How (well) do Datalog, SPARQL and RIF interplay?

Axel Polleres, Siemens AG Österreich
What have You heard about “Semantic Web Standards”?

- Many of you have probably heard about mostly OWL and Description Logics… not today.
- … in fact two other W3C standards are probably much closer to Datalog:
  - SPARQL – RDF Query language
  - RIF – Rule Interchange Format
- In this Tutorial:
  - How close are they to Datalog, where do they differ?
Semantic Web Standards?

- RDF and Datalog
- SPARQL and Datalog
- RIF and Datalog
- SPARQL1.1 and Datalog - An Outlook

```
dbpedia:Vienna dbpedia-ont:country dbpedia:Austria .
dbpedia:Vienna rdfs:label "Wien"@de .
_:_x foaf:name "Reinhard Pichler" .
_:_x foaf:based_near dbpedia:Vienna .
```

Blanknodes: “existential variables in the data” to express incomplete information, written as _:_x or []

URIs, e.g.
- http://xmlns.com/foaf/0.1/name
- http://dbpedia.org/resource/Vienna
- http://dbpedia.org/resource/Austria

Various syntaxes, RDF/XML, Turtle, N3, RDFa,...

Subject U U B
x
Predicate U
x
Object U U B U L

Literals, e.g.
- "2012"^^xsd:gYear
- "Wien"@de
- "Vienna"@en
- "Reinhard Pichler"
RDF – Adoption

RDF in Datalog? (Almost) No problem

dbpediares:Vienna  dbpedia-ont:country  dbpediares:Austria .
dbpediares:Vienna  rdfs:label  "Wien"@de .
_:x  foaf:name  "Reinhard Pichler" .
_:x  foaf:based_near  dbpediares:Vienna .

\[ \exists X \]

\[
\text{triple(vienna, country, austria)} \land \\
\text{triple(vienna, label, "Wien"@de)} \land \\
\text{triple(x, name, "ReinhardPichler")} \land \\
\text{triple(x, based_near, vienna)}
\]

What about Blank nodes? …

… let’s just use local constants ("Skolemize")
RDF in Datalog? (Almost) No problem

dbpediares:Vienna dbpedia-ont:country dbpediares:Austria.
dbpediares:Vienna rdfs:label "Wien"@de.
_:x foaf:name "Reinhard Pichler".
_:x foaf:based_near dbpediares:Vienna.

EDB:

triple( vienna, country, austria ).
triple( vienna, label, "Wien"@de ).
triple( b1, name, "Reinhard Pichler").
triple( b1, based_near, vienna).

What about Blank nodes? …

… let’s just use local constants ("Skolemize")
Even RDF Schema (RDFS) is easy with Datalog …

- lightweight ontology language to infer new implicit information from RDF
- formal semantics [W3C, 2004]
- can be captured by Datalog style rules [W3C, 2004 §7], e.g. …

\[
\text{dbpediares:Vienna} \text{ dbpedia-ont:country} \text{ dbpediares:Austria} .
\text{dbpediares:Vienna} \text{ rdfs:label} "\text{Wien}@de" .
\text{_:x foaf:name} "\text{Reinhard Pichler}" .
\text{_:x foaf:based_near} \text{dbpediares:Vienna} .
\]

\[
\text{dbpediares:Austria} \text{ rdf:type} \text{ dbpedia-owl:Country} .
\text{_:x rdfs:label} "\text{Reinhard Pichler}" .
\]

\[
\text{foaf:name} \text{ rdfs:subPropertyOf} \text{ rdfs:label} .
\text{dbpedia-ont:country} \text{ rdfs:range} \text{ dbpedia-owl:Country} .
\]

• formal semantics [W3C, 2004]
• can be captured by Datalog style rules [W3C, 2004 §7], e.g. …

\[
\begin{array}{|c|c|c|}
\hline
\text{rdfs3} & \text{aaa rdfs:range} & \text{XXX} .
\text{uuu aaa vvv} .
\text{VVV rdf:type} & \text{XXX} .
\
\text{rdfs7} & \text{aaa rdfs:subPropertyOf} & \text{bbb} .
\text{uuu aaa yyy} .
\text{uuu bbb yyy} .
\hline
\end{array}
\]

• … with some caveats [ter Horst, 2005], [Muñoz+, 2009]
Even RDF Schema (RDFS) is easy with Datalog…

RDFS:
• lightweight ontology language to infer new implicit information from RDF
• formal semantics [W3C, 2004]
• can be captured by Datalog style rules [W3C, 2004 §7], e.g. …

```
triple( O, rdf:type, C) :- triple( P, rdf:range, C), triple(S,P,O) .
triple( S, Q, O) :- triple( P, rdfs:subPropertyOf, Q), triple(S,P,O) .
```

• … with some caveats [ter Horst, 2005], [Muñoz+, 2009]
RDF Schema 2/2 – RDF(S) Entailment

Core problem described in RDF Semantics document is RDF(S) Entailment [W3C, 2004]

\[ G_1 \models_{RDFS} G_2 \]

Is there a blank node homomorphism \( \mu \) from \( G_2 \) to \( G_1 \) such that

\[ \mu(G_2) \subseteq Cl_{RDFS}(G_1) \]

RDFS Entailment checking can be easily done in Datalog [Bruijn&Heymans,2007], [Muñoz+, 2009] , [Ianni+, 2009], cf. also [Gutierrez+,2011].

1) Encode \( G_1 \) + RDFS Entailment rules in Datalog EDB+IDB

2) Encode \( G_2 \) as boolean conjunctive query

EDB (G1)

\[
\text{triple}(x, \text{name}, "Reinhard") .
\text{triple}(\text{name}, \text{rdfs:subPropertyOf}, \text{label}).
\]

IDB (RDFS)

\[
\text{triple}(S, Q, O) :- \text{triple}(P, \text{rdfs:subPropertyOf}, Q), \text{triple}(S, P, O) .
\]

...  

Query

\[
\text{answer} :- \text{triple}(X, \text{name},"Reinhard"), \text{triple}(Y, \text{label}, "Reinhard")
\]
Now how to query RDF?

SPARQL1.0 [W3C, 2008]
in a Nutshell...

... i.e.,
nonrecursive Datalog\textsuperscript{not}
in a Nutshell...

[Angles, Gutierrez, 2008]
SPARQL + Linked Data give you Semantic search almost “for free”

Query: Scientists born in Vienna? (Conjunctive Query)
How’d we do it in SQL?

```
SELECT t1.s
FROM triple t1, triple t2
WHERE t1.s = t2.s AND t1.p = dbpedia:birthPlace AND t1.o = Vienna
    AND t2.p = rdf:type AND t2.o = dbpedia:Scientist
```

Obviously, we know how to do that in Datalog...

```
answer(X) :-
    triple( X, birthPlace, Vienna ),
    triple( X, type, Scientist ).
```
SPARQL + Linked Data give you Semantic search almost “for free”

Query: Scientists born in Vienna? (Conjunctive Query)

Now how does it look in SPARQL?

```
SELECT ?X
WHERE {
  ?X dbpedia:birthPlace <dbpedia.org/resource/Vienna> .
  ?X rdf:type dbpedia:Scientist.
}
```

Obviously, we know how to do that in Datalog...

```
answer(X) :-
  triple( X, birthPlace , Vienna ) ,
  triple( X, type , Scientist ) .

... and SPARQL looks quite similar!
SPARQL – Standard RDF Query Language and Protocol

SPARQL 1.0 (2008):

- SQL “Look-and-feel” for the Web
- Essentially “graph matching” by basic graph patterns (BGPs)
- Allows conjunction (.) , disjunction (UNION), optional (OPTIONAL) patterns and filters (FILTER)
- Construct new RDF from existing RDF (CONSTRUCT)
- Solution modifiers (DISTINCT, ORDER BY, LIMIT, …)
- A standardized HTTP based protocol:

```sql
SELECT ?X
WHERE {
  ?X dbpedia:birthPlace <dbpedia.org/resource/Vienna> .
  ?X rdf:type dbpedia:Scientist.
}
```
Definition 1:

The evaluation of the BGP P over a graph G, denoted by eval(P,G), is the set of all mappings $\mu: \text{Var} \rightarrow V(G)$ such that:

- $\text{dom}(\mu)$ is exactly the set of variables occurring in P and
- $\mu(P) \subseteq G$ (actually, in the official W3C spec it is rather $G \models_{\text{RDF}} \mu(P)$).

Example RDF Graph (G):

```plaintext
:tim foaf:knows :jim .
:jim foaf:knows :tim .
:jim foaf:knows :juan .
```

Example Pattern (P):

```
```

$\text{eval}(P,G) = \{ \mu_1 = \{ ?x \rightarrow \text{:tim}, ?y \rightarrow \text{:jim}, ?z \rightarrow \text{:tim} \},$

$\mu_2 = \{ ?x \rightarrow \text{:tim}, ?y \rightarrow \text{:jim}, ?z \rightarrow \text{:juan} \} \}$
Definition 2: mappings $\mu_1, \mu_2$ are compatible iff they agree in their shared variables.

Let $M_1, M_2$ be sets of mappings

Definition 3:

Join:
$M_1 \bowtie M_2 = \{ \mu_1 \cup \mu_2 : \mu_1 \in M_1, \mu_2 \in M_2, \text{ and } \mu_1, \mu_2 \text{ are compatible} \}$

Union:
$M_1 \cup M_2 = \{ \mu : \mu \in M_1 \text{ or } \mu \in M_2 \}$

Diff:
$M_1 \setminus M_2 = \{ \mu \in M_1 : \forall \mu' \in M_2, \mu \text{ and } \mu' \text{ are not compatible} \}$

LeftJoin:
$M_1 \bowtie M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

Filter:
$M|_R = \{ \mu : \mu \in M \text{ and } \mu(R) = \text{true} \}$
Semantics full as per [Perez et al.2006]

\[
eval(BGP, G) \quad \ldots \text{ see Definition 1}
\]

\[
eval(P_1 . P_2, G) = \eval(P_1, G) \otimes \eval(P_2, G)
\]

\[
eval(P_1 \text{ UNION } P_2, G) = \eval(P_1, G) \cup \eval(P_2, G)
\]

\[
eval(P_1 \text{ OPTIONAL } P_2, G) = \eval(P_1, G) \otimes \eval(P_2, G)
\]

\[
eval(P \text{ FILTER } R, G) = \eval(P, G) \mid_R
\]

**Example \otimes:**

\[
P = \{ \ ?X \text{ foaf:knows } \ ?Y . \ ?Y \text{ foaf:knows } \ ?Z \}
\]

\[
eval(P_1, G) \otimes \eval(P_2, G) =
\]

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>tim</td>
<td>jim</td>
<td></td>
</tr>
<tr>
<td>jim</td>
<td>tim</td>
<td></td>
</tr>
<tr>
<td>jim</td>
<td>juan</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccc|}
X & Y & Z & X & Y & Z \\
\mid & \otimes & = & \mid & \otimes & = \\
\mid & \mid & \mid & \mid & \mid & \mid \\
\mid & \mid & \mid & \mid & \mid & \mid \\
\mid & \mid & \mid & \mid & \mid & \mid \\
\end{array}
\]

\[
\begin{array}{cccc}
X & Y & Z \\
\mid & \mid & \mid \\
\mid & \mid & \mid \\
\mid & \mid & \mid \\
\end{array}
\]

\[
\begin{array}{cccc}
X & Y & Z \\
\mid & \mid & \mid \\
\mid & \mid & \mid \\
\mid & \mid & \mid \\
\end{array}
\]
Back to “real” SPARQL examples: UNION

Example RDF Graph:

```plaintext
```

Example Query:

```sql
SELECT ?X
WHERE {
  { :jim foaf:knows ?X }
  UNION
  { :jim foaf:worksWith ?X }
}
```

```
X
| tim |
U
| X   |
|   |
| X   |
|   |
| tim | =
|   |
| juan|
|   |
```
Back to “real” SPARQL examples: UNION

Example RDF Graph in Datalog EDB:

```
triple( :tim, knows, :jim ).
triple( :jim, knows, :tim ).
triple( :jim, worksWith, :juan ).
```

In Datalog:

```
answer(X) :- evalP(X).
evalP(X) :-
  triple( :jim, knows, X ).
evalP(X) :-
  triple( :jim, worksWith, X ).
```

```
<table>
<thead>
<tr>
<th>X</th>
<th>U</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>tim</td>
<td></td>
<td>juan</td>
</tr>
</tbody>
</table>
```

© Siemens AG Österreich 2012. All rights reserved.
Siemens AG Österreich
Back to “real” SPARQL examples: UNION

Example RDF Graph:

:tim foaf:knows :jim .
:jim foaf:knows :tim .
:jim :worksWith :juan .

Example Query:

```
SELECT ?X ?Y
WHERE {
  { :jim foaf:knows ?X }
  UNION
  { :jim foaf:worksWith ?Y }
}
```

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>tim</td>
<td>null</td>
</tr>
<tr>
<td>null</td>
<td>juan</td>
</tr>
</tbody>
</table>

© Siemens AG Österreich 2012. All rights reserved.
Siemens AG Österreich
Back to “real” SPARQL examples: UNION

Example RDF Graph:

```
triple( :tim, knows, :jim ) .
triple( :jim, knows, :tim ) .
triple( :jim, worksWith, :juan ) .
```

Example Query:

```
answer(X,Y) :- evalP(X,Y).
evalP(X,null) :-
    triple( :jim, knows, X ) .
evalP(null,Y) :-
    triple( :jim, worksWith, Y ) .
```

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>tim</td>
<td>null</td>
</tr>
<tr>
<td>null</td>
<td>Juan</td>
</tr>
</tbody>
</table>
Back to “real” SPARQL examples: OPTIONAL

Give me people who know somebody and OPTIONALLY their email address:

:tim foaf:knows :jim .  :tim :email <mailto:timbl@w3.org> .
:jim foaf:knows :tim .
:jim :worksWith :juan .

Example Query:

```
SELECT ?X ?M
WHERE {
  { ?X foaf:knows ?Y }
  OPTIONAL
  { ?X :email ?M }
}
```

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>tim</td>
<td>jim</td>
</tr>
<tr>
<td>jim</td>
<td>tim</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>tim</td>
<td><a href="mailto:timbl@w3.org">timbl@w3.org</a></td>
</tr>
<tr>
<td>jim</td>
<td></td>
</tr>
</tbody>
</table>

© Siemens AG Österreich 2012. All rights reserved.
Siemens AG Österreich
Back to “real” SPARQL examples: OPTIONAL

Give me people who know somebody and OPTIONALLY their email address:

```
triple( :tim, knows, :jim ).
triple( :tim, email, timbl@w3.org ).
triple( :jim, knows, :tim ).
triple( :jim, worksWith, :juan ).
```

Example Query:

```
answer(X,M) :- evalP(X,Y,M) .

evalP(X,Y,M) :- triple( X, knows, Y ), triple( X, email, M ).

evalP(X,Y,,null) :- triple( X, knows, Y ), not evalP1(X).
evalP1(X) :- triple( X, email, M ).
```

<table>
<thead>
<tr>
<th>X</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>tim</td>
<td><a href="mailto:timbl@w3.org">timbl@w3.org</a></td>
</tr>
<tr>
<td>jim</td>
<td></td>
</tr>
<tr>
<td>tim</td>
<td><a href="mailto:timbl@w3.org">timbl@w3.org</a></td>
</tr>
<tr>
<td>jim</td>
<td>tim</td>
</tr>
</tbody>
</table>
ATTENTION:

needs some attention!

Eval(P1,G)  Eval(P2,G)

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>d</td>
<td>f</td>
</tr>
</tbody>
</table>

evalP(X,Y,Z) :- evalP1( X, Y ), evalP2( Y, Z ).

Doesn’t work!

Recall (Definition 3):

Join:

\[ M_1 \cup M_2 = \{ \mu_1 \cup \mu_2 | \mu_1 \in M_1, \mu_2 \in M_2, \text{ and } \mu_1, \mu_2 \text{ are compatible} \} \]

Rather:

evalP(X,Y,Z) :- evalP1( X, Y ), evalP2( Y1, Z), join(Y,Y1).

join(X,X) :- HU_G(X).
join(X,null) :- HU_G(X).
join(null(X) :- HU_G(X).

... where HU_G(X) is a predicate defining the Herbrand Universe of G.
FILTERs 1/3

Give me people with an email address where the email doesn’t contain “w3”:

:tim foaf:knows :jim . :tim :email &lt;mailto:timbl@w3.org&gt; .
:jim foaf:knows :tim . :jim :email &lt;mailto:hendler@cs.rpi.edu&gt; .
:jim :worksWith :juan .

Example Query:

```
SELECT ?X ?M
WHERE {
  FILTER( ! Regex(Str(?M), "w3") )
}
```

Complex FILTER expressions allowed ( !, &&, || )

<table>
<thead>
<tr>
<th>X</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>tim</td>
<td><a href="mailto:timbl@w3.org">timbl@w3.org</a></td>
</tr>
<tr>
<td>jim</td>
<td><a href="mailto:hendler@cs.rpi.edu">hendler@cs.rpi.edu</a></td>
</tr>
</tbody>
</table>
FILTERs 2/3

People who know someone & optionally their email where the email doesn’t contain “w3”:

:tim foaf:knows :jim . :tim :email <mailto:timbl@w3.org> .
:juan foaf:knows :jim .

Example Query:

```
SELECT ?X ?M
WHERE {
  ?X foaf:knows ?Y
  OPTIONAL {?X :email ?M} 
  FILTER( !Regex(Str(?M), “W3”) )
}
```

<table>
<thead>
<tr>
<th>X</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>jim</td>
<td><a href="mailto:hendler@cs.rpi.edu">hendler@cs.rpi.edu</a></td>
</tr>
</tbody>
</table>

Note: FILTERs are evaluated under a three-values semantics! (True, False, Error), e.g.
FILTERs 3/3

A special FILTER function is `bound()` – Can be used to "encode" Negation as failure in SPARQL 1.0:

Give me people **without** an email address:

```
SELECT ?X ?M
WHERE {
  ?X foaf:knows ?Y
  OPTIONAL {?X :email ?M .}
  FILTER(! bound(?M))
}
```

What about Datalog?

SPARQL FILTERs can in principle be encoded in Datalog,
- need built-ins (or be pre-compiled for HU_G)
- need to encode three-valued semantics for !, &&, ||
SPARQL 1.0 = nonrecursive Datalog\textsuperscript{not}

[Polleres, 2007] shows that all of SPARQL 1.0 can be translated to (safe) nonrecursive Datalog\textsuperscript{not}.

In fact, [Angles&Gutierrez 2008] vice versa show that (safe) nonrecursive Datalog\textsuperscript{not} likewise be encoded into SPARQL.

PSPACE Program-Complexity for SPARQL 1.0 follows from [Perez et al. 2006] or alternatively [Angles&Gutierrez 2008] + [Dantsin et al. 2001].
Some notable peculiarities about SPARQL1.0 ...
Notable about the official SPEC semantics 1/2
SPARQL allows duplicates!

A slightly modified RDF Graph:

```prolog
triple( :jim, knows, :tim ).
triple( :jim, worksWith, :tim ).
```

Example Query:

```prolog
answer(X, U) :- evalP(X, U).
evalP(X, u1) :-
  triple( :jim, knows, X ) .
evalP(X, u2) :-
  triple( :jim, worksWith, X ) .
```

<table>
<thead>
<tr>
<th>X</th>
<th>Union1</th>
</tr>
</thead>
<tbody>
<tr>
<td>tim</td>
<td>u1</td>
</tr>
<tr>
<td>tim</td>
<td>u2</td>
</tr>
</tbody>
</table>
Notable about the official SPEC semantics 2/2
FILTERS can make OPTIONAL non-compositional!

- “Conditional OPTIONAL”
  - “Give me emails, and the friends only of those whose email contains ‘W3’”

```
SELECT ?N ?F
WHERE{ ?X :email ?M
  OPTIONAL { ?X foaf:knows ?F
                FILTER ( regex( str(?M), "w3" ) ) } }
```

[Angles&Gutierrez, 2008] showed compositional semantics can be achieved by a rewriting, but non-compositional semantics can be actually be directly encoded in Datalog [Polleres&Schindlauer, 2007]…
Adapting [Perez et al. 2006] to match the W3C SPARQL1.0 specification

1) Algebra operations need to be adapted to multiset/bag semantics:

Let $M_1$, $M_2$ be multisets of mappings

**Definition 3:**

Join:
\[ M_1 \bowtie M_2 = \{ \mu_1 \cup \mu_2 | \mu_1 \in M_1, \mu_2 \in M_2, \text{and } \mu_1, \mu_2 \text{ are compatible} \} \]

Union:
\[ M_1 \cup M_2 = \{ \mu | \mu \in M_1 \text{ or } \mu \in M_2 \} \]

Diff:
\[ M_1 \setminus M_2 = \{ \mu | M_1 \text{ and } \mu \not\in M_2 \} \]

LeftJoin:
\[ M_1 \bowtie M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2) \]

Filter:
\[ M|_R = \{ \mu | \mu \in M \text{ and } \mu(R) = \text{true} \} \]

2) non-compositionality of FILTERs in OPTIONAL
Adapting [Perez et al. 2006] to match the W3C SPARQL1.0 specification

\[ \text{eval}(BGP, G) \quad \text{... see Definition 1} \]

\[ \text{eval}(P_1 \cdot P_2, G) = \text{eval}(P_1, G) \times \text{eval}(P_2, G) \]

\[ \text{eval}(P_1 \text{ UNION } P_2, G) = \text{eval}(P_1, G) \cup \text{eval}(P_2, G) \]

\[ \text{eval}(P \text{ FILTER } R, G) = \text{eval}(P, G) \mid_R \]

**eval**(*P_1* **OPTIONAL** \{\(P_2\) **FILTER** \(R\)\}, \(G\)) consists of all \(\mu\) such that:

1. \(\mu = \mu_1 \cup \mu_2\), such that \(\mu_1 \in \text{eval}(P_1, G)\) and \(\mu_2 \in \text{eval}(P_2, G)\) are compatible and \(\mu(R) = \text{true}\), or
2. \(\mu \in \text{eval}(P_1, G)\) and there is no compatible \(\mu_2 \in \text{eval}(P_2, G)\) for \(\mu\), or
3. \(\mu \in \text{eval}(P_1, G)\) and for any compatible \(\mu_2 \in \text{eval}(P_2, G)\), \(\mu \cup \mu_2\) does not satisfy \(R\).
What again about Blank nodes?

Related to duplicates: Notably, blank nodes might also be considered surprising in SPARQL:

1) Blank nodes in the data:
   Two RDF(S)-equivalent graphs can yield different answers, in SPARQL!

SELECT ?X ?Y
FROM G1
WHERE {
  ?X foaf:name ?Y.
}

More on blank nodes [Mallea+,2011]

© Siemens AG Österreich 2012. All rights reserved.
Siemens AG Österreich
What again about Blank nodes?

**Related to duplicates:** Notably, blank nodes might also be considered surprising in SPARQL:

1) Blank nodes in the **data**:
   Two RDF(S)-**equivalent graphs** can yield **different answers**, in SPARQL!

```sparql
SELECT ?X ?Y FROM G2 WHERE {
  _:x foaf:name "Reinhard" .
  _:y foaf:name "Reinhard" .
}
```

More on blank nodes [Mallea+,2011]
What again about Blank nodes?

**Related to duplicates:** Notably, blank nodes might also be considered surprising in SPARQL:

1) Blank nodes in the data:
   Two RDF(S)-**equivalent graphs** can yield **different answers**, in SPARQL!

more on blank nodes [Mallea+, 2011]
What again about Blank nodes?

Related to duplicates: Notably, blank nodes might also be considered surprising in SPARQL:

1) Blank nodes in the data:
Two RDF(S)-equivalent graphs can yield different answers, in SPARQL!

2) Blank nodes in query patterns:
Blank nodes in queries are behaving just like (distinguished) variables

More on blank nodes [Mallea+,2011]
We can encode SPARQL fairly straightforwardly in nonrecursive Datalog\textsuperscript{not}. [Angle&Gutierrez, 2008] Polleres&Schindlauer, 2007

Duplicates a bit tricky, but
• duplicates by UNION can be covered easily
• we may consider projection (SELECT) as postprocessing

Alternative: How about Datalog with bag semantics?
[Singh, et al. 1993][Green+,2007]
However, bag semantics is problematic, even for conjunctive queries (containment undecidable, cf. [Jayram+, 2006])

Other features not encodable directly in Datalog:
LIMIT, ORDER BY, OFFSET
???(Work-Arounds could be thought of, likely not to be very elegant)
The Rule Interchange Format (RIF)

RIF and Datalog

[W3C, 2010]

from http://rossiter-designs.blogspot.co.at/2011/04/reading-is-fun.html
What is RIF?

• RIF is a Rule Interchange Format (XML) to exchange rules
  • different dialects (Core, Basic Logic (RIF-BLD), Production Rules (RIF-PRD))
  • Closest to Datalog: RIF Core

• RIF Core [W3C, 2010a] is (essentially)
  • Positive Datalog
  • With equality (in facts).
  • With a standard library of Built-in functions and predicates (RIF-DTB), [W3C, 2010b]
  • Interplays well with RDF+OWL [W3C, 2010c]
Example – Why Rules?

Full name in FOAF from givenName, familyName, assuming Datalog with built-ins:

```
triple(F, foaf:name, N ) :-
    triple(X, rdf:type, foaf:Person),
    triple(X, foaf:givenName, F ),
    triple(X, foaf:familyName S ), N = fn:concat(F, " ", S) .
```

- Not expressible in SPARQL1.0 CONSTRUCT (neither in OWL, btw)

```
CONSTRUCT { ?X foaf:name ?N }
WHERE {?X a foaf:Person; foaf:givenName ?F ; foaf:familyName ?S
    FILTER (?N = fn:concat(?F, " ", ?S)) }
```
Example – RIF Core

Full name in FOAF from givenName, familyName

```prolog
?f[->foaf:name ?N] :-
    ?X[rdf:type->foaf:Person]
    ?X[foaf:givenName->?F],
    ?X[foaf:familyName->?S],
```

- We use a simplified version of RIF’s presentation syntax here.
- RIF has chosen F-Logic style Frames (e.g. FLORA-2) to represent RDF-Triples, cf. [W3C 2010c]
- Can just be viewed as “syntactic sugar” for the triple() predicate we used before
RIF and RDF

1) RDFS entailment rules encodable in RIF Core … obvious.

2) RIF Core Semantics has Datatype reasoning built-in!

RDF Graph:

document1 :language "en"^^xsd:language .

RIF Rule:


The RDF+RIF combined semantics [W3C,2010d] would entail

RIF and SPARQL

Can we Interpret SPARQL CONSTRUCT as a “rules language”?  
[Polleres, 2007], [Schenk&Staab, 2008], [Knublauch et al. 2011]  
Would this rule language be exchangeable in RIF Core?

3 main obstacles:
1) Built-ins: 
   → A RIF dialect including SPARQL built-ins would need specific built-ins.  
      (e.g. bound(), datatype() are not in DTB)  
   → The error semantics of complex FILTERs in SPARQL would need to be emulated in RIF.
2) Negation as failure or something like OPTIONAL would be needed.
3) Datatype Reasoning is built-in into RIF but not in SPARQL.

CONSTRUCT { ?X rdf:type :EngDocument }  

Bottomline: at seems that SPARQL has both more and less than RIF-Core  
→ RIF-SPARQL would need an own RIF-“Dialect”
RIF and Datalog – Summary:

- Positive Datalog is in RIF Core.

- To “cover” RIF Core, you’d need Datalog+Built-ins. Termination problems, could be remedied by syntactic restrictions, e.g. “Strong safeness” [W3C, 2010a, §6,2], inspired by [Eiter+,2006]

- Common extensions to Datalog would need an own RIF Dialect (e.g. not)

- In combination with SPARQL, some obstacles would need to be overcome.
Coming soon!

SPARQL1.1
Why SPARQL 1.1 was needed…

In 2009, a new W3C SPARQL WG was chartered to common feature requests by the community in the query language:

1. Negation
2. Assignment/Project Expressions
3. Property paths
4. Subqueries
5. Aggregate functions (SUM, AVG, MIN, MAX, COUNT, …)
6. Simple query federation
7. Entailment Regimes

- **Goal:** SPARQL 1.1 W3C Recommendation by end of this year
Negation can now be directly expressed in SPARQL1.1:

*Give me people without an email address:*

```sparql
SELECT ?X ?M
WHERE {
  ?X foaf:knows ?Y
  OPTIONAL {
  }
  FILTER(!bound(?M))
}
MINUS {
}
```

We know how to do that... Negation as failure.
Assignment/Project Expressions

Adds the ability to create new values

```
CONSTRUCT { ?X foaf:name ?N }
WHERE { ?X a foaf:Person;
  ?X foaf:givenName ?F ; foaf:familyName ?S
  BIND( fn:concat(?F, " ", ?S) AS ?N ) }
```

*We spoke about this already, in the context of RIF, need built-ins.*
PropertyPaths in SPARQL1.1

```
SELECT ?X
    WHERE {:tim foaf:knows+ ?X }
```

That’s transitive closure, we know how to do this!

```
answer(X) :- Path+(tim,knows,X).

Path+(X,P,Y) :- triple(X, P, Y).
Path+(X,P,Z) :- triple(X, P, Y), Path+(Y,P,Z).
```

**Remark1**: Only linear recursion added!

**Remark2**: No duplicates for *,+ … An earlier WD of the SPARQL1.1 WG had defined a semantics for property paths with duplicates… caused difficulties for implementations and complexity explosion [Arenas et al., 2012], [Losemann&Martens, 2012]
PropertyPaths in SPARQL1.1

```
SELECT ?X
  WHERE {:
    tim foaf:knows* ?X
  }
```

That’s transitive closure, we know how to do this!

```
answer(X) :- Path*(tim,knows,X).
Path*(X,P,X).
Path+(X,P,Y) :- triple(X, P, Y).
Path+(X,P,Z) :- triple(X, P, Y), Path+(Y,P,Z).
```

Remark1: Only linear recursion added!
Remark2: No duplicates for *,+ … An earlier WD of the SPARQL1.1 WG had defined a semantics for property paths with duplicates… caused difficulties for implementations and complexity explosion [Arenas et al., 2012], [Losemann&Martens, 2012]
PropertyPaths in SPARQL1.1 + RDFS

SELECT ?X ?L
WHERE {?X rdf:type foaf:Person. ?X rdfs:label ?L}

Include RDFS inferences by property paths:

SELECT ?X ?L
WHERE {?X rdf:type/rdfs:subClassOf* foaf:Person.

Remark3: Essential RDFS reasoning can be “encoded” in property paths.
cf. also PSPARQL [Alkateeb+, 2009], nSPARQL [Perez+, 2010]
An RDF Graph including RDF lists:

:s  :p  _:b1.
   _:b1 rdf:first 1 .  _:b1 rdf:rest  _:b2 .
   _:b2 rdf:first 1 .  _:b1 rdf:rest  _:b3 .
   _:b3 rdf:first 2 .  _:b1 rdf:rest rdf:nil.

Example Query: Members of the list?

SELECT  ?X
WHERE {  :s :p/rdf:rest*/rdf:first ?X}

Expected result (by majority in the W3C WG):

<table>
<thead>
<tr>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Again! Duplicates (by –implicit – projection)
Subqueries

“Give me a list of scientists (that have been born or died there) for cities in Austria”

```
SELECT ?X
{ ?Y dbpedia:country dbpediares:Austria .
  { SELECT DISTINCT ?Y ?X
    WHERE { { ?X dbpedia:birthPlace ?Y } UNION { ?X dbpedia:deathPlace ?Y }
      ?X rdf:type dbpedia:Scientist. }
  }
}
```

Implications:

1) For one: adds “real” projection
2) Can be combined with other features of SPARQL (DISTINCT, LIMIT, ORDER…)

Note that subqueries in SPARQL 1.1 are very simple [Angles&Gutierrez,2011]
Why SPARQL1.1 was needed…

In 2009, a new W3C SPARQL WG was chartered to common feature requests by the community in the query language:

1. Negation
2. Assignment/Project Expressions
3. Property paths
4. Subqueries
5. Aggregate functions (SUM, AVG, MIN, MAX, COUNT, …) related to aggregates in Datalog, e.g. [Faber+, 2011]?
6. Simple query federation
7. Entailment Regimes (extensions of BGP matching)
   RDFS essentially doable with Entailment Rules, OWL …

… Reading W3C specifications is fun! Enjoy! 😊


References 2/5


References 3/5


[Arenas+, 2012] Marcelo Arenas, Sebastián Conca, Jorge Pérez: Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. WWW 2012: 629-638


http://www.w3.org/TR/rif-overview/

http://www.w3.org/TR/rif-core/
References 4/5

[W3C 2010b] RIF Datatypes and Built-Ins 1.0. Axel Polleres et al. (Eds.) W3C Recommendation 22 June 2010 http://www.w3.org/TR/rif-dtb/


References 5/5


The latest drafts of the SPARQL1.1 working group are available at:
http://www.w3.org/TR/sparql11-overview/ … SPARQL 1.1 Overview
http://www.w3.org/TR/sparql11-query/ … SPARQL 1.1 Query Language
http://www.w3.org/TR/sparql11-entailment/ … SPARQL 1.1 Entailment Regimes
http://www.w3.org/TR/sparql11-federated-query/ … SPARQL 1.1 Federated Query
http://www.w3.org/TR/sparql11-update/ … SPARQL 1.1 Update
http://www.w3.org/TR/sparql11-protocol/ … SPARQL 1.1 Protocol
http://www.w3.org/TR/sparql11-http-rdf-update/ … SPARQL 1.1 Graph Store HTTP Protocol