Multi-Edge support in RDFn

Souripriya Das, Ph.D.

Architect, Spatial and Graph,
Oracle America Inc.

December 02, 2022

Souripriya Das (Souri)

https://www.linkedin.com/in/souripriya-souri-das-ph-d-48801911/

Architect at Oracle

- Database
- RDF Knowledge Graph
- Property Graph

Education

- Ph.D., Rutgers University
- M.S., Vanderbilt University
- B.Tech., Indian Institute of Technology (IIT), Kharagpur

Standards Activity

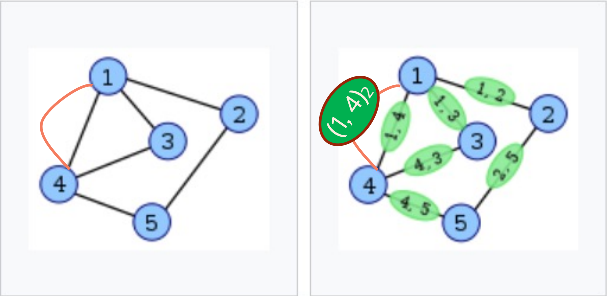
- W3C RDB2RDF, Editor of R2RML
- W3C SPARQL 1.0 and 1.1
- W3C RDF 1.1

Publications in Database, Semantic Web, Knowledge Graphs

- ICDE, VLDB, EDBT, CIKM, KGC
- Patents in Database and Graph technologies

Background: Edges as Vertices, Multigraph, Multi-Edge (or Parallel Edges)

- A Line Graph (LG) converts edges to vertices and then eliminates the original vertices and …



Graph G

Vertices in L(G) constructed from edges in G

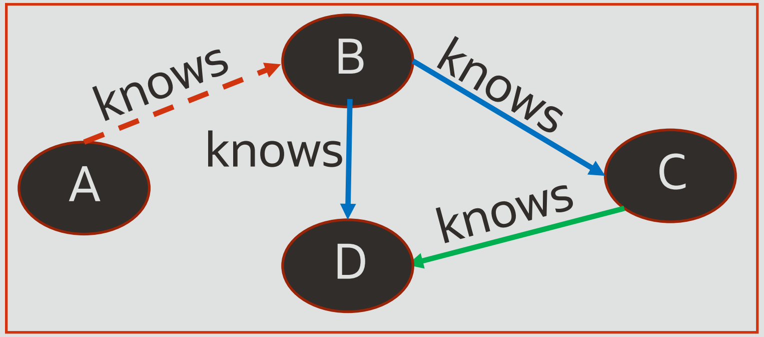
What if G is a multigraph?

Two parallel edges between vertices 1 and 4.

Both edges cannot be named (1, 4). A custom name, e. g., $(1, 4)_2$, may be used.

- RDF-star, keeps, and allows unrestricted use of, both edge-vertices and the original vertices, as vertices.
- Property Graph (PG) supports it too, but limits edge-vertices to only connect *to* scalar values.

Edge-as-Vertex: Rel./SQL vs. Turtle-star/SPARQL-star vs. Turtlen/SPARQLn



Relational

x	y	color	type
A	B	red	--
B	C	blue	__
B	D	blue	__
C	D	green	__

knows

Expected Result

Turtle-star

```
:A :knows :B {|
    :color "red" ; :type "--" |} .
:B :knows :C {|
    :color "blue" ; :type "__" |} .
:B :knows :D {|
    :color "blue" ; :type "__" |} .
:C :knows :D {|
    :color "green" ; :type "__" |} .
```

Turtlen

```
:A :knows :B {|
    :color "red" ; :type "--" |} .
:B :knows :C {|
    :color "blue" ; :type "__" |} .
:B :knows :D {|
    :color "blue" ; :type "__" |} .
:C :knows :D {|
    :color "green" ; :type "__" |} .
```

Query

Find who knows whom and
in what color and dash type.

SQL

```
SELECT x, y,
   color, type
FROM knows;
```

SPARQL-star

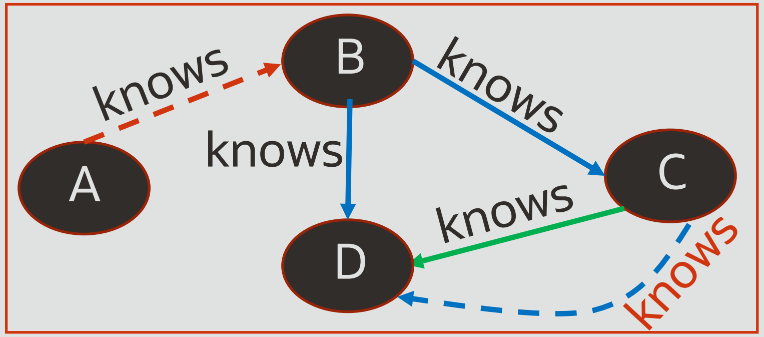
```
select ?x ?y ?color ?type {
?x :knows ?y {|
     :color ?color ; :type ?type |}
}
```

SPARQLn

```
select ?x ?y ?color ?type {
?x :knows ?y {|
     :color ?color ; :type ?type |}
}
```

Adding a Parallel Edge (to create a Multi-Edge)

Add a parallel edge



Relational

x	y	color	type
A	B	red	--
B	C	blue	__
B	D	blue	__
C	D	green	__
C	**D**	blue	--

knows

Turtle-star

```
:A :knows :B {|
    :color “red” ; :type “--” |} .
:B :knows :C {|
    :color “blue” ; :type “__” |} .
:B :knows :D {|
    :color “blue” ; :type “__” |} .
:C :knows :D {|
    :color “green” ; :type “__” |} .
```

```
:C :knows :D {| :occursAs  :cd2 |}
:cd2 :color “blue” ; :type “--” .
```

Turtlen

```
:A :knows :B {|
    :color “red” ; :type “--” |} .
:B :knows :C {|
    :color “blue” ; :type “__” |} .
:B :knows :D {|
    :color “blue” ; :type “__” |} .
:C :knows :D {|
    :color “green” ; :type “__” |} .
```

```
:C :knows :D | :cd2 {|
    :color “blue” ; :type “--” |} .
```

Expected Result

Query

Find who knows whom and in what color and dash type.

SQL

```
SELECT x, y,
   color, type
FROM knows;
```

no changes

SPARQL-star

```
select ?x ?y ?color ?type {
{ ?x :knows ?y {|
    :color ?color ; :type ?type |} }
UNION
{ ?x :knows ?y {| :occursAs ?occ2 |}
  ?occ2 :color ?color ; :type ?type } }
```

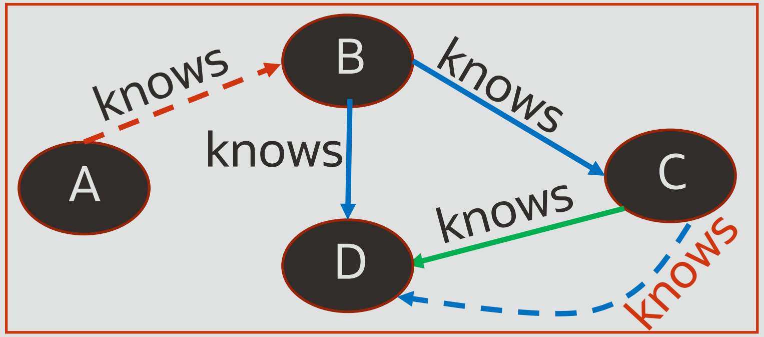
SPARQLn

```
select ?x ?y ?color ?type {
?x :knows ?y {|
    :color ?color ; :type ?type |}
}
```

no changes

Multi-Edge handling using Explicit Names or Occurrences only

Add a parallel edge



Expected Result

Relational

x	y	color	type
A	B	red	--
B	C	blue	__
B	D	blue	__
C	D	green	__
C	**D**	blue	--

knows

Turtle-star

```
:A :knows :B {|
    :color “red” ; :type “--” |} .
:B :knows :C {|
    :color “blue” ; :type “__” |} .
:B :knows :D {|
    :color “blue” ; :type “__” |} .
:C :knows :D {|
   :occursAs :cd1, :cd2 |} .
```

```
:cd1 :color “green” ; :type “__” .
:cd2 :color “blue”   ; :type “--” .
```

Turtlen

```
:A :knows :B {|
    :color “red” ; :type “--” |} .
:B :knows :C {|
    :color “blue” ; :type “__” |} .
:B :knows :D {|
    :color “blue” ; :type “__” |} .
:C :knows :D | (:cd1, :cd2).
```

```
:cd1 :color “green” ; :type “__” .
:cd2 :color “blue”   ; :type “--” .
```

Query

Find who knows whom and
in what color and dash type.
Also, return the name or occ. id.

SQL

```
SELECT rowid,x, y,
   color, type
FROM knows;
```

SPARQL-star

```
select ?occ ?x ?y ?color ?type {
?x :knows ?y {| :occursAs ?occ |}
?occ :color ?color ; :type ?type }
```

SPARQLn

```
select ?n ?x ?y ?color ?type {
?x :knows ?y | ?n {|
     :color ?color ; :type ?type |}
}
```

Labeled Multidigraphs are Not Uncommon in Practice

Examples:

- :servedAs :POTUS” (Cleveland Grover did two non-consecutive terms)
- :deposit :myBankAccount (multiple transactions by same person to same account)
- :called :mySister (call data records: multiple calls by same person to his/her sister)
- :hasManager :myManager (multiple stints)
- :won :Wimbledon (same person wins multiple times)
- :won :SoccerWorldCup (same country wins multiple times)
- etc.

It will be great to incorporate seamless support for this in the RDF-star Recommendation.

RDFn Semantics: Essentials, in a few words

- Every RDFn statement
  - has a unique name
  - the name can be an implicit (auto-generated) name or an explicit name
  - is represented by the tuple <s, p, o, n>

- An RDFn dataset is a set of <s, p, o, n> tuples where each distinct s-p-o triple …
  - must be associated with at least one name
  - at most one of its names can be an implicit name
  - it may be associated with 0 or more explicit names

RDFn Semantics: Essentials Beyond RDF

An RDFn *statement* is a tuple of the form: <s, p, o, n> where n is:

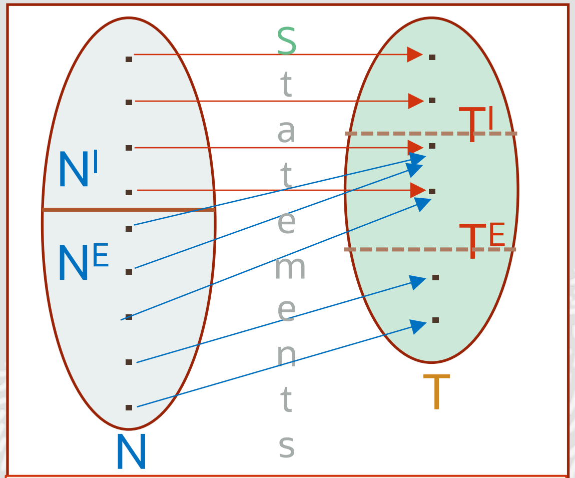
- either an implicit (auto-generated) name, n_i, that is an IRI in an exclusive namespace (e.g., rdfn: ..)
- or an explicit (assigned) name, n_e, that is an IRI, *not* in the above namespace, or is a blank node
- n may be used as subject or object of other triples (provided its use causes no *name defn. cycle*)

Suppose, for a given RDF dataset

- N is the set of names and T is the set of triples, and
- N^I and N^E are the sets of implicit and explicit names, resp., and
- T^I and T^E are the sets of implicitly and explicitly named triples, resp.

Then, the following must hold:

- $N = N^I \cup N^E$ and $T = T^I \cup T^E$
- $N^I \cap N^E = \Phi$ (Note: $T^I \cap T^E$ need not be empty. See diagram →.)
- N^I and T^I are related by one-to-one correspondence
- N^E to T^E mapping is injective.
- ➔ N to T is injective ➔ Every statement, <s, p, o, n>, has a unique (explicit or implicit) name.



Statements: Names ➔Triples

Merging of Datasets: Validating the Name Uniqueness Constraint

- Merging of RDFn datasets must prevent potential violation of the name uniqueness constraint
  - if violated, the same **explicit** name may get associated with multiple distinct s-p-o triples
  - in that case, edge-properties of multiple statements may get combined.
    Example:
    - dataset 1 => :John :depositedTo :Acct1 | :n {| :amount 100 |}
    - dataset 2 => :Mary :depositedTo :Acct2 | :n {| :amount 200 |}
    - merging these two datasets without checking the constraint causes the same name :n
      - to get associated with two distinct triples and
      - to have two edge-properties that lose their associations with the individual statements

- Note: This uniqueness constraint applies to implicit names as well. In that case, however, it is equivalent to the original s-p-o uniqueness constraint in RDF. This is guaranteed by ensuring that the implicit names generated for distinct s-p-o triples are always different.

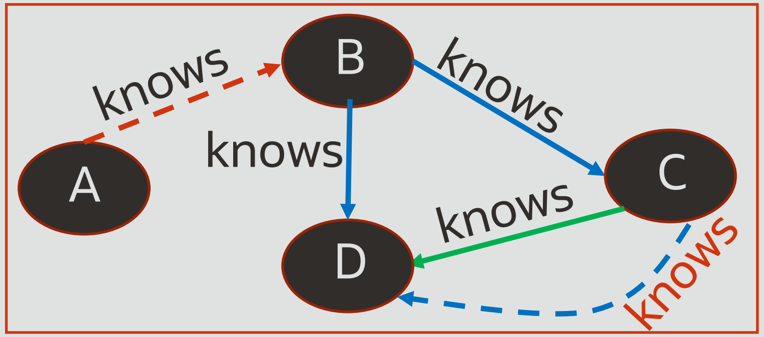
Federated Query: Returning Implicit Names

- A triplestore has **local autonomy** regarding how it creates implicit names. (It is assumed, however, that implicit names can be distinguished from explicit names.)
- When a SERVICE query must return a binding that happens to be an implicit name, it needs to instead return the triple associated with the implicit name.

TripleStore 1

```
:A :knows :B {| :color "red" |} .
:B :knows :C {| :color "blue" |} .
:B :knows :D {| :color "blue" |} .
:C :knows :D | (:cd1, :cd2) .
```

```
:cd1 :color "green" .
:cd2 :color "blue" .
```



```
:A :knows :B {| :type "--" |} .
:B :knows :C {| :type "__" |} .
:B :knows :D {| :type "__" |} .
:C :knows :D | (:cd1, :cd2) .
```

```
:cd1 :type "__" .
:cd2 :type "--" .
```

TripleStore 2

Federated Query issued at TripleStore1

```
select ?n ?x ?y ?color {
  ?x :knows ?y {| :color ?color |} .
  SERVICE :TripleStore2
   { ?x :knows ?y | ?n {| :type "__" |} } }
```

find color of solid edges

Results received from TripleStore2

```
[ ?n = (:B :knows :C), ?x = :B, ?y = :C ]
[ ?n = (:B :knows :D), ?x = :B, ?y = :D ]
[ ?n = (:C :knows :D), ?x = :C, ?y = :D ]
```

Query returns following after "localization"

```
[ ?n = rdfn:_1, ?x = :B, ?y = :C, ?color = "blue" ]
[ ?n = rdfn:_2, ?x = :B, ?y = :D, ?color = "blue" ]
[ ?n = rdfn:_3, ?x = :C, ?y = :D, ?color = "green" ]
```

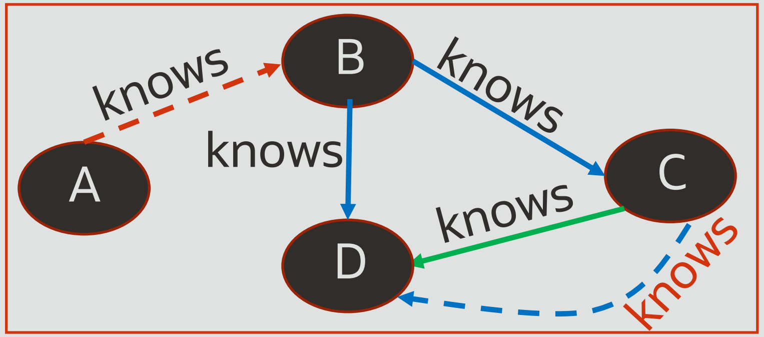
Federated Query: Returning Explicit Names

- It is possible that the same explicit name may be associated with different s-p-o triples in different triplestores.
- When a SERVICE query must return a binding that happens to be an explicit name, it needs to return the corresponding triple as well. This helps in recognizing a name uniqueness violation.

TripleStore 1

```
:A :knows :B {| :color “red” |} .
:B :knows :C {| :color “blue” |} .
:B :knows :D {| :color “blue” |} .
:C :knows :D | (:cd1, :cd2) .

:cd1 :color “green” .
:cd2 :color “blue” .
```



TripleStore 2

```
:A :knows :B {| :type “--” |} .
:B :knows :C {| :type “__” |} .
:B :knows :D {| :type “__” |} .
:C :knows :D | (:cd1, :cd2) .

:cd1 :type “__” .
:cd2 :type “--” .
```

Federated Query issued at TripleStore1

```
select ?n ?x ?y ?color {
  ?x :knows ?y {| :color ?color |} .
  SERVICE :TripleStore2
   { ?x :knows ?y | ?n {| :type “--" |} } }
```

find color of dotted edges

Results received from TripleStore2

```
[ ?n = (:A :knows :B), ?x = :A, ?y = :B ]
[ ?n = (:C :knows :D | :cd2 )
  , ?x = :C, ?y = :D ]
```

Query returns following after “localization”

```
[ ?n = rdfn:_1, ?x = :A, ?y = :B, ?color = “red” ]
[ ?n = :cd2, ?x = :C, ?y = :D, ?color = “blue” ]
```

if conflict: ?n = :TripleStore2#name=:cd2, …]

Connecting or Isolating Resources (or Names) in Different TripleStores

RDF:

- Use of IRIs for Resources
  - Benefit: Allows sharing of resources across multiple triplestores.
  - Drawback: Accidental sharing is a risk. (e.g., :JohnSmith in two triplestores).
- Use of blank nodes for Resources
  - Benefit: Allows isolating resources to a local triplestore.
  - Drawback: Prevents any form of sharing.

RDFn

- Use of IRIs for Explicit Names
  - Same benefits and drawbacks as in RDF case.
- Use of blank nodes for Explicit Names
  - Same benefits and drawbacks as in RDF case.

Enabling Explicit Naming in RDF-star, Serializations, and SPARQL-star

- RDF-star
  - Allow inclusion of explicit name in the def. of a statement: <s, p, o> → <s, p, o, n>
  - Add the name uniqueness constraint
- Serialization Formats: N-Triple/N-Quad, Turtle/TriG, RDF/XML, JSON-LD
  - Extend syntax to allow explicit name specification.
  - Ex (N-Triple): :John :spouseOf :Mary | :JsM .
- SPARQL-star Query and Federated (SERVICE) Query
  - extend syntax to include name or name variable, and
  - add new functions: isName(<var>), isImplicitName(<var>), isExplicitName(<var>)
  - Extend remote query response to include the triple when returning name as value
    Ex: [?n = (:John :spouseOf :Mary | :JsM), …] (instead of just [?n = :JsM, …])
- SPARQL-star Update
  - INSERT → name uniqueness constraint violation?. DELETE → CASCADE?

ORACLE