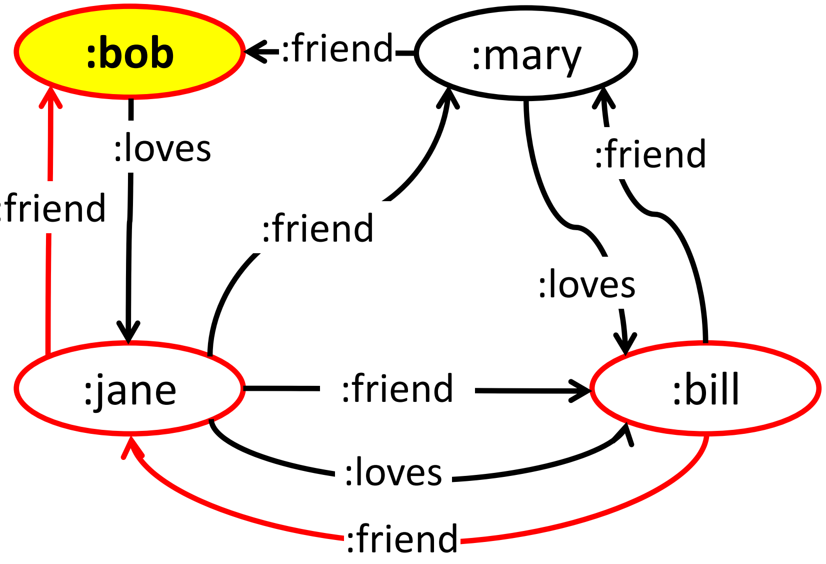
```
SELECT  *
WHERE
  { :bill (:friend)+ ?x}

---------
| x     |
=========
| :jane |
| :bill |
| :mary |
| :bob  |
---------
```

multiple matches do not produce duplicate results!



Property Paths: Transitive Closure (one or more) +

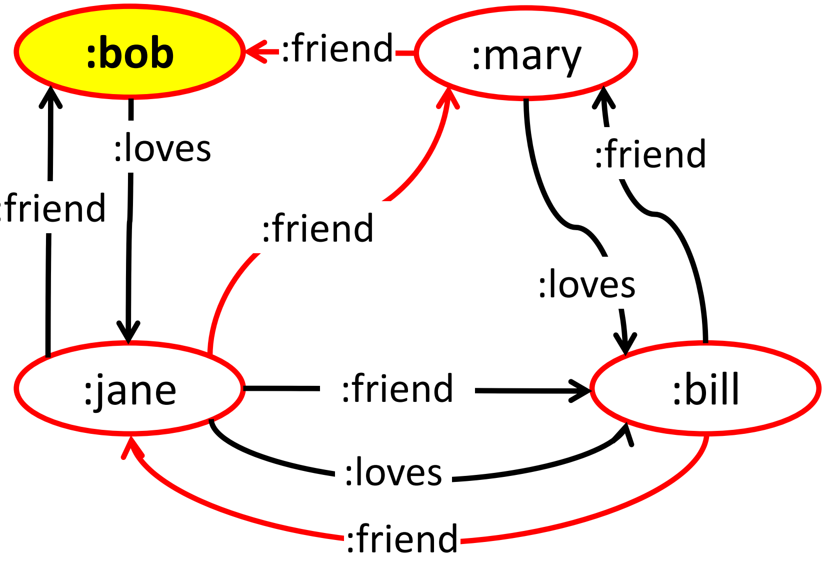
```
SELECT  *
WHERE
  { :bill (:friend)+ ?x}

---------
| x       |
=========
| :jane |
| :bill |
| :mary |
| :bob  |
---------
```



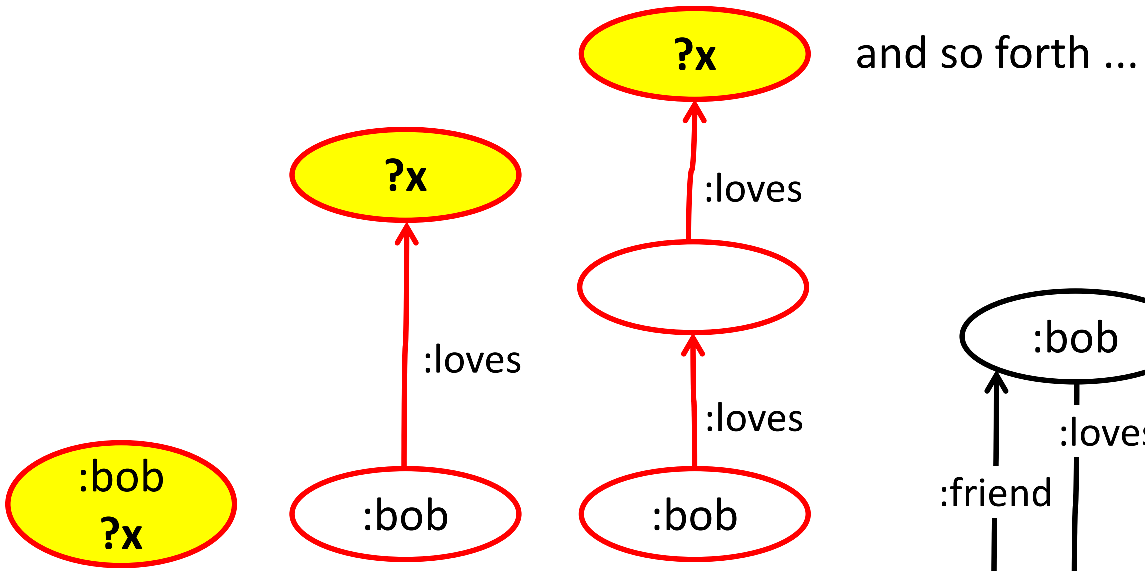
Property Paths: Transitive Closure (one or more) +

```
SELECT  *
WHERE
  { :bill (:friend)+ ?x}

---------
| x     |
=========
| :jane |
| :bill |
| :mary |
| :bob  |
---------
```

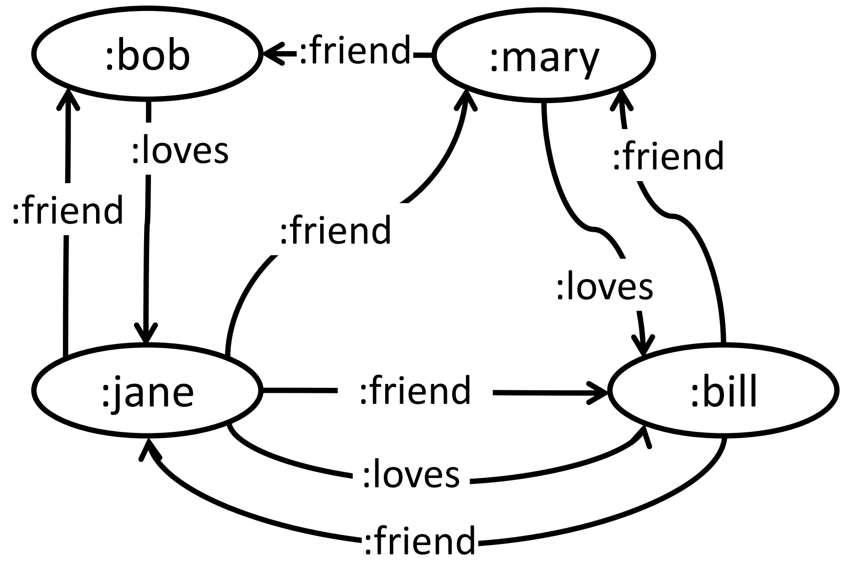
multiple matches do not produce duplicate results!



Property Paths: Transitive-reflexive closure (zero or more) *

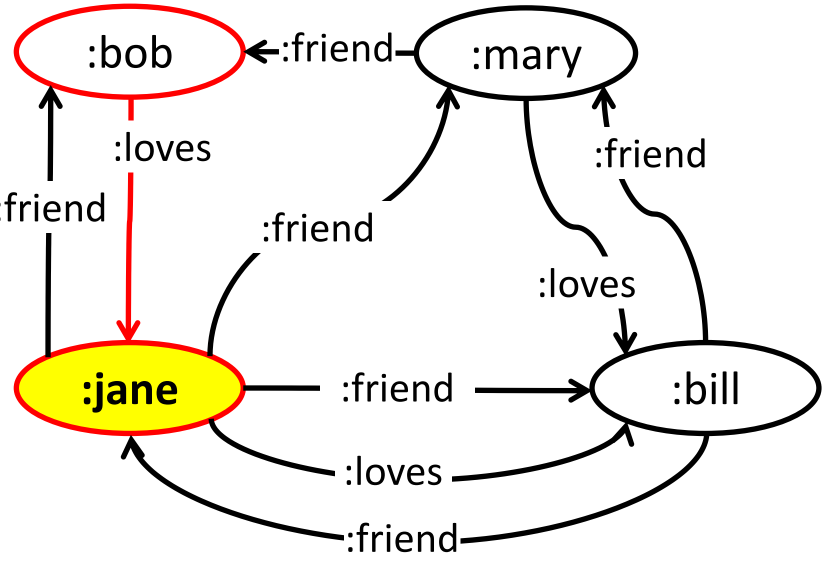


```
SELECT  *
WHERE
  { :bob (:loves)* ?x}
```



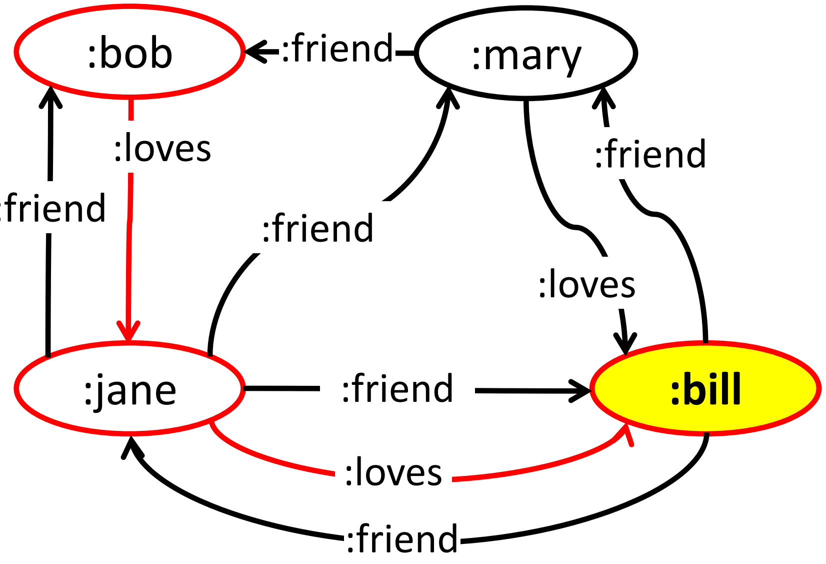
Property Paths: Transitive-reflexive closure (zero or more) *

```
SELECT  *
WHERE
  { :bob (:loves)* ?x}

---------
| x       |
=========
| :jane |
| :bill |
| :bob  |
---------
```



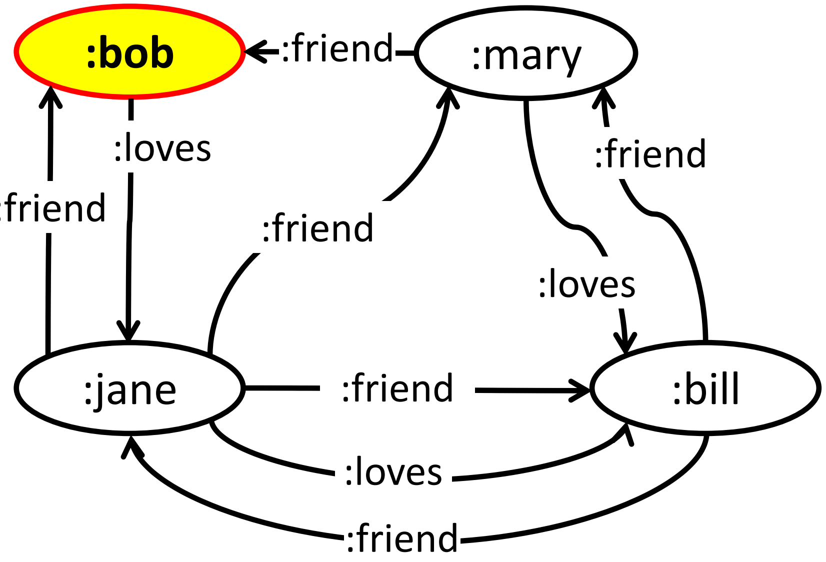
Property Paths: Transitive-reflexive closure (zero or more) *

```
SELECT  *
WHERE
  { :bob (:loves)* ?x}

---------
| x     |
=========
| :jane |
| :bill |
| :bob  |
---------
```



Property Paths: Transitive-reflexive closure (zero or more) *

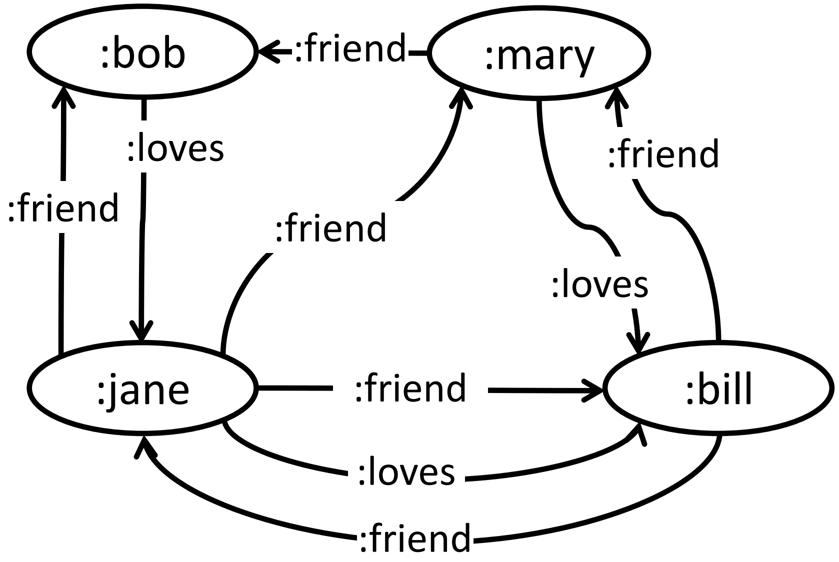
path with length 0



```
SELECT  *
WHERE
  { :bob (:loves)* ?x}

---------
| x     |
=========
| :jane |
| :bill |
| :bob  |
---------
```

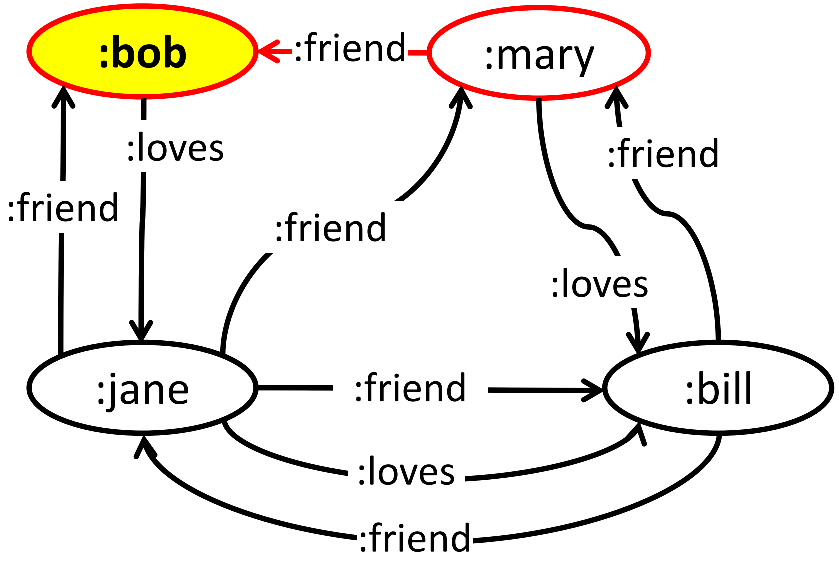
Property Paths: zero or one ?

```
SELECT *
WHERE
  { :mary (:loves|:friend)? ?x }
```



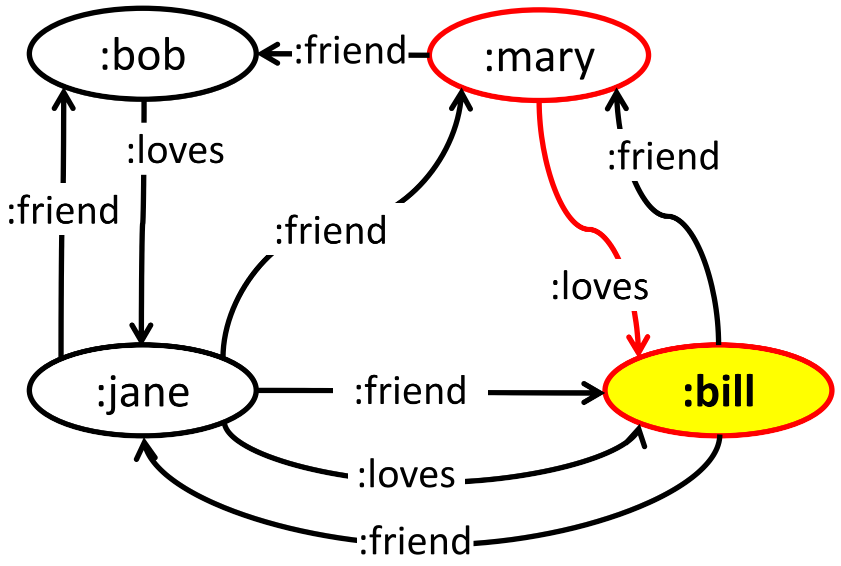
Property Paths: zero or one ?

```
SELECT *
WHERE
  { :mary (:loves|:friend)? ?x }

---------
| x     |
=========
| :bob  |
| :bill |
| :mary |
---------
```



Property Paths: zero or one ?

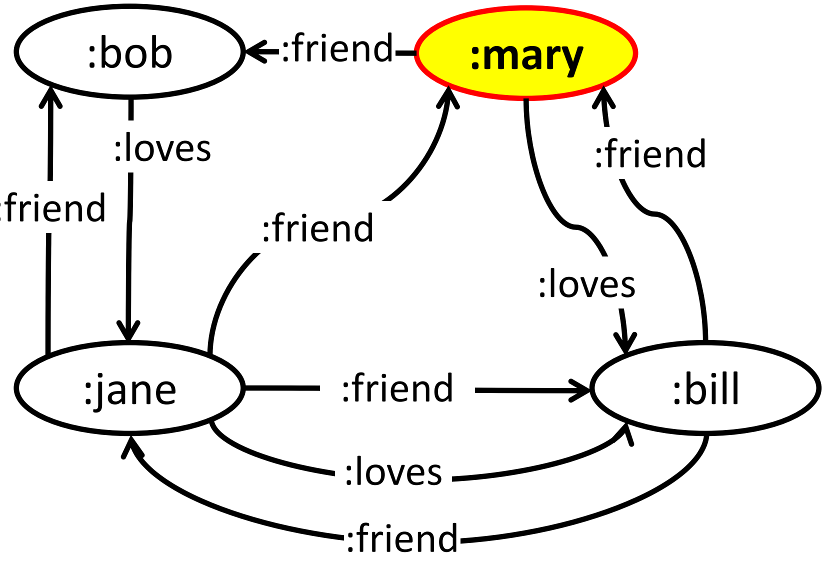
```
SELECT *
WHERE
  { :mary (:loves|:friend)? ?x }

---------
| x     |
=========
| :bob  |
| :bill |
| :mary |
---------
```



Property Paths: zero or one ?

```
SELECT *
WHERE
  { :mary (:loves|:friend)? ?x }

---------
| x     |
=========
| :bob  |
| :bill |
| :mary |
---------
```

path with length 0

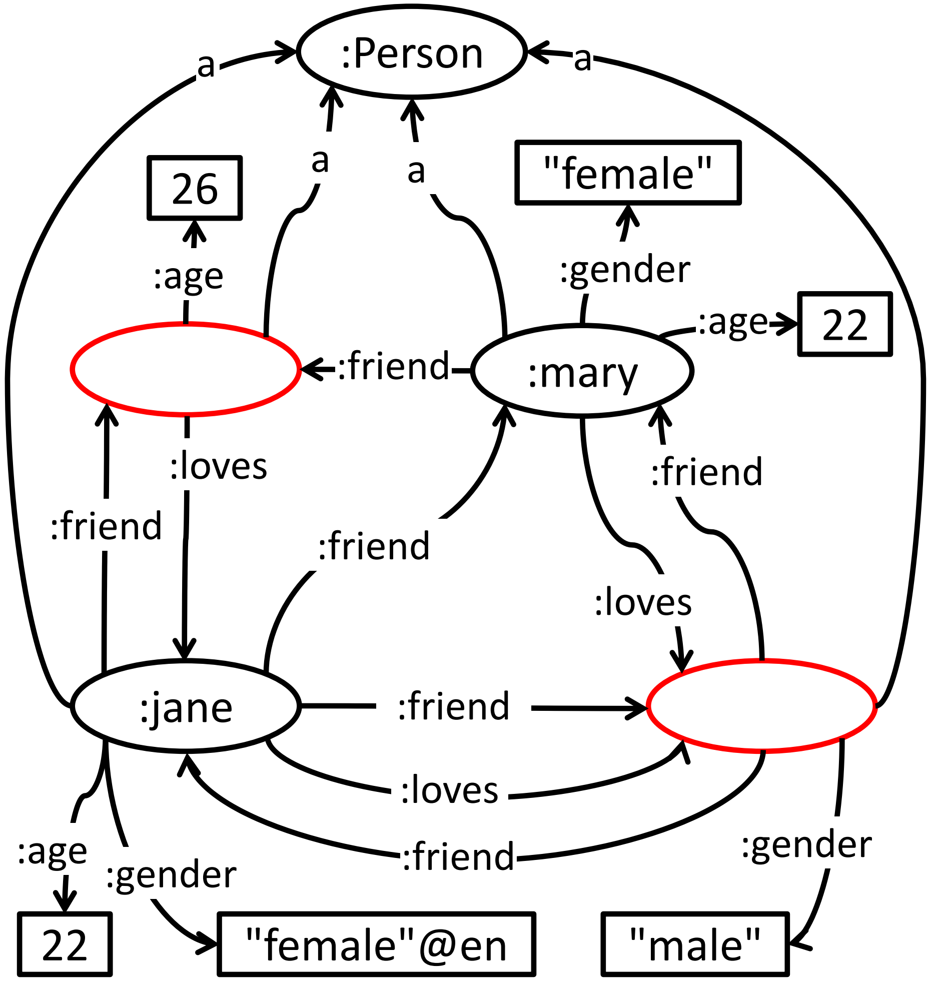


Blank Nodes

RDF GRAPHS WITH BLANK NODES

RDF Graphs with Blank Nodes

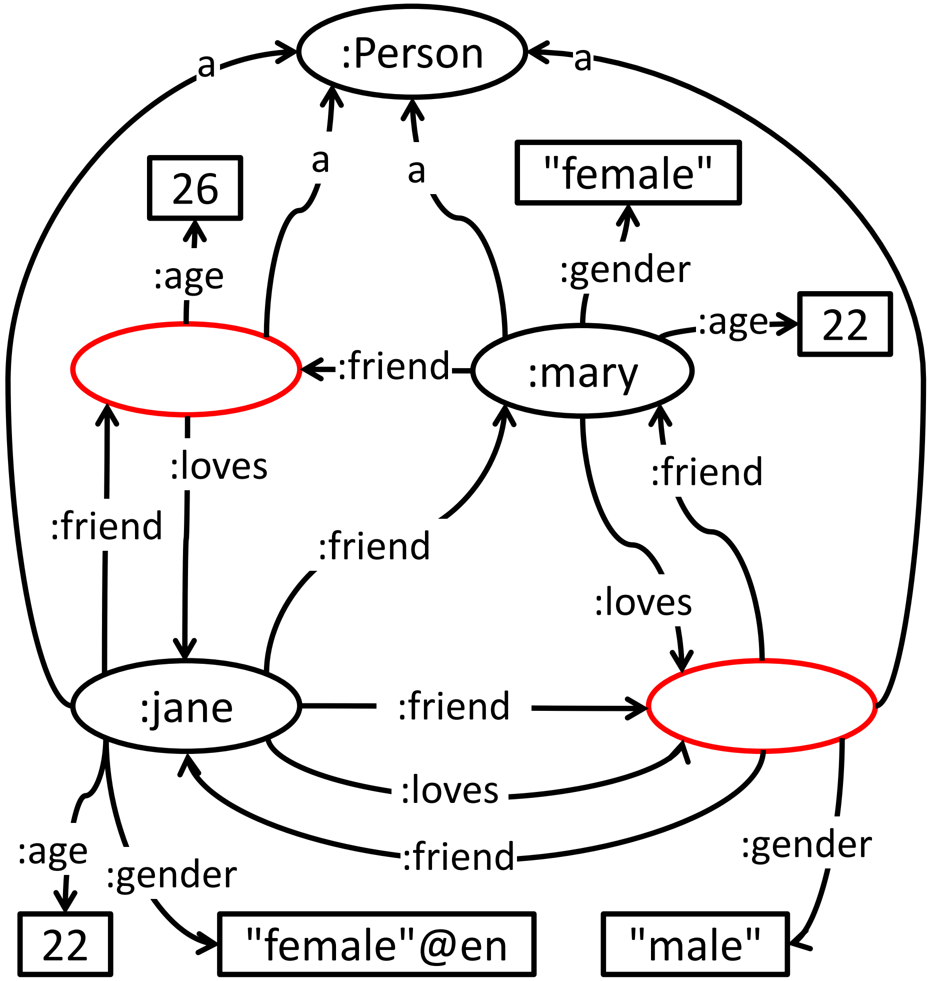
```
@prefix : <http://example.org/> .

:jane  a :Person;
          :gender "female"@en; :age 22;
          :friend :mary, _:2, _:1;
          :loves _:1.

:mary  a :Person;
          :gender "female"; :age 22;
          :friend _:2;
          :loves _:1.

_:2  a :Person;
          :age 26;
          :loves :jane.

_:1  a :Person;
          :gender "male";
          :friend :mary, :jane.
```



Querying RDF Graphs with Blank Nodes

```
SELECT  *                                    zero/one
WHERE                                           |
  { :mary (:loves|:friend)? ?x}
```

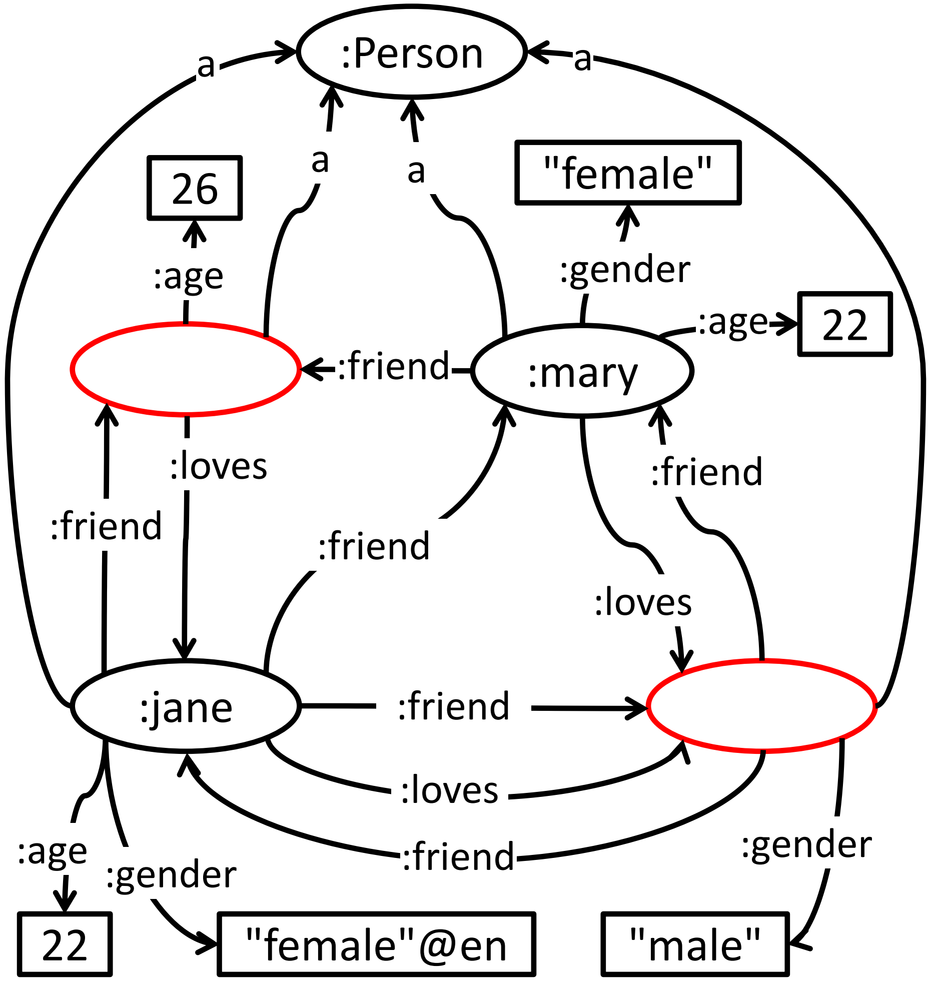
```
----------
| x      |
==========
| :mary  |
| _:b0   |
| _:b1   |
----------
```



Querying RDF Graphs with Blank Nodes

```
SELECT  *
WHERE
  { :mary (:loves|:friend)? ?x
    FILTER isBlank(?x)
  }
```

```
----------
| x      |
==========
| _:b0   |
| _:b1   |
----------
```



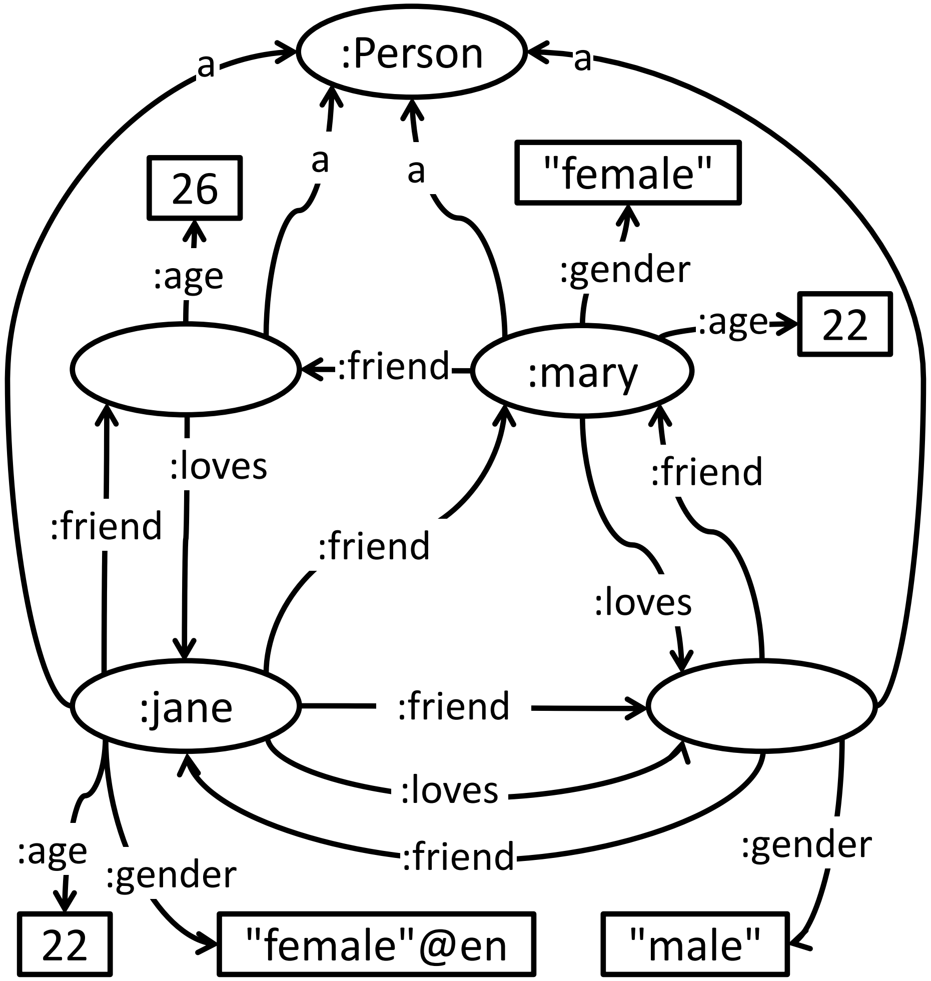
Blank Node Syntax in Graph Patterns

verstehe diese folie nicht.

? keine visuelle erklärung...

```
SELECT *
WHERE
  { _:a :friend ?x. }
```

```
----------
| x      |
==========
| :jane  |
| :mary  |
| _:b0   |
| _:b1   |
| _:b0   |
| :mary  |
----------
```

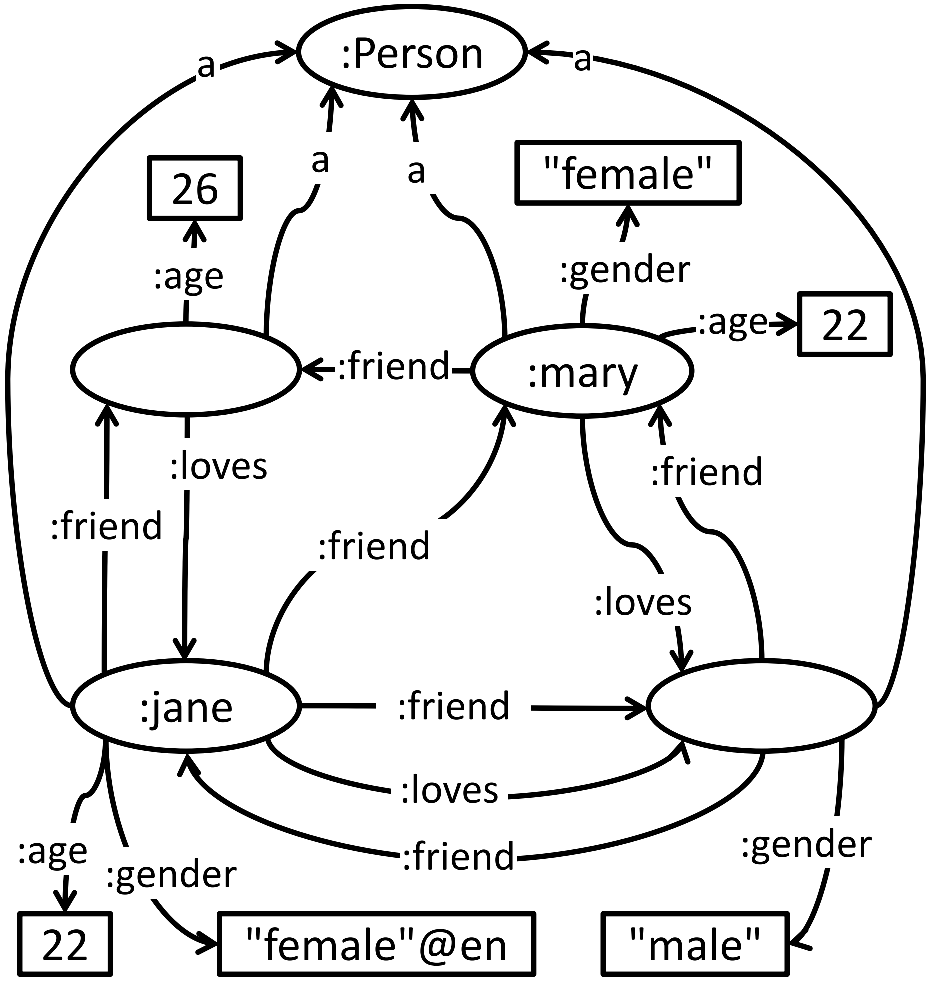


Blank Node Syntax in Graph Patterns

```
SELECT *
WHERE
  { :jane :friend [ :friend ?y]. }
```

was heißt die eckige klammer??

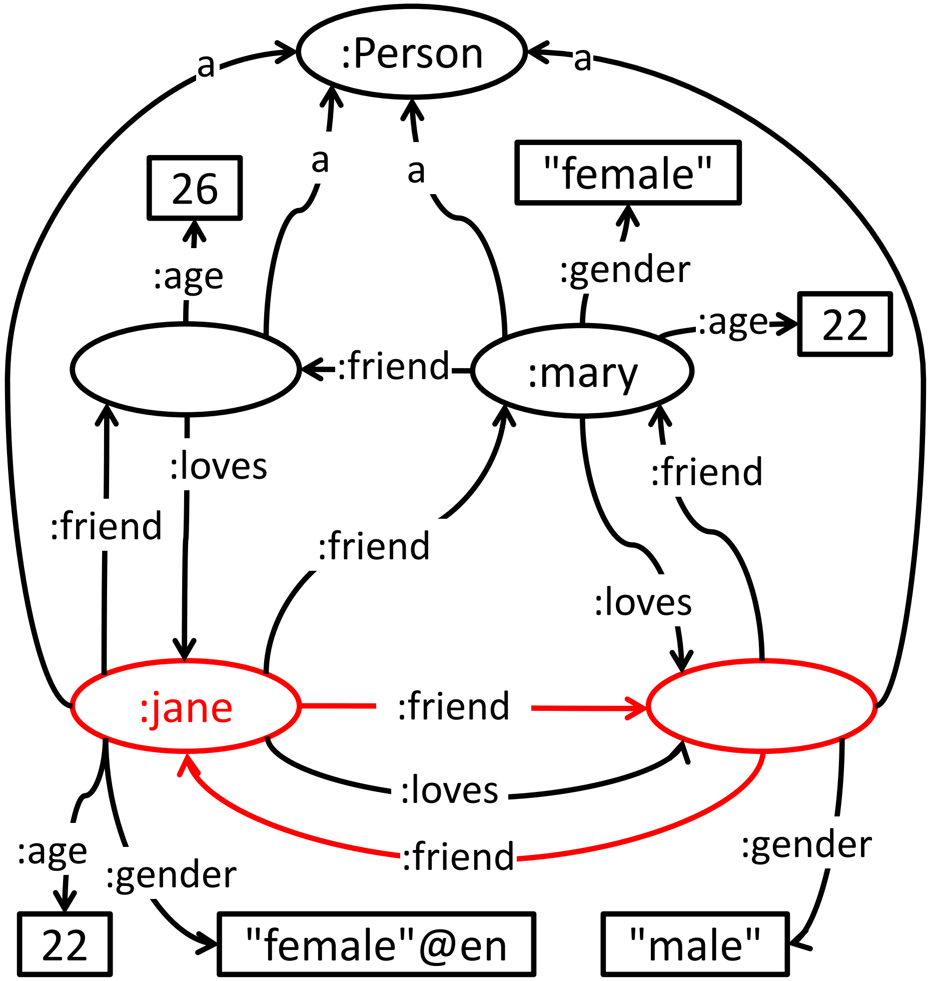
```
      ----------
      | y      |
      ==========
      | :jane  |
      | :mary  |
      | _:b0   |
      ----------
```



Blank Node Syntax in Graph Patterns

```
SELECT *
WHERE
  { :jane :friend [ :friend ?y]. }
```

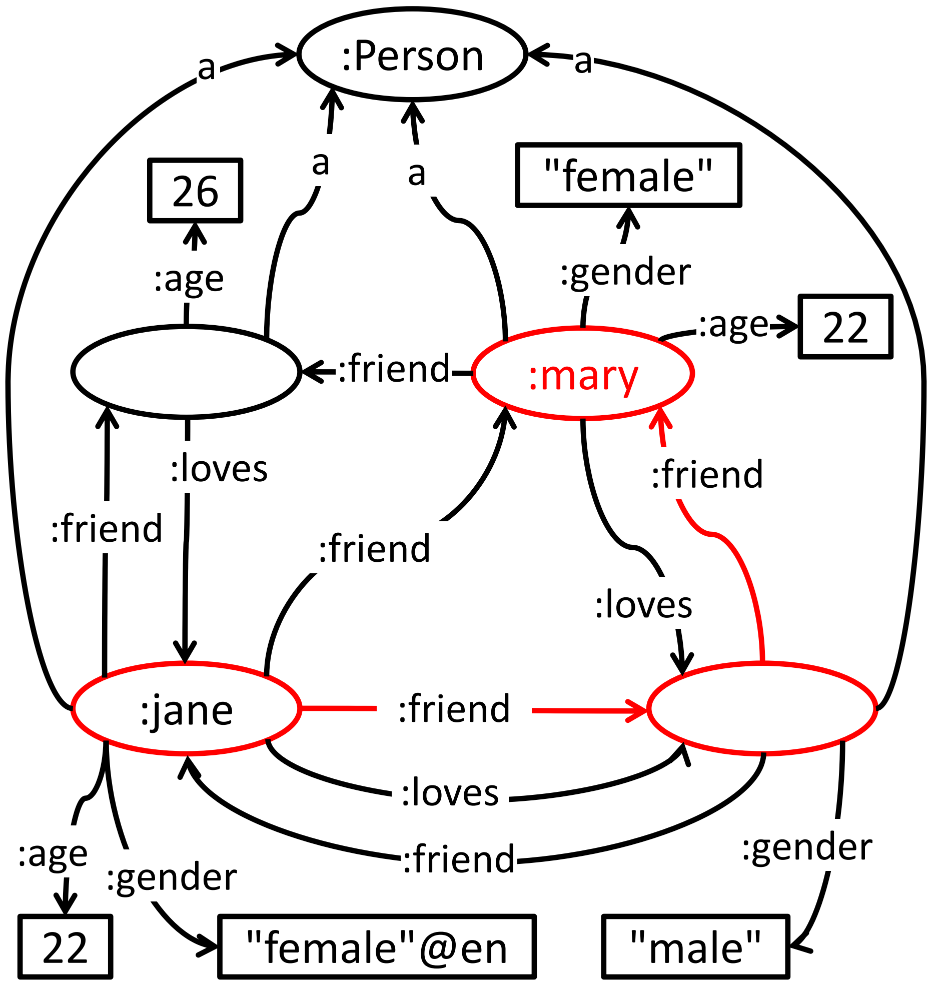
```
      ----------
      | y      |
      ==========
      | :jane  |
      | :mary  |
      | _:b0   |
      ----------
```



Blank Node Syntax in Graph Patterns

```
SELECT *
WHERE
  { :jane :friend [ :friend ?y]. }
```

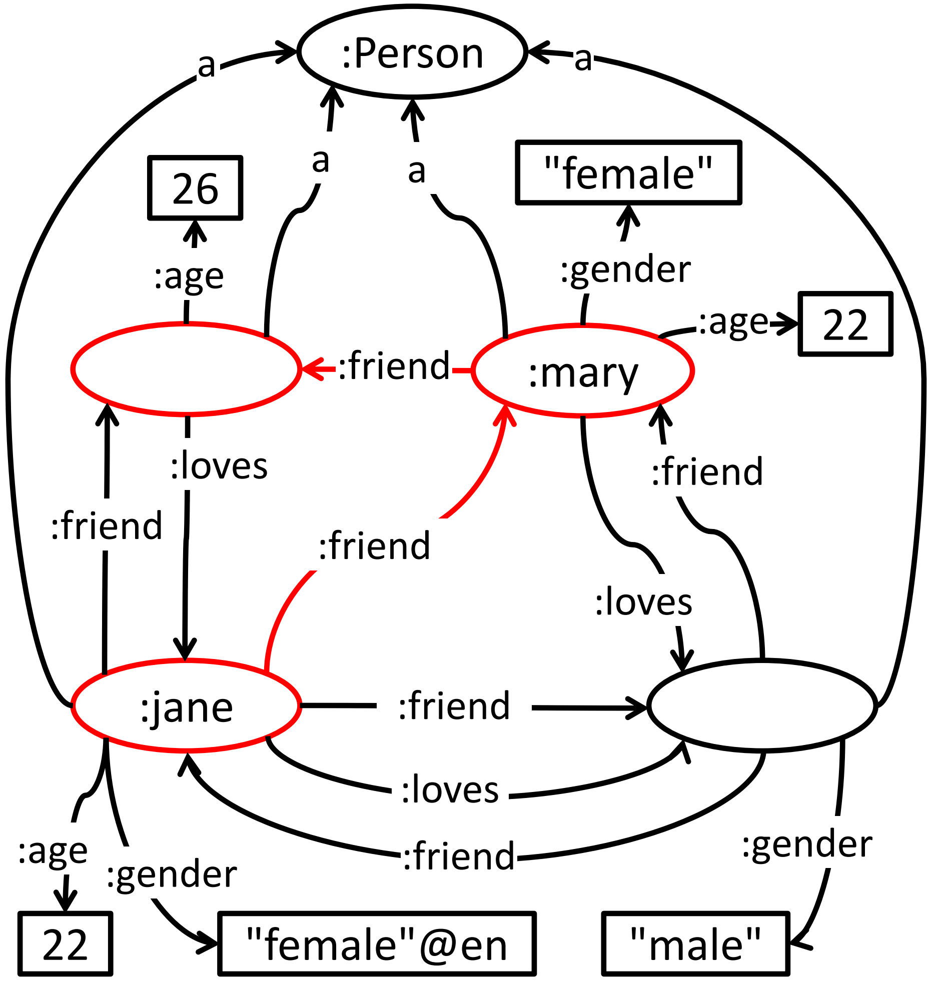
```
  ----------
  | y      |
  ==========
  | :jane  |
  | :mary  |
  | _:b0   |
  ----------
```



Blank Node Syntax in Graph Patterns

```
SELECT *
WHERE
   { :jane :friend [ :friend ?y]. }
```

```
----------
| y      |
==========
| :jane  |
| :mary  |
| _:b0   |
----------
```



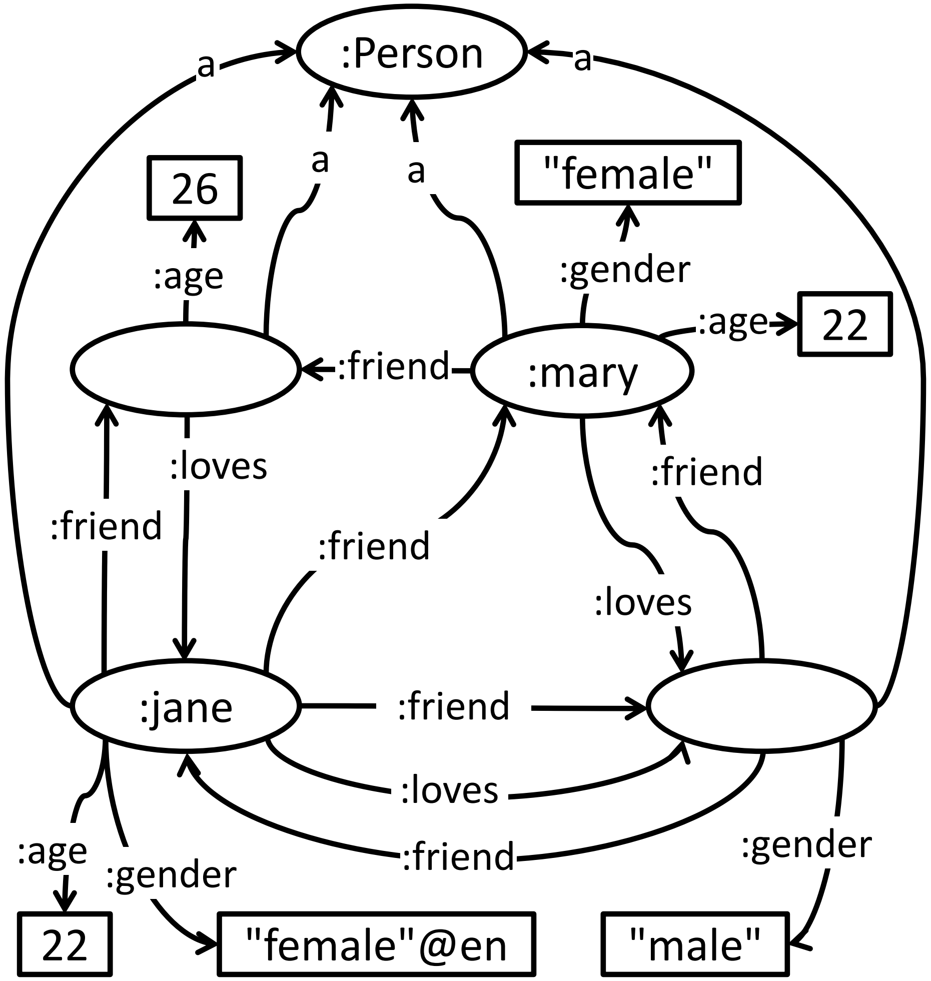
Blank Node Syntax in Graph Patterns

```
SELECT *
WHERE
  { :jane :friend [ :friend ?y]. }
```

```
      ----------
      | y      |
      ==========
      | :jane  |
      | :mary  |
      | _:b0   |
      ----------
```

```
SELECT ?y
WHERE
  { :jane :friend ?x.
    ?x     :friend ?y
    FILTER isBlank(?x)  }
```

```
      ----------
      | y      |
      ==========
      | :jane  |
      | :mary  |
      ----------
```



Summary and Outlook

- Today we covered most of the **SPARQL 1.1 Query Language** W3C Recommendation.
- See http://www.w3.org/TR/sparql11-query/ for further details, especially concerning functions and operators to be used in expressions: http://www.w3.org/TR/sparql11-query/#expressions

- In the coming lectures we will also discuss:
  - SPARQL Queries over RDF Datasets, Named Graphs
  - SPARQL Update

SPARQL 1.1 Overview

- SPARQL 1.1 is a set of specifications that provide languages and protocols to **query and manipulate RDF** graph content **on the Web or in an RDF store**. The standard comprises the following specifications:
  - SPARQL 1.1 **Query Language** - A query language for RDF.
  - SPARQL 1.1 **Query Results JSON Format** and **SPARQL 1.1 Query Results CSV and TSV** Formats - Apart from the standard SPARQL Query Results XML Format [SPARQL-XML-Result], SPARQL 1.1 now allows three alternative popular formats to exchange answers to SPARQL queries, namely JSON, CSV (comma separated values) and TSV (tab separated values) which are described in these two documents.
  - SPARQL 1.1 **Federated Query** - A specification defining an extension of the SPARQL 1.1 Query Language for executing queries distributed over different SPARQL endpoints.
  - SPARQL 1.1 **Entailment Regimes** - A specification defining the semantics of SPARQL queries under entailment regimes such as RDF Schema, OWL, or RIF.
  - SPARQL 1.1 **Update Language** - An update language for RDF graphs.
  - SPARQL 1.1 **Protocol for RDF** - A protocol defining means for conveying arbitrary SPARQL queries and update requests to a SPARQL service.
  - SPARQL 1.1 **Service Description** - A specification defining a method for discovering and a vocabulary for describing SPARQL services.
  - SPARQL 1.1 **Graph Store HTTP Protocol** - As opposed to the full SPARQL protocol, this specification defines minimal means for managing RDF graph content directly via common HTTP operations.
  - SPARQL 1.1 **Test Cases** - A suite of tests, helpful for understanding corner cases in the specification and assessing whether a system is SPARQL 1.1 conformant

see: http://www.w3.org/TR/sparql11-overview/