Next Steps on SPARQL

Axel Polleres (DERI Galway)

Joint work with:
R. Schindlauer (Univ Calabria/TU Vienna), G. Frazzingaro (Univ Calabria), T.Krennwallner (DERI Galway/TU Vienna), F. Scharffe (LFU Innsbruck)

January 09, 2008
Outline

From SPARQL to LP
- Basic Graph Patterns
- GRAPH Patterns
- FILTERs
- UNION Patterns
- OPTIONAL and Negation as failure

Full SPARQL-Spec compliance
- ORDER BY, LIMIT, OFFSET
- Multi-set semantics
- FILTERs in OPTIONALs

SPARQL++ for Ontology alignment
- Mapping by SPARQL
- Examples
- Implementation
- Example Translation
- RDFS

Wrap-up
Starting point: SQL can (to a large extent) be encoded in LP with *negation as failure* (=Datalog$^\text{not}$)

Example: Two tables containing addressbooks
myAddr(Name, Street, City, Telephone)

```
SELECT name FROM myAddr WHERE City = "Cosenza"
    UNION
SELECT name FROM yourAddresses
```

answer1(Name) :- myAddr(Name, Street, "Cosenza", Tel).
answer1(Name) :- yourAddr(Name, Address).

?- answer1(Name).

That was easy... Now what about SPARQL?

OPTIONAL and UNION cause some trouble, also FILTERs and CONSTRUCTs
Starting point: SQL can (to a large extent) be encoded in LP with *negation as failure* (=Datalog\(^{\text{not}}\))

Example: Two tables containing addressbooks

myAddr(Name, Street, City, Telephone)

yourAddr(Name, Address)

SELECT name FROM myAddr WHERE City = "Cosenza"

UNION

SELECT name FROM yourAddresses

answer1(Name) :- myAddr(Name, Street, "Cosenza", Tel).

answer1(Name) :- yourAddr(Name, Address).

?- answer1(Name).

That was easy... Now what about SPARQL?

OPTIONAL and UNION cause some trouble, also FILTERs and CONSTRUCTs
Starting point: SQL can (to a large extent) be encoded in LP with *negation as failure* (=Datalog$^\text{not}$)

Example: Two tables containing address books

\begin{verbatim}
myAddr(Name, Street, City, Telephone)
yourAddr(Name, Address)

SELECT name FROM myAddr WHERE City = "Cosenza"
  UNION
SELECT name FROM yourAddresses

answer1(Name) :- myAddr(Name, Street, "Cosenza", Tel).
answer1(Name) :- yourAddr(Name, Address).
?- answer1(Name).
\end{verbatim}

That was easy... Now what about SPARQL?

- OPTIONAL and UNION cause some trouble, also FILTERs and CONSTRUCTs
Starting point: SQL can (to a large extent) be encoded in LP with negation as failure (=Datalog\textsuperscript{not})

Example: Two tables containing address books

```sql
myAddr(Name, Street, City, Telephone)
yourAddr(Name, Address)
```

```lp
SELECT name FROM myAddr WHERE City = "Cosenza"
    UNION
SELECT name FROM yourAddresses
```

```lp
answer1(Name) :- myAddr(Name, Street, "Cosenza", Tel).
answer1(Name) :- yourAddr(Name, Address).
?- answer1(Name).
```

That was easy... Now what about SPARQL?

OPTIONAL and UNION cause some trouble, also FILTERs and CONSTRUCTs
Starting point: SQL can (to a large extent) be encoded in LP with *negation as failure* (=Datalog\(^\text{not}\))

Example: Two tables containing addressbooks

\[
\text{myAddr}(\text{Name, Street, City, Telephone}) \quad \text{yourAddr}(\text{Name, Address})
\]

SELECT name FROM myAddr WHERE City = "Cosenza"
UNION
SELECT name FROM yourAddresses

\[
\text{answer1}(\text{Name}) :- \text{myAddr}(\text{Name, Street, "Cosenza", Tel}).
\]
\[
\text{answer1}(\text{Name}) :- \text{yourAddr}(\text{Name, Address}).
\]

?- \text{answer1}(\text{Name}).

That was easy... Now what about SPARQL?

OPTIONAL and UNION cause some trouble, also FILTERs and CONSTRUCTs
We take as an example the language of dlvhex
(http://www.kr.tuwien.ac.at/research/dlvhex):

- Prolog-like syntax
- We assume availability of built-in predicate `rdf[URL](S,P,O)` to import RDF data.
- dlvhex is implemented on top of the DLV engine (http://www.dlvsystem.com/)
- supports so-called answer set semantics (extension of the stable model semantics) for a language extending Datalog [Eiter et al., 2006].
- plugin-mechanism for easy integration of external function calls (built-in predicates, also called HEX-atoms).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data, more HEX-atoms later.
We take as an example the language of dlvhex
(http://www.kr.tuwien.ac.at/research/dlvhex):

- Prolog-like syntax
- We assume availability of built-in predicate
  `rdf[URL](S,P,O)` to import RDF data.
- dlvhex is implemented on top of the DLV engine
  (http://www.dlvsystem.com/)
- supports so-called answer set semantics (extension of the stable model
  semantics) for a language extending Datalog [Eiter et al., 2006].
- plugin-mechanism for easy integration of external function calls (built-in
  predicates, also called HEX-atoms).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data, more HEX-atoms
  later.
We take as an example the language of dlvhex
(http://www.kr.tuwien.ac.at/research/dlvhex):

- Prolog-like syntax
- We assume availability of built-in predicate `rdf[URL](S,P,O)` to import RDF data.
- dlvhex is implemented on top of the DLV engine (http://www.dlvsystem.com/)
- supports so-called answer set semantics (extension of the stable model semantics) for a language extending Datalog [Eiter et al., 2006].
- plugin-mechanism for easy integration of external function calls (built-in predicates, also called HEX-atoms).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data, more HEX-atoms later.
We take as an example the language of dlvhex
(http://www.kr.tuwien.ac.at/research/dlvhex):

- **Prolog-like syntax**

- **We assume availability of built-in predicate**
  \[\text{rdf[URL]}(S,P,O)\] **to import RDF data.**

- **dlvhex is implemented on top of the DLV engine**
  (http://www.dlvsystem.com/)

- **supports so-called answer set semantics (extension of the stable model semantics) for a language extending Datalog [Eiter et al., 2006].**

- plugin-mechanism for easy integration of external function calls (built-in predicates, also called HEX-atoms).

- **\[\text{rdf[URL]}(S,P,O)\] is one such built-in to import RDF data, more HEX-atoms later.**
We take as an example the language of dlvhex (http://www.kr.tuwien.ac.at/research/dlvhex):

- Prolog-like syntax
- We assume availability of built-in predicate `rdf[URL](S,P,O)` to import RDF data.
- `dlvhex` is implemented on top of the DLV engine (http://www.dlvsystem.com/)
- Supports so-called answer set semantics (extension of the stable model semantics) for a language extending Datalog [Eiter et al., 2006].
- Plugin-mechanism for easy integration of external function calls (built-in predicates, also called HEX-atoms).
- `rdf[URL](S,P,O)` is one such built-in to import RDF data, more HEX-atoms later.
SPARQL and LP: Basic Graph Patterns

- We import all triples in a predicate `triple(Subj,Pred,Object,Graph)` which carries an additional argument for the dataset.
- For the import, we use the `rdf[URL](S,P,O)` built-in.

"select persons and their names"

```
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }
```

```
answer1(X,Y,def) :- triple(X,"rdf:type","foaf:Person",def),
                   triple(X,"foaf:name",Y,def).
?- answer1(X,Y,def).
```
We import all triples in a predicate `triple(Subj,Pred,Object,Graph)` which carries an additional argument for the dataset.

For the import, we use the `rdf[URL](S,P,O)` built-in.

> **“select persons and their names”**

```sparql
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }
```

```prolog
answer1(X,Y,def) :- triple(X,"rdf:type","foaf:Person",def),
                  triple(X,"foaf:name",Y,def).
?- answer1(X,Y,def).
```

A. Polleres

2008-01-09
SPARQL and LP: Basic Graph Patterns

- We import all triples in a predicate `triple(Subj,Pred,Object,Graph)` which carries an additional argument for the dataset.
- For the import, we use the `rdf[URL](S,P,O)` built-in.

"select persons and their names"

```
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }
```

```
answer1(X,Y,def) :- triple(X,"rdf:type","foaf:Person",def),
                   triple(X,"foaf:name",Y,def).

?- answer1(X,Y,def).
```
We import all triples in a predicate `triple(Subj, Pred, Object, Graph)` which carries an additional argument for the dataset.

For the import, we use the `rdf[URL](S, P, O)` built-in.

"select persons and their names"

```sparql
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }
```

```prolog
answer1(X,Y,def) :- triple(X,"rdf:type","foaf:Person",def),
                 triple(X,"foaf:name",Y,def).
?- answer1(X,Y,def).
```
We import all triples in a predicate triple(Subj,Pred,Object,Graph) which carries an additional argument for the dataset.

For the import, we use the rdf[URL](S,P,O) built-in.

"select persons and their names"

```sql
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }
```

```prolog
answer1(X,Y,def) :- triple(X,"rdf:type","foaf:Person",def),
                    triple(X,"foaf:name",Y,def).
?- answer1(X,Y,def).
```
"select creators of graphs and the persons they know"

SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
          GRAPH ?G { ?X foaf:knows ?Y. } }

triple(S,P,O,def) :- rdf["alice.org"](S,P,O).
triple(S,P,O,"alice.org") :- rdf["alice.org"](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf["ex.org/bob"](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),

For legibility we left out the http:// prefix
“select creators of graphs and the persons they know”

```
SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
    GRAPH ?G { ?X foaf:knows ?Y. } }
```

```
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).
triple(S,P,O,"alice.org") :- rdf["alice.org"](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf["ex.org/bob"](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),
```

For legibility we left out the http:// prefix
SPARQL and LP: GRAPH Patterns and NAMED graphs

“select creators of graphs and the persons they know”

SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
    GRAPH ?G { ?X foaf:knows ?Y. } }

triple(S,P,O,def) :- rdf["alice.org"](S,P,O).
triple(S,P,O,"alice.org") :- rdf["alice.org"](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf["ex.org/bob"](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),

For legibility we left out the http:// prefix
"select creators of graphs and the persons they know"

```sparql
SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
  GRAPH ?G { ?X foaf:knows ?Y. } } }
```

```prolog
triple(S,P,O,def) :- rdf[alice.org](S,P,O).
triple(S,P,O,"alice.org") :- rdf[alice.org](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf[ex.org/bob](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),
```

For legibility we left out the `http://` prefix
"select creators of graphs and the persons they know"

SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
             GRAPH ?G { ?X foaf:knows ?Y. } }

triple(S,P,O,def) :- rdf["alice.org"](S,P,O).
triple(S,P,O,"alice.org") :- rdf["alice.org"](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf["ex.org/bob"](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),

For legibility we left out the http:// prefix
**SPARQL and LP: GRAPH Patterns and NAMED graphs**

"select creators of graphs and the persons they know"

```
SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
    GRAPH ?G { ?X foaf:knows ?Y. } }
```

```
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).
triple(S,P,O,"alice.org") :- rdf["alice.org"](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf["ex.org/bob"](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),
```

For legibility we left out the http:// prefix
FILTERs are used to filter the result set of a query. FILTER expressions can be encoded by built-in predicates:

```
SELECT ?X
FROM ... 
  FILTER( ?Age > 30 )
}
```

```
answer1(X,def) :-
  triple(X,foaf:mbox,M,def), triple(X,:age,Age,def),
  Age > 30.
```

Unbound variables in FILTERs need to be replaced by constant, to avoid unsafe rules.
**SPARQL and LP: FILTERs**

FILTERs are used to filter the result set of a query. FILTER expressions can be encoded by built-in predicates:

```
SELECT ?X
FROM ...
FILTER( ?Age > 30 )
}
```

```
answer1(X,def) :-
    triple(X,foaf:mbox,M,def), triple(X,:age,Age,def),
    Age > 30.
```

Unbound variables in FILTERs need to be replaced by constant, to avoid unsafe rules.
**SPARQL and LP: FILTERs**

FILTERs are used to filter the result set of a query. FILTER expressions can be encoded by built-in predicates:

```
SELECT ?X
FROM ...
    FILTER( ?Age > 30 ) }
```

```
answer1(X,def) :-
    triple(X,foaf:mbox,M,def),
    null > 30.
```

unbound variables in FILTERs need to be replaced by constant, to avoid unsafe rules.
UNIONs are split off into several rules:

"select Persons and their names or nicknames"

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

triple(S, P, O, def) :- ...
answer1(X, Y, def) :- triple(X, "foaf:name", Y, def).
answer1(X, Y, def) :- triple(X, "foaf:nick", Y, def).
UNIONs are split off into several rules:

"select Persons and their names or nicknames"

```
SELECT ?X ?Y
FROM ... WHERE { ?X foaf:name ?Y . } UNION { ?X foaf:nick ?Y . }
```

```
triple(S,P,O,def) :- ...
answer1(X,Y,def) :- triple(X,"foaf:name",Y,def).
answer1(X,Y,def) :- triple(X,"foaf:nick",Y,def).
```
UNIONs are split off into several rules:

“select Persons and their names or nicknames”

```prolog
SELECT ?X ?Y
FROM ...
WHERE { { ?X foaf:name ?Y . }
UNION { ?X foaf:nick ?Y . } }

triple(S,P,O,def) :- ...
answer1(X,Y,def) :- triple(X,"foaf:name",Y,def).
answer1(X,Y,def) :- triple(X,"foaf:nick",Y,def).
```
What if variables of the constituent patterns don’t coincide? Slightly different than in SQL!

We emulate this by special null values!

```
SELECT ?X ?Y ?Z
FROM ...
WHERE { { ?X foaf:name ?Y . } 
  UNION { ?X foaf:nick ?Z .} }
```

Data:

```
<alice.org#me> foaf:name "Alice".
<ex.org/bob#me> foaf:name "Bob"; foaf:nick "Bobby".
```

Result:

<table>
<thead>
<tr>
<th>?X</th>
<th>?Y</th>
<th>?Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;alice.org#me&gt;</td>
<td>&quot;Alice&quot;</td>
<td></td>
</tr>
<tr>
<td>&lt;ex.org/bob#me&gt;</td>
<td>&quot;Bob&quot;</td>
<td>&quot;Bobby&quot;</td>
</tr>
<tr>
<td>&lt;ex.org/bob#me&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What if variables of the constituent patterns don’t coincide? Slightly different than in SQL!

We emulate this by special null values!

```
SELECT ?X ?Y ?Z
FROM ...
WHERE { { ?X foaf:name ?Y . } UNION { ?X foaf:nick ?Z . } }
```

Data:
```
<alice.org#me> foaf:name "Alice".
<ex.org/bob#me> foaf:name "Bob"; foaf:nick "Bobby".
```

Result:
```
<table>
<thead>
<tr>
<th>?X</th>
<th>?Y</th>
<th>?Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;alice.org#me&gt;</td>
<td>&quot;Alice&quot;</td>
<td></td>
</tr>
<tr>
<td>&lt;ex.org/bob#me&gt;</td>
<td>&quot;Bob&quot;</td>
<td>&quot;Bobby&quot;</td>
</tr>
</tbody>
</table>
```
What if variables of the constituent patterns don’t coincide? Slightly different than in SQL!

We emulate this by special null values!

```
SELECT ?X ?Y ?Z
FROM ...
WHERE { { ?X foaf:name ?Y . } 
        UNION { ?X foaf:nick ?Z . } }
```

Data:
```
<alice.org#me> foaf:name "Alice".
<ex.org/bob#me> foaf:name "Bob"; foaf:nick "Bobby".
```

Result:
```
<table>
<thead>
<tr>
<th>?X</th>
<th>?Y</th>
<th>?Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;alice.org#me&gt;</td>
<td>&quot;Alice&quot;</td>
<td></td>
</tr>
<tr>
<td>&lt;ex.org/bob#me&gt;</td>
<td>&quot;Bob&quot;</td>
<td></td>
</tr>
<tr>
<td>&lt;ex.org/bob#me&gt;</td>
<td>&quot;Bobby&quot;</td>
<td></td>
</tr>
</tbody>
</table>
What if variables of the constituent patterns don’t coincide? Slightly different than in SQL!

We emulate this by special null values!

```
SELECT ?X ?Y ?Z
FROM ...
WHERE { { ?X foaf:name ?Y . } 
    UNION { ?X foaf:nick ?Z .} }
```

Data:
```
<alice.org#me> foaf:name "Alice".
<ex.org/bob#me> foaf:name "Bob"; foaf:nick "Bobby".
```

Result:
```
<table>
<thead>
<tr>
<th>?X</th>
<th>?Y</th>
<th>?Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;alice.org#me&gt;</td>
<td>&quot;Alice&quot;</td>
<td>null</td>
</tr>
<tr>
<td>&lt;ex.org/bob#me&gt;</td>
<td>&quot;Bob&quot;</td>
<td>null</td>
</tr>
<tr>
<td>&lt;ex.org/bob#me&gt;</td>
<td>null</td>
<td>&quot;Bobby&quot;</td>
</tr>
</tbody>
</table>
```
SPARQL and LP: UNION Patterns 2/2

What if variables of the constituent patterns don’t coincide? Slightly different than in SQL!

We emulate this by special null values!

```
SELECT ?X ?Y ?Z
FROM ...
WHERE { { ?X foaf:name ?Y . } 
  UNION { ?X foaf:nick ?Z . } }
```

```
triple(S,P,O,def) :- ...
answer1(X,Y,null,def) :- triple(X,"foaf:name",Y,def).
```
What if variables of the constituent patterns don’t coincide? Slightly different than in SQL!

We emulate this by special null values!

```
SELECT ?X ?Y ?Z
FROM ...
WHERE { { ?X foaf:name ?Y . } 
          UNION { ?X foaf:nick ?Z . } }
```

```
triple(S,P,O,def) :- ...
answer1(X,Y,null,def) :- triple(X,"foaf:name",Y,def).
```
"select all persons and optionally their names"

```sql
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}
```

OPTIONAL is similar to an OUTER JOIN in SQL, actually it is a combination of a join and set difference (see [Pérez et al., 2006]):

\[
\{ P_1 \text{ OPTIONAL } \{ P_2 \} \} : \quad M_1 \bowtie M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)
\]

where \( M_1 \) and \( M_2 \) are variable bindings for \( P_1 \) and \( P_2 \), resp.
“select all persons and optionally their names”

```
SELECT * 
WHERE 
{ 
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N } 
}
```

OPTIONAL is similar to an OUTER JOIN in SQL, actually it is a combination of a join and set difference (see [Pérez et al., 2006]):

\[
\{ P_1 \text{ OPTIONAL } \{ P_2 \} \}: \quad M_1 \bowtie M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)
\]

where \( M_1 \) and \( M_2 \) are variable bindings for \( P_1 \) and \( P_2 \), resp.
**SPARQL and LP: ** *OPTIONAL* Patterns 1/2

"select all persons and optionally their names"

```sql
SELECT *
WHERE 
{
    ?X a foaf:Person .
    OPTIONAL {?X foaf:name ?N }
}
```

*OPTIONAL* is similar to an OUTER JOIN in SQL, actually it is a combination of a *join* and *set difference* (see [Pérez et al., 2006]):

\[
\{ P_1 \text{ OPTIONAL } \{ P_2 \} \}: \quad M_1 \Join M_2 = (M_1 \Join M_2) \cup (M_1 \setminus M_2)
\]

where \( M_1 \) and \( M_2 \) are variable bindings for \( P_1 \) and \( P_2 \), resp.
Observation: SPARQL allows to express set difference / negation as failure by combining OPT and FILTER !bound

“select all persons without an email address”

```sparql
SELECT ?X
WHERE
{
    ?X a ?Person
    OPTIONAL { ?X :email ?Email }
    FILTER ( !bound( ?Email ) )
}
```

Same effect as “NOT EXISTS” in SQL, set difference!.

We’ve seen before that OPTIONAL, has set difference inherent, with the “!bound” we get it back again “purely”.
SPARQL’s OPTIONAL has “negation as failure”, hidden:

- Observation: SPARQL allows to express set difference / negation as failure by combining OPT and FILTER !bound

“select all persons without an email address”

```sparql
SELECT ?X
WHERE
{
 ?X a ?Person
 OPTIONAL {?X :email ?Email }
 FILTER ( !bound( ?Email ) )
}
```

- Same effect as “NOT EXISTS” in SQL, set difference!.

- We’ve seen before that OPTIONAL, has set difference inherent, with the “!bound” we get it back again “purely”. 
Observation: SPARQL allows to express set difference / negation as failure by combining OPT and FILTER !bound

“select all persons *without* an email address”

```sparql
SELECT ?X
WHERE {
  ?X a ?Person
  OPTIONAL {?X :email ?Email }
  FILTER ( !bound( ?Email ) )
}
```

Same effect as “NOT EXISTS” in SQL, set difference!.

We’ve seen before that OPTIONAL, has set difference inherent, with the “!bound” we get it back again “purely”.
### SPARQL and LP: OPT Patterns – First Try

SELECT *  
WHERE 
{  
  ?X a foaf:Person .  
  OPTIONAL {?X foaf:name ?N }  
}

Recall: \((P_1 \text{ OPT } P_2)\):  
\[ M_1 \bowtie M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2) \]

\begin{verbatim}
triple(S,P,O,def) :- ...  
answer1(X,N,def) :- triple(X,"rdf:type","foaf:Person",def),  
                  triple(X,"foaf:name",N,def).  
answer1(X,null,def) :- triple(X,"rdf:type","foaf:Person",def),  
                    not answer2(X).  
answer2(X) :- triple(X,"foaf:name",N,def).
\end{verbatim}

We use `null` and negation as failure `not` to "emulate" set difference.
SPARQL and LP: OPT Patterns – First Try

```
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}
```

Recall: \((P_1 \text{ OPT } P_2)\): \(M_1 \bowtie M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)\)

```
triple(S,P,O,def) :- ... 
answer1(X,N,def) :- triple(X,"rdf:type","foaf:Person",def),
  triple(X,"foaf:name",N,def).
answer1(X,null,def) :- triple(X,"rdf:type","foaf:Person",def),
  not answer2(X).
answer2(X) :- triple(X,"foaf:name",N,def).
```

We use null and negation as failure not to “emulate” set difference.
SELECT *  
WHERE  
{
    ?X a foaf:Person .  
    OPTIONAL {?X foaf:name ?N }
}

Recall: \((P_1 \text{ OPT } P_2)\): \[ M_1 \Join M_2 = (M_1 \Join M_2) \cup (M_1 \setminus M_2) \]

\[
\begin{align*}
\text{triple}(S,P,O,\text{def}) & :\quad \ldots \\
\text{answer1}(X,N,\text{def}) & :\quad \text{triple}(X,"\text{rdf:type}","\text{foaf:Person}",\text{def}), \\
& \quad \text{triple}(X,"\text{foaf:name}",N,\text{def}). \\
\text{answer1}(X,\text{null},\text{def}) & :\quad \text{triple}(X,"\text{rdf:type}","\text{foaf:Person}",\text{def}), \\
& \quad \text{not} \ \text{answer2}(X). \\
\text{answer2}(X) & :\quad \text{triple}(X,"\text{foaf:name}",N,\text{def}).
\end{align*}
\]

We use \textit{null} and negation as failure \textit{not} to “emulate” set difference.
SPARQL and LP: OPT Patterns – First Try

```sparql
SELECT *
WHERE {
    ?X a foaf:Person .
    OPTIONAL {?X foaf:name ?N }
}
```

Recall: \((P_1 \text{ OPT } P_2)\): \(M_1 \bowtie M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)\)

```prolog
triple(S,P,O,def) :- ... 
answer1(X,N,def) :- triple(X,"rdf:type","foaf:Person",def),
    triple(X,"foaf:name",N,def).
answer1(X,null,def) :- triple(X,"rdf:type","foaf:Person",def),
    not answer2(X).
answer2(X) :- triple(X,"foaf:name",N,def).
```

We use `null` and negation as failure `not` to “emulate” set difference.
SPARQL and LP: OPT Patterns – First Try

```
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}

Recall: \((P_1 \text{ OPT } P_2)\) : \(\mathcal{M}_1 \cong \mathcal{M}_2 = (\mathcal{M}_1 \cong \mathcal{M}_2) \cup (\mathcal{M}_1 \setminus \mathcal{M}_2)\)

\begin{align*}
\text{triple}(S,P,O,\text{def}) & : - \ldots \\
\text{answer1}(X,N,\text{def}) & : - \text{triple}(X,\text{"rdf:type"},\text{"foaf:Person"},\text{def}), \text{triple}(X,\text{"foaf:name"},N,\text{def}). \\
\text{answer1}(X,\text{null},\text{def}) & : - \text{triple}(X,\text{"rdf:type"},\text{"foaf:Person"},\text{def}), \text{not answer2}(X). \\
\text{answer2}(X) & : - \text{triple}(X,\text{"foaf:name"},N,\text{def}).
\end{align*}
```

We use \texttt{null} and negation as failure \texttt{not} to “emulate” set difference.
SPARQL and LP: OPT Patterns – First Try

```sparql
SELECT *
WHERE {
    ?X a foaf:Person .
    OPTIONAL {?X foaf:name ?N }
}
```

Recall: \((P_1 \text{ OPT } P_2)\): \(M_1 \bowtie M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)\)

```prolog
triple(S,P,O,def) :- ...
answer1(X,N,def) :- triple(X,"rdf:type","foaf:Person",def),
                   triple(X,"foaf:name",N,def).
answer1(X,null,def) :- triple(X,"rdf:type","foaf:Person",def),
                      not answer2(X).
answer2(X) :- triple(X,"foaf:name",N,def).
```

We use \textbf{null} and negation as failure \textbf{not} to “emulate” set difference.

A. Polleres
SPARQL and LP: OPT Patterns – Example

# Graph: ex.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix bob: <ex.org/bob#> .

  <ex.org/bob> foaf:maker __a.
  __:a a foaf:Person ; foaf:name "Bob";
    foaf:knows __:b.

  __:b a foaf:Person ; foaf:nick "Alice".
  <alice.org/> foaf:maker __:b

# Graph: alice.org
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix alice: <alice.org#> .

  alice:me a foaf:Person ; foaf:name "Alice" ;
    foaf:knows __:c.

  __:c a foaf:Person ; foaf:name "Bob" ;
    foaf:nick "Bobby".

SELECT *
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . OPTIONAL { ?X foaf:name ?N } }
SPARQL and LP: OPT Patterns – Example

```sparql
# Graph: ex.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix bob: <ex.org/bob#> .

<ex.org/bob> foaf:maker __:a.
__:a a foaf:Person ; foaf:name "Bob";
    foaf:knows __:b.

__:b a foaf:Person ; foaf:nick "Alice".

# Graph: alice.org

@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix alice: <alice.org#> .

alice:me a foaf:Person ; foaf:name "Alice" ;
    foaf:knows __:c.

__:c a foaf:Person ; foaf:name "Bob" ;
    foaf:nick "Bobby".

SELECT *
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . OPTIONAL { ?X foaf:name ?N } }

Result:

<table>
<thead>
<tr>
<th>?X</th>
<th>?N</th>
</tr>
</thead>
<tbody>
<tr>
<td>__:a</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>__:b</td>
<td></td>
</tr>
<tr>
<td>__:c</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>alice.org#me</td>
<td>&quot;Alice&quot;</td>
</tr>
</tbody>
</table>

{ answer1("__:a","Bob",def),
  answer1("__:b",null, def),
  answer1("__:c","Bob",def),
  answer1("alice.org#me","Alice", def)}
```

A. Polleres
# SPARQL and LP: OPT Patterns – Example

```sparql
# Graph: ex.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix bob: <ex.org/bob#> .

<ex.org/bob> foaf:maker __:a.
__:a a foaf:Person ; foaf:name "Bob";
foaf:knows __:b.
__:b a foaf:Person ; foaf:nick "Alice".
<alice.org/> foaf:maker __:b

# Graph: alice.org
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix alice: <alice.org#> .

alice:me a foaf:Person ; foaf:name "Alice";
foaf:knows __:c.
__:c a foaf:Person ; foaf:name "Bob";
foaf:nick "Bobby".
```

SELECT *
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . OPTIONAL { ?X foaf:name ?N } }

Result:

<table>
<thead>
<tr>
<th>?X</th>
<th>?N</th>
</tr>
</thead>
<tbody>
<tr>
<td>__:a</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>__:b</td>
<td>null</td>
</tr>
<tr>
<td>__:c</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>alice.org#me</td>
<td>&quot;Alice&quot;</td>
</tr>
</tbody>
</table>

{ answer1("__:a","Bob",def), answer1("__:b",null, def),
  answer1("__:c","Bob",def), answer1("alice.org#me","Alice", def) }
Ask for pairs of persons \(?X1, ?X2\) who share the same name and nickname where both, name and nickname are optional:

```
SELECT *
FROM ... 
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } } 
   { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>__:a</td>
<td>&quot;Bob&quot;</td>
<td>__:a</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>__:b</td>
<td>&quot;Bob&quot;</td>
<td>__:b</td>
<td>&quot;Bobby&quot;</td>
</tr>
<tr>
<td>__:c</td>
<td>Alice.org#me</td>
<td>__:c</td>
<td>Alice.org#me</td>
</tr>
</tbody>
</table>

Now this is strange, as we join over unbound variables.

**Remark:** this pattern is not well-designed, following [Pérez et al., 2006]
Ask for pairs of persons \(?X1\), \(?X2\) who share the same name and nickname where both, name and nickname are optional:

```
SELECT *
FROM ...
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } } 
  { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }
```

<table>
<thead>
<tr>
<th>(?X1)</th>
<th>(?N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>__:a</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>__:b</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>__:c</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>alice.org#me</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(?X2)</th>
<th>(?N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>__:a</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>__:b</td>
<td>&quot;Bobby&quot;</td>
</tr>
<tr>
<td>__:c</td>
<td></td>
</tr>
<tr>
<td>alice.org#me</td>
<td></td>
</tr>
</tbody>
</table>

Now this is strange, as we join over unbound variables.

**Remark:** this pattern is not well-designed, following [Pérez et al., 2006]
Ask for pairs of persons $?X1, ?X2 who share the same name and nickname where both, name and nickname are optional:

```
SELECT *
FROM ... 
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } } 
{ ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }
```

Now this is strange, as we join over unbound variables.

**Remark:** this pattern is not well-designed, following [Pérez et al., 2006]
SPARQL and LP: OPT Patterns – With our translation?:

What’s wrong here? Join over `null`, as if it was a normal constant. Compared with SPARQL’s notion of compatibility of mappings, this is too cautious!
**SPARQL and LP: OPT Patterns – With our translation?:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>__:a</td>
<td>&quot;Bob&quot;</td>
<td>__:a</td>
<td>null</td>
</tr>
<tr>
<td>__:b</td>
<td>null</td>
<td>__:b</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>__:c</td>
<td>&quot;Bob&quot;</td>
<td>__:c</td>
<td>&quot;Bobby&quot;</td>
</tr>
<tr>
<td>alice.org#me</td>
<td></td>
<td>alice.org#me</td>
<td>null</td>
</tr>
</tbody>
</table>

What’s wrong here? Join over `null`, as if it was a normal constant.

Compared with SPARQL’s notion of compatibility of mappings, this is too **cautious!**
From SPARQL to LP Spec compliance Ontology alignment Wrap-up

Basic Graph Patterns GRAPH Patterns FILTERs UNION Patterns OPTIONAL and Negation as failure

SPARQL and LP: OPT Patterns – Correct Result:

<table>
<thead>
<tr>
<th>?X1</th>
<th>?N</th>
</tr>
</thead>
<tbody>
<tr>
<td>__:a</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>__:b</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>__:c</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>alice.org#me</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>?X2</th>
<th>?N</th>
</tr>
</thead>
<tbody>
<tr>
<td>__:a</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>__:b</td>
<td>&quot;Bobby&quot;</td>
</tr>
<tr>
<td>__:c</td>
<td>alice.org#me</td>
</tr>
</tbody>
</table>

SPARQL defines a very brave way of joins: unbound, i.e. null should join with anything!
SPARQL and LP: OPT Patterns – Correct Result:

<table>
<thead>
<tr>
<th>?X1</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>__:a</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>__:b</td>
<td></td>
</tr>
<tr>
<td>__:c</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>alice.org#me</td>
<td>&quot;Alice&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>?X2</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>__:a</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>__:b</td>
<td>&quot;Bobby&quot;</td>
</tr>
<tr>
<td>__:c</td>
<td>alice.org#me</td>
</tr>
</tbody>
</table>

SPARQL defines a very brave way of joins: unbound, i.e. null should join with anything!

A. Polleres
Semantic variations of SPARQL

We could call these alternatives of treatment of possibly null-joining values alternative semantics for SPARQL:

- **c-joining**: cautiously joining semantics
- **b-joining**: bravely joining semantics (normative)

Which is the most intuitive? DAWG basically decided for b-join.

Now let's see to how to fix our translation to logic programs...
Semantic variations of SPARQL

We could call these alternatives of treatment of possibly null-joining values alternative semantics for SPARQL:

- **c-joining**: cautiously joining semantics
- **b-joining**: bravely joining semantics (normative)

Which is the most intuitive? DAWG basically decided for b-join.

Now let's see how to fix our translation to logic programs...
Semantic variations of SPARQL

We could call these alternatives of treatment of possibly null-joining values alternative semantics for SPARQL:

▶ c-joining: cautiously joining semantics
▶ b-joining: bravely joining semantics (normative)

Which is the most intuitive? DAWG basically decided for b-join.

Now let’s see to how to fix our translation to logic programs...
SELECT *
FROM ...
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } } 
    { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }

triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).

answer2(N, X1,def) :- triple(X1,"a","Person",def),
    triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def),
    not answer3(X1,def).
answer3(X1,def) :- triple(X1,"name",N,def).

answer4(N, X2,def) :- triple(X2,"a","Person",def),
    triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def),
    not answer5(X2,def).
answer5(X2,def) :- triple(X2,"nick",N,def).

Here is the problem! Join over a possibly null-joining variable
SELECT *
FROM ...

triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).

answer2(N, X1,def) :- triple(X1,"a","Person",def),
                 triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def),
                     not answer3(X1,def).
answer3(X1,def) :- triple(X1,"name",N,def).

answer4(N, X2,def) :- triple(X2,"a","Person",def),
                 triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def),
                     not answer5(X2,def).
answer5(X2,def) :- triple(X2,"nick",N,def).

Here is the problem! Join over a possibly null-joining variable.
SELECT *
FROM ...
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } }
{ ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }

triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).

answer2(N, X1,def) :- triple(X1,"a","Person",def),
                triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def),
                not answer3(X1,def).
answer3(X1,def) :- triple(X1,"name",N,def).

answer4(N, X2,def) :- triple(X2,"a","Person",def),
               triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def),
               not answer5(X2,def).
answer5(X2,def) :- triple(X2,"nick",N,def).

Here is the problem! Join over a possibly null-joining variable.

A. Polleres
SELECT *
FROM ... 
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } } 
{ ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } } 

triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).

answer2(N, X1,def) :- triple(X1,"a","Person",def), 
triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def), 
not answer3(X1,def).
answer3(X1,def) :- triple(X1,"name",N,def).

answer4(N, X2,def) :- triple(X2,"a","Person",def), 
triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def), 
not answer5(X2,def).
answer5(X2,def) :- triple(X2,"nick",N,def).

Here is the problem! Join over a possibly null-joining variable
SELECT *  
FROM ...  
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } }  
           { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } } 

triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O). 
triple(S,P,O,def) :- rdf["alice.org"](S,P,O). 

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def). 
answer2(N, X1,def) :- triple(X1,"a","Person",def),   
                         triple(X1,"name",N,def). 
answer2(null,X1,def) :- triple(X1,"a","Person",def),   
                         not answer3(X1,def). 
answer3(X1,def)       :- triple(X1,"name",N,def). 

answer4(N, X2,def) :- triple(X2,"a","Person",def),   
                         triple(X2,"nick",N,def). 
answer4(null,X2,def) :- triple(X2,"a","Person",def),   
                         not answer5(X2,def). 
answer5(X2,def)       :- triple(X2,"nick",N,def). 

Here is the problem! Join over a possibly null-joining variable
How do I emulate b-joining Semantics? **Solution:**

We need to take care for variables which are joined and possibly unbound, due to the special notion of compatibility in SPARQL.

```prolog
triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).
answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(null,X2,def).
answer1(N,X1,X2,def) :- answer2(null,X1,def), answer4(N,X2,def).

answer2(N, X1,def) :- triple(X1,"a","Person",def), triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def), not answer3(X1,def).
answer3(X1,def) :- triple(X1,"name",N,def).

answer4(N, X2,def) :- triple(X2,"a","Person",def), triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def), not answer5(X2,def).
answer5(X2,def) :- triple(X2,"nick",N,def).
```
How do I emulate b-joining Semantics? **Solution:**

We need to take care for variables which are joined and possibly unbound, due to the special notion of compatibility in SPARQL

```prolog
triple(S,P,O,def) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(N,X2,def).
answer1(N,X1,X2,def) :- answer2(N,X1,def), answer4(null,X2,def).
answer1(N,X1,X2,def) :- answer2(null,X1,def), answer4(N,X2,def).

answer2(N, X1,def) :- triple(X1,"a","Person",def),
                            triple(X1,"name",N,def).
answer2(null,X1,def) :- triple(X1,"a","Person",def),
                            not answer3(X1,def).
answer3(X1,def) :- triple(X1,"name",N,def).

answer4(N, X2,def) :- triple(X2,"a","Person",def),
                            triple(X2,"nick",N,def).
answer4(null,X2,def) :- triple(X2,"a","Person",def),
                            not answer5(X2,def).
answer5(X2,def) :- triple(X2,"nick",N,def).
```
Attention:

- The “fix” we used to emulate b-joining semantics is potentially exponential in the number of possibly-null-joining variables.

- This is not surprising, since the complexity of OPTIONAL/UNION corner cases is PSPACE, see [Pérez et al., 2006].

- But: A slight modification of the translation (in the tech. report version of [Polleres, 2007]) shows that this translation is optimal: Non-recursive Datalog with negation as failure is also PSPACE complete!
Attention:

- The “fix” we used to emulate b-joining semantics is potentially exponential in the number of possibly-null-joining variables.

- This is not surprising, since the complexity of OPTIONAL/UNION corner cases is PSPACE, see [Pérez et al., 2006].

- But: A slight modification of the translation (in the tech. report version of [Polleres, 2007]) shows that this translation is optimal: Non-recursive Datalog with negation as failure is also PSPACE complete!
Attention:

- The “fix” we used to emulate b-joining semantics is potentially exponential in the number of possibly-null-joining variables.
- This is not surprising, since the complexity of OPTIONAL/UNION corner cases is PSPACE, see [Pérez et al., 2006].
- But: A slight modification of the translation (in the tech. report version of [Polleres, 2007]) shows that this translation is optimal: Non-recursive Datalog with negation as failure is also PSPACE complete!
Outline

From SPARQL to LP
- Basic Graph Patterns
- GRAPH Patterns
- FILTERs
- UNION Patterns
- OPTIONAL and Negation as failure

Full SPARQL-Spec compliance
- ORDER BY, LIMIT, OFFSET
- Multi-set semantics
- FILTERs in OPTIONALs

SPARQL++ for Ontology alignment
- Mapping by SPARQL
- Examples
- Implementation
- Example Translation
- RDFS

Wrap-up

A. Polleres
That’s all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite …

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk).
That’s all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite . . .

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk).
That’s all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite …

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)
SPARQL Specification compliance

That’s all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite …

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)
SPARQL Specification compliance

That’s all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite . . .

▶ What we presented so far was reflecting [Pérez et al., 2006] semantics.

▶ The SPARQL spec defines an algebra which adds some peculiarities, namely:

1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
2. SPARQL defines a multi-set semantics.
3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)
SPARQL Specification compliance

That’s all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite . . .

▶ What we presented so far was reflecting [Pérez et al., 2006] semantics.

▶ The SPARQL spec defines an algebra which adds some peculiarities, namely:

1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
2. SPARQL defines a multi-set semantics.
3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)
That’s all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite …

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:
  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)
That’s all? So, can we use a bottom-up datalog engine like delvhex as a SPARQL engine? Not quite . . .

- What we presented so far was reflecting [Pérez et al., 2006] semantics.
- The SPARQL spec defines an algebra which adds some peculiarities, namely:

  1. How to deal with solution modifiers (ORDER BY, LIMIT, OFFSET).
  2. SPARQL defines a multi-set semantics.
  3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
  4. SPARQL allows blank nodes in the result form of CONSTRUCT queries (more on that in the 3rd part of the talk)
Not treated at the moment in our implementation, in principle doable by postprocessing of the results:

Data:
<ex.org/bob#me> foaf:name "Bob" .
<alice.org#me> foaf:name "Alice".
<ex.org/bob#me> foaf:nick "Bobby".

SELECT ?Y
WHERE { ?X foaf:name ?Y }
ORDER BY ?Y LIMIT 1

Result: { answer1("Bob",def), answer1("Alice",def) }
Sort answer set by parameter corresponding to ?Y (ORDER BY),
only output first result (LIMIT 1) ⇒ "Alice"
Not treated at the moment in our implementation, in principle doable by postprocessing of the results:

Data:

<ex.org/bob#me> foaf:name "Bob" .
<alice.org#me> foaf:name "Alice".
<ex.org/bob#me> foaf:nick "Bobby".

SELECT ?Y
WHERE { ?X foaf:name ?Y }
ORDER BY ?Y LIMIT 1

Result: { answer1("Bob",def), answer1("Alice",def) }
Sort answer set by parameter corresponding to ?Y (ORDER BY),
only output first result (LIMIT 1) ⇒ "Alice"
SPARQL Specification: ORDER BY, LIMIT, OFFSET

Not treated at the moment in our implementation, in principle doable by postprocessing of the results:

Data:
<ex.org/bob#me> foaf:name "Bob" .
<alice.org#me> foaf:name "Alice".
<ex.org/bob#me> foaf:nick "Bobby".

SELECT ?Y
WHERE { ?X foaf:name ?Y }
ORDER BY ?Y LIMIT 1

Result: { answer1("Bob",def), answer1("Alice",def) }
Sort answer set by parameter corresponding to ?Y (ORDER BY), only output first result (LIMIT 1) ⇒ "Alice"
Not treated at the moment in our implementation, in principle doable by postprocessing of the results:

Data:
<ex.org/bob#me> foaf:name "Bob" .
<alice.org#me> foaf:name "Alice".
<ex.org/bob#me> foaf:nick "Bobby".

SELECT ?Y
WHERE { ?X foaf:name ?Y }
ORDER BY ?Y LIMIT 1

Result: \{ answer1("Bob",def), answer1("Alice",def) \}
Sort answer set by parameter corresponding to ?Y (ORDER BY), only output first result (LIMIT 1) ⇒ "Alice"
SPARQL Specification: multi-set semantics

1. be careful with projections (SELECT)
2. add some machinery for UNIONs

Data:
:alice foaf:knows _:a .
_:a foaf:name "Bob".  _:a foaf:nick "Bob" .

SELECT ?Y WHERE {?X foaf:name ?Y }

answer1(Y,def) :- triple(X,foaf:name,Y,def).

Answer set: { answer("Bob") },
but expected 2 (identical) solutions!
SPARQL Specification: multi-set semantics

1. be careful with projections (SELECT)

2. add some machinery for UNIONs

Data:
:alice foaf:knows _:a .
_:a foaf:name "Bob".  _:a foaf:nick "Bob" .

SELECT ?Y WHERE {?X foaf:name ?Y } 

answer1(Y,def) :- triple(X,foaf:name,Y,def).

Answer set: { answer("Bob") },
but expected 2 (identical) solutions!
SPARQL Specification: multi-set semantics

1. be careful with projections (SELECT)
2. add some machinery for UNIONS

Data:
:alice foaf:knows _:a .
_:a foaf:name "Bob". _:a foaf:nick "Bob" .

SELECT ?Y WHERE {?X foaf:name ?Y }

answer1(Y,def) :- triple(X,foaf:name,Y,def).

Answer set: { answer("Bob") }, but expected 2 (identical) solutions!
SPARQL Specification: multi-set semantics

1. be careful with projections (SELECT)

2. add some machinery for UNIONS

Data:
:alice foaf:knows _:a .
_:a foaf:name "Bob".  _:a foaf:nick "Bob" .

SELECT ?Y WHERE {?X foaf:name ?Y }

answer1(X,Y,def) :- triple(X,foaf:name,Y,def).

Answer set: { answer1(.,"Bob"), answer1(.,"Bob") },
2 solutions, leave projection to postprocessing!
**SPARQL Specification: multi-set semantics**

1. be careful with projections (SELECT)

2. add some machinery for UNIONS

Data:

```turtle
:alice foaf:knows _:a .
_:a foaf:name "Bob".  _:a foaf:nick "Bob" .
```

```sparql
SELECT ?N
```

answer1(?N,?X,def) :- triple(X,foaf:name,Y ,def).
answer1(?N,?X,def) :- triple(X,foaf:nick,Y ,def).

Answer set: 
{ answer1(..., "Bob"), answer1(..., "Bobby"),
answer1(..., "Bob") },

but expected 4 solutions!
SPARQL Specification: multi-set semantics

1. be careful with projections (SELECT)
2. add some machinery for UNIONs

Data:
alice foaf:knows _:a .
_:a foaf:name "Bob". _:a foaf:nick "Bob" .

SELECT ?N

answer1(?N,?X,def) :- triple(X,foaf:name,Y,def).
answer1(?N,?X,def) :- triple(X,foaf:nick,Y,def).

Answer set: { answer1(..., "Bob"), answer1(..., "Bobby"),
answer1(..., "Bob") },
but expected 4 solutions!
SPARQL Specification: multi-set semantics

1. be careful with projections (SELECT)
2. add some machinery for UNIONs

Data:
:alice foaf:knows _:a .
_:a foaf:name "Bob". _:a foaf:nick "Bob" .

SELECT ?N

answer1(?N,?X,1,def) :- triple(X,foaf:name,Y,def).
answer1(?N,?X,2,def) :- triple(X,foaf:nick,Y,def).

Answer set: { answer1(...,"Bob"), answer1(...,"Bobby"),
answer1(...,"Bob"), answer1(...,"Bob") },
Add a new constant for any "branch" of a UNION.
"select names and email addresses only of those older than 30"

    OPTIONAL { ?X foaf:mbox ?M . FILTER(?Age > 30) }}

Needs 3 case distinctions:

▶ There is an email address and the FILTER is fulfilled (join)
▶ There is an email address and the FILTER is not fulfilled (leave ?M unbound)
▶ There is no email address (leave ?M unbound)
“select names and email addresses only of those older than 30”

```
SELECT ?N ?M WHERE {
  OPTIONAL {
    FILTER(?Age > 30)
  }
}
```

Needs 3 case distinctions:

- There is an email address and the FILTER is fulfilled (join)
- There is an email address and the FILTER is not fulfilled (leave ?M unbound)
- There is no email address (leave ?M unbound)
"select names and email addresses only of those older than 30"

SELECT ?N ?M WHERE {
  OPTIONAL {
    FILTER(?Age > 30)
  }
}

answer1_P(Age,N,M,X,def) :- triple_Q(X,foaf:name,N,def), triple_Q(X,:age,Age,def),
  answer2_P(M,X,def), Age > 30.

answer1_P(Age,N,null,X,def) :- triple_Q(X,foaf:name,N,def),
  triple_Q(X,:age,Age,def),
  answer2_P(M,X,def), not Age > 30.

answer1_P(Age,N,null,X,def) :- triple_Q(X,foaf:name,N,def),
  triple_Q(X,:age,Age,def),
  not answer2'_P(X,def).

answer2_P(M,X,def) :- triple_Q(X,foaf:mbox,M,def).

answer2'_P(X,def) :- answer2_P(M,X,def).

answer_Q(N,M) :- answer1_P(Age,N,M,X,def).
Outline

From SPARQL to LP
- Basic Graph Patterns
- GRAPH Patterns
- FILTERs
- UNION Patterns
- OPTIONAL and Negation as failure

Full SPARQL-Spec compliance
- ORDER BY, LIMIT, OFFSET
- Multi-set semantics
- FILTERs in OPTIONALs

SPARQL++ for Ontology alignment
- Mapping by SPARQL
- Examples
- Implementation
- Example Translation
- RDFS

Wrap-up
Use Case – Ontology Alignment/Mapping

- Typically: Description of correspondences and overlaps between ontological entities (properties, classes, individuals, etc.)
- W3C standards for writing ontologies in place (RDFS, OWL), but limited expressivity for describing mappings.
- Which language to use?
- How to publish mappings/alignments? This is important to make Open Linked Data\(^1\) happen!

We define some useful extensions of SPARQL – SPARQL++ – and our translation towards a language to define such mappings

\(^1\)Combining RDF data that is “out there”, e.g. Sindice, DBPedia, SWPipes etc.
Use Case – Ontology Alignment/Mapping

- Typically: Description of correspondences and overlaps between ontological entities (properties, classes, individuals, etc.)
- W3C standards for writing ontologies in place (RDFS, OWL), but limited expressivity for describing mappings.
- Which language to use?
- How to publish mappings/alignments? This is important to make Open Linked Data¹ happen!

We define some useful extensions of SPARQL – SPARQL++ – and our translation towards a language to define such mappings

¹Combining RDF data that is “out there”, e.g. Sindice, DBPedia, SWPipes etc.
Use Case – Ontology Alignment/Mapping

- Typically: Description of correspondences and overlaps between ontological entities (properties, classes, individuals, etc.)
- W3C standards for writing ontologies in place (RDFS, OWL), but limited expressivity for describing mappings.
- Which language to use?
  - How to publish mappings/alignments? This is important to make *Open Linked Data*\(^1\) happen!

We define some useful extensions of SPARQL – SPARQL++ – and our translation towards a language to define such mappings.

\(^1\)Combining RDF data that is “out there”, e.g. Sindice, DBPedia, SWPipes etc.
Use Case – Ontology Alignment/Mapping

- Typically: Description of correspondences and overlaps between ontological entities (properties, classes, individuals, etc.)
- W3C standards for writing ontologies in place (RDFS, OWL), but limited expressivity for describing mappings.
- Which language to use?
- How to publish mappings/alignments? This is important to make Open Linked Data\(^1\) happen!

We define some useful extensions of SPARQL – SPARQL++ – and our translation towards a language to define such mappings

---

\(^1\)Combining RDF data that is “out there”, e.g. Sindice, DBPedia, SWPipes etc.
Mapping Scenarios

Map from vCard to FOAF:

Expressible by \textit{rdfs:subPropertyOf}:

\begin{verbatim}
VCard:FN rdfs:subPropertyOf foaf:name .
\end{verbatim}
Mapping Scenarios

Map from vCard to FOAF:

vCard

VCard:FN

foaf:Person

Also expressible in RDFS or in OWL DL:

VCard:FN rdfs:subPropertyOf foaf:name.
VCard:FN rdfs:domain foaf:Person.
Mapping Scenarios

Map from vCard to FOAF:

Also expressible in RDFS or in OWL DL:

\[ \text{vCard:FN} \sqsubseteq \text{foaf:name} \]
\[ \exists \text{vCard:FN}. \top \sqsubseteq \text{foaf:Person} \]
Mapping Scenarios

Map from vCard to FOAF:

vCard

VCard:FN

VCard:Given

VCard:Family

FOAF

foaf:name

foaf:Person

Needs string concatenation, not expressible in OWL or RDFS... maybe SWRL can help, but (1) implementations missing (2) no W3C stamp
Mapping Scenarios

Map from vCard to FOAF:

What shall we do here?
Needs conversion from String to rdf:Resource (URI)...how?
Let’s see what SPARQL can do for us...
Observation:
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

CONSTRUCT { Basic triple patterns }  
FROM dataset (source graph)  
WHERE {Pattern}

▶ This may be read as a view definition ...
▶ ... and views can be understood as (mapping) rules

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!
▶ By OPTIONAL patterns you get even non-monotonicity (negation as failure)
▶ By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS? (SPARQL is only defined in terms of simple RDF entailment)
Mapping by SPARQL

Observation:
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}

- This may be read as a view definition ...
- ... and views can be understood as (mapping) rules

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!

- By OPTIONAL patterns you get even non-monotonicity (negation as failure)
- By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS? (SPARQL is only defined in terms of simple RDF entailment)
Mapping by SPARQL

**Observation:**
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

```
CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}
```

- This may be read as a *view* definition ...
- ... and views can be understood as *(mapping) rules*

**Attention:** if you allow such views to mutually refer to each other, you get a recursive rules language!

- By OPTIONAL patterns you get even non-monotonicity (negation as failure)
- By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS? (SPARQL is only defined in terms of simple RDF entailment)
Mapping by SPARQL

Observation:
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

```
CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}
```

- This may be read as a view definition ...
- ... and views can be understood as (mapping) rules

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!

- By OPTIONAL patterns you get even non-monotonicity (negation as failure)
- By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS? (SPARQL is only defined in terms of simple RDF entailment)
Observation:
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}

▶ This may be read as a view definition ...

▶ ... and views can be understood as (mapping) rules

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!

▶ By OPTIONAL patterns you get even non-monotonicity (negation as failure)

▶ By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS? (SPARQL is only defined in terms of simple RDF entailment)
Mapping by SPARQL

Observation:
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

```sparql
CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}
```

▶ This may be read as a view definition ...
▶ ... and views can be understood as (mapping) rules

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!
▶ By OPTIONAL patterns you get even non-monotonicity (negation as failure)
▶ By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS? (SPARQL is only defined in terms of simple RDF entailment)
Mapping by SPARQL

Observation:
SPARQL (Proposed W3C Rec since two weeks, BTW) offers CONSTRUCT queries to generate new graphs from existing ones

CONSTRUCT { Basic triple patterns }
FROM dataset (source graph)
WHERE {Pattern}

▶ This may be read as a view definition ...
▶ ... and views can be understood as (mapping) rules

Attention: if you allow such views to mutually refer to each other, you get a recursive rules language!
▶ By OPTIONAL patterns you get even non-monotonicity (negation as failure)
▶ By bnodes in the CONSTRUCT part, you might run into non-termination issues!

BTW: How can this interact with ontological inferences of OWL and RDFS? (SPARQL is only defined in terms of simple RDF entailment)
Example 1

CONSTRUCT { ?X foaf:name ?Y }
WHERE { ?X VCard:FN ?Y }

Easy!
Example 1

**CONSTRUCT** { ?X foaf:name ?Y }
**WHERE** { ?X VCard:FN ?Y }

Easy!
Example 2

CONSTRUCT { ?X foaf:name ?Y . ?X rdf:type foaf:person . }
WHERE { ?X VCard:FN ?Y }

No problem either.
Example 2

CONSTRUCT { ?X foaf:name ?Y . ?X rdf:type foaf:person . }
WHERE { ?X VCard:FN ?Y }

No problem either.
CONSTRUCT { ?X foaf:name ??? }
Example 3

CONSTRUCT { ?X foaf:name ??? }

How to tackle? FILTERs?
Example 3

```
CONSTRUCT { ?X foaf:name ?FN }  
FILTER( ?FN = fn:concat(?N," ",?F))}

 Doesn’t work :-| FILTERs only bind variables, can’t create new bindings
```
CONSTRUCT { ?X foaf:name fn:concat(?N," ",?F) }

You rather want built-in functions in the CONSTRUCT part. This is what SPARQL++ provides.
Example 3

CONSTRUCT { ?X foaf:name fn:concat(\(?N,"\",\?F) \) }  
WHERE { ?X VCard:Given \?N. ?X VCard:Family \?F  
}  

You rather want built-in functions in the CONSTRUCT part.  
This is what SPARQL++ provides.  
**Attention:** Value generation in the CONSTRUCT part might again raise non-termination issues!
With value generation in CONSTRUCTs and respective built-in support, this becomes easy again in SPARQL++:

```
```
Example 4

With value generation in CONSTRUCTs and respective built-in support, this becomes easy again in SPARQL++:

```
CONSTRUCT { ?X foaf:phone
    rdf:Resource(fn:concat("tel:",fn:encode-for-uri(?T)) . }
WHERE { ?X VCard:tel ?T . }
```
Example 4

With value generation in CONSTRUCTs and respective built-in support, this becomes easy again in SPARQL++:

```
CONSTRUCT { ?X foaf:phone
            rdf:Resource(fn:concat("tel:", fn:encode-for-uri(?T)) . }
WHERE { ?X VCard:tel ?T . }
```
Example 5

We want more: **Aggregates!**

Example: Map from DOAP to RDF Open Source Software Vocabulary:

```
CONSTRUCT { ?P os:latestRelease
WHERE { ?P rdf:type doap:Project . }
```
Example 5

We want more: Aggregates!

Example: Map from DOAP to RDF Open Source Software Vocabulary:

CONSTRUCT { ?P os:latestRelease
WHERE { ?P rdf:type doap:Project . }
Example 6

Note: “Views” – as we use them here for mappings – are also good for defining implicit knowledge within an RDF graph:

Example: “Import” my co-authors in my FOAF file, mapping from myPubl.rdf which uses the Dublin Core (DC) Vocabulary: “I know all my co-authors”

```
foafWithImplicitData.rdf

:me a foaf:Person.
:me foaf:name "Axel Polleres".
CONSTRUCT{ :me foaf:knows _:P . _:P foaf:name ?N } 
FROM <http://www.polleres.net/myPubl.rdf>
:me foaf:knows [foaf:name "Stefan Decker"].
:me foaf:knows [foaf:name "Manfred Hauswirth"].
```

SPARQL++ allows such extended RDF Graphs!
Example 6

Note: “Views” – as we use them here for mappings – are also good for defining implicit knowledge within an RDF graph:

Example: “Import” my co-authors in my FOAF file, mapping from `myPubl.rdf` which uses the Dublin Core (DC) Vocabulary: “I know all my co-authors”

```
CONSTRUCT{ :me foaf:knows _:P . _:P foaf:name ?N }
FROM <http://www.polleres.net/myPubl.rdf>
```

```
:me foaf:knows [foaf:name "Stefan Decker"].
:me foaf:knows [foaf:name "Manfred Hauswirth"].
```

SPARQL++ allows such extended RDF Graphs!
Example 6

Note: “Views” – as we use them here for mappings – are also good for defining implicit knowledge within an RDF graph:

Example: “Import” my co-authors in my FOAF file, mapping from myPubl.rdf which uses the Dublin Core (DC) Vocabulary: “I know all my co-authors”

```
foafWithImplicitData.rdf

:me a foaf:Person.
:me foaf:name "Axel Polleres".
CONSTRUCT{ :me foaf:knows _:P . _:P foaf:name ?N } FROM <http://www.polleres.net/myPubl.rdf>
    ?P dc:author ?N. FILTER(?N != "Axel Polleres"). } :
:me foaf:knows [foaf:name "Stefan Decker"].
:me foaf:knows [foaf:name "Manfred Hauswirth"].
```

SPARQL++ allows such extended RDF Graphs!
Example 6

Note: “Views” – as we use them here for mappings – are also good for defining implicit knowledge within an RDF graph:

Example: “Import” my co-authors in my FOAF file, mapping from myPubl.rdf which uses the Dublin Core (DC) Vocabulary: “I know all my co-authors”

```
foafWithImplicitData.rdf

:me a foaf:Person.
:me foaf:name "Axel Polleres".
CONSTRUCT{ :me foaf:knows _:_P . _:_P foaf:name ?N }
FROM <http://www.polleres.net/myPubl.rdf>
:me foaf:knows [foaf:name "Stefan Decker"].
:me foaf:knows [foaf:name "Manfred Hauswirth"].
```

SPARQL++ allows such extended RDF Graphs!
Goal: you can publish extended RDF Graphs, linked via mappings!
Open Linked data with extended RDF Graphs:

Goal: you can publish extended RDF Graphs, linked via mappings!
Open Linked data with extended RDF Graphs:

Goal: you can publish extended RDF Graphs, linked via mappings!
Open Linked data with extended RDF Graphs:

Goal: you can publish extended RDF Graphs, linked via mappings!
Open Linked data with extended RDF Graphs:

Goal: you can publish extended RDF Graphs, linked via mappings!

Web = HTML + Links
Open Linked data with extended RDF Graphs:

Goal: you can publish extended RDF Graphs, linked via mappings!

Semantic Web = RDF + Mappings
Our Implementation: HEX-Programs

- We again translate (possibly nested and cross-referencing) SPARQL queries to Logic Programs with external atoms (HEX-atoms).
- HEX-programs are Datalog programs with negation as failure and a very generic Built-in mechanism.
- A HEX-program is a set of rules:

\[
 h \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n
\]  

(1)

- where so-called external atoms of the form

\[
 EXT[Input](Output)
\]  

(2)

are allowed.

- Note: External Atoms can take predicates as inputs → More generic than “normal” built-in predicates in logic programming!

\[2\text{Note: Generally, HEX-programms also allow disjunctive rules, but not necessary here.}\]
Our Implementation: HEX-Programs

- We again translate (possibly nested and cross-referencing) SPARQL queries to Logic Programs with external atoms (HEX-atoms).
- HEX-programs are Datalog programs with negation as failure and a very generic Built-in mechanism.
- A HEX-program is a set of rules:

\[ h \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n \]  \hspace{1cm} (1)

- where so-called external atoms of the form

\[ \text{EXT}[\text{Input}](\text{Output}) \]  \hspace{1cm} (2)

are allowed.

- Note: External Atoms can take predicates as inputs → More generic than “normal” built-in predicates in logic programming!

\(^2\)Note: Generally, HEX-programms also allow disjunctive rules, but not necessary here.
Our Implementation: HEX-Programs

- We again translate (possibly nested and cross-referencing) SPARQL queries to Logic Programs with external atoms (HEX-atoms)
- HEX-programs are Datalog programs with negation as failure and a very generic Built-in mechanism.
- A HEX-program is a set of rules:\(^2\)

\[ h ← b_1, \ldots, b_m, \text{ not } b_{m+1}, \ldots \text{ not } b_n \] (1)

- where so-called external atoms of the form

\[ \text{EXT}[\text{Input}](\text{Output}) \] (2)

are allowed.

- Note: External Atoms can take predicates as inputs → More generic than “normal” built-in predicates in logic programming!

\(^2\)Note: Generally, HEX-programms also allow disjunctive rules, but not necessary here.
Our Implementation: HEX-Programs

- We again translate (possibly nested and cross-referencing) SPARQL queries to Logic Programs with external atoms (HEX-atoms).
- HEX-programs are Datalog programs with negation as failure and a very generic Built-in mechanism.
- A HEX-program is a set of rules:

\[
h \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots \text{not } b_n
\]  

(1)

- where so-called *external atoms* of the form

\[
\text{EXT}[\text{Input}](\text{Output})
\]  

(2)

are allowed.

- Note: External Atoms can take *predicates* as inputs → More generic than “normal” built-in predicates in logic programming!

---

\[\text{Note: Generally, HEX-programms also allow disjunctive rules, but not necessary here.}\]
Our Implementation: HEX-Programs

- We again translate (possibly nested and cross-referencing) SPARQL queries to Logic Programs with external atoms (HEX-atoms).
- HEX-programs are Datalog programs with negation as failure and a very generic Built-in mechanism.
- A HEX-program is a set of rules:

\[ h ← b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots \text{ not } b_n \]  

(1)

- where so-called **external atoms** of the form

\[ EXT[Input](Output) \]  

(2)

are allowed.

- **Note:** External Atoms can take predicates as inputs → More generic than “normal” built-in predicates in logic programming!

\[ ^2 \text{Note: Generally, HEX-programms also allow disjunctive rules, but not necessary here.} \]
For the additional features, we need more than just the `rdf` atom from before:

- `rdf[URL](S,P,O)` ... imports all RDF Triples from a given URL.
- `CONCAT[Str_1,...,Str_n](Str)` concatenates Strings.
- `COUNT[Predicate, BindingPattern](Cnt)` ... returns the count of a certain predicate extension, given a certain binding pattern.
- `MAX[Predicate, BindingPattern](MaxVal)` ... returns the is the lexicographically greatest value among the parameters of `Predicate` in the whole extension (`MIN` analogously).
- `SK[Id,V_1,...,V_n](SKTerm)` ... similar to `CONCAT`, but returns a Skolem term, with Skolem function id `Id`. We need this for `bnode` generation in `CONSTRUCTs`.
- ... plus some more for handling `FILTERs` in SPARQL.
SPARQL-specific external Atoms:

For the additional features, we need more than just the `rdf` atom from before:

- `rdf[URL](S,P,O)` ... imports all RDF Triples from a given URL
- `CONCAT[Str₁,...,Strₙ](Str)` concatenates Strings.
- `COUNT[Predicate, BindingPattern](Cnt)` ... returns the count of a certain predicate extension, given a certain binding pattern.
- `MAX[Predicate, BindingPattern](MaxVal)` ... returns the lexicographically greatest value among the parameters of Predicate in the whole extension (MIN analogously).
- `SK[Id,V₁,...,Vₙ](SKTerm)` ... similar to CONCAT, but returns a Skolem term, with Skolem function id `Id`. We need this for bnode generation in CONSTRUCTs.
- ... plus some more for handling FILTERs in SPARQL.
SPARQL-specific external Atoms:

For the additional features, we need more than just the \texttt{rdf} atom from before:

- \texttt{rdf[URL](S,P,O)} \ldots imports all RDF Triples from a given URL
- \texttt{CONCAT[Str_1,\ldots,Str_n](Str)} concatenates Strings.
- \texttt{COUNT[Predicate, BindingPattern](Cnt)} \ldots returns the count of a certain predicate extension, given a certain binding pattern.
- \texttt{MAX[Predicate, BindingPattern](MaxVal)} \ldots returns the is the lexicographically greatest value among the parameters of \texttt{Predicate} in the whole extension (MIN analogously).
- \texttt{SK[Id,V_1,\ldots,V_n](SKTerm)} \ldots similar to \texttt{CONCAT}, but returns a Skolem term, with Skolem function id \texttt{Id}. We need this for bnode generation in \texttt{CONSTRUCTs}.
- \ldots plus some more for handling \texttt{FILTERs} in \texttt{SPARQL}.
SPARQL-specific external Atoms:

For the additional features, we need more than just the \texttt{rdf} atom from before:

- \texttt{rdf[URL](S,P,O)} \ldots imports all RDF Triples from a given URL
- \texttt{CONCAT[Str_1,\ldots,Str_n](Str)} concatenates Strings.
- \texttt{COUNT[Predicate, BindingPattern](Cnt)} \ldots returns the count of a certain predicate extension, given a certain binding pattern.
- \texttt{MAX[Predicate, BindingPattern](MaxVal)} \ldots returns the lexicographically greatest value among the parameters of \texttt{Predicate} in the whole extension (\texttt{MIN} analogously).
- \texttt{SK[Id,V_1,\ldots,V_n](SKTerm)} \ldots similar to \texttt{CONCAT}, but returns a Skolem term, with Skolem function id \texttt{Id}. We need this for \texttt{bnode} generation in CONSTRUCTs.
- \ldots plus some more for handling FILTERs in SPARQL.
SPARQL-specific external Atoms:

For the additional features, we need more than just the \texttt{rdf} atom from before:

- \texttt{rdf[URL](S,P,O)} \ldots imports all RDF Triples from a given URL.
- \texttt{CONCAT[Str}_1,\ldots,\texttt{Str}_n](\texttt{Str}) \ldots concatenates Strings.
- \texttt{COUNT[Predicate, BindingPattern](Cnt)} \ldots returns the count of a certain predicate extension, given a certain binding pattern.
- \texttt{MAX[Predicate, BindingPattern](MaxVal)} \ldots returns the lexicographically greatest value among the parameters of \texttt{Predicate} in the whole extension (\texttt{MIN} analogously).
- \texttt{SK[Id,V}_1,\ldots,\texttt{V}_n](\texttt{SKTerm}) \ldots similar to \texttt{CONCAT}, but returns a Skolem term, with Skolem function id \texttt{Id}. We need this for bnode generation in CONSTRUCTs.

\ldots plus some more for handling \texttt{FILTERs} in SPARQL.
SPARQL-specific external Atoms:

For the additional features, we need more than just the `rdf` atom from before:

- `rdf[URL](S,P,O)` ... imports all RDF Triples from a given URL
- `CONCAT[Str₁,...,Strₙ](Str)` concatenates Strings.
- `COUNT[Predicate, BindingPattern](Cnt)` ... returns the count of a certain predicate extension, given a certain binding pattern.
- `MAX[Predicate, BindingPattern](MaxVal)` ... returns the lexicographically greatest value among the parameters of `Predicate` in the whole extension (`MIN` analogously).
- `SK[Id,V₁,...,Vₙ](SKTerm)` ... similar to `CONCAT`, but returns a Skolem term, with Skolem function id `Id`. We need this for `bnode` generation in `CONSTRUCTs`.
- ... plus some more for handling `FILTERs` in SPARQL.
Demo Translation

Data in myPubl.rdf:
:p1 a :Publ.
:p1 dc:author "Axel Polleres".
:p1 dc:author "Francois Scharffe".
:p1 dc:author "Roman Schindlauer".
...

Query:

CONSTRUCT{ :me foaf:knows _:P . _:P foaf:name ?N } 
FROM <http://www.polleres.net/myPubl.rdf> 
    FILTER(?N != "Axel Polleres") }
Demo Translation

Data in myPubl.rdf:
:p1 a :Publ.
:p1 dc:author "Axel Polleres".
:p1 dc:author "Francois Scharffe".
:p1 dc:author "Roman Schindlauer".
...

Translated HEX Program:

triple(S,P,O) :- &rdf["http://www.polleres.net/myPubl.rdf"](S,P,O).
Demo Translation

Data in myPubl.rdf:

:p1 a :Publ.
:p1 dc:author "Axel Polleres".
:p1 dc:author "Francois Scharffe".
:p1 dc:author "Roman Schindlauer".
...

Translated HEX Program:

\[
\text{triple}(S, P, O) \leftarrow \&\text{rdf}["http://www.polleres.net/myPubl.rdf"](S, P, O).
\]

\[
\text{answer}(N, P) \leftarrow \text{triple}(P, "\text{rdf:}\text{type}", "\text{:Publ}"),
\text{triple}(P, "\text{dc:}\text{author}", N),
N \neq "\text{Axel Polleres}".
\]
Demo Translation

Data in myPubl.rdf:
:p1 a :Publ.
:p1 dc:author "Axel Polleres".
:p1 dc:author "Francois Scharffe".
:p1 dc:author "Roman Schindlauer".
...

Translated HEX Program:

\[
\text{triple}(S,P,O) \leftarrow \text{&rdf["http://www.polleres.net/myPubl.rdf"]}(S,P,O).
\]

\[
\text{answer}(N,P) \leftarrow \text{triple}(P,"\text{rdf:type}",:)\text{Publ}),
\text{triple}(P,"\text{dc:author}"),N),
N \neq "\text{Axel Polleres}".
\]

\[
\text{triple_result}("me","\text{foaf:knows}"\text{Blank_P}) \leftarrow \\
\text{answer}(N,P), \text{&SK[ "#genid_P",N,P]}(\text{Blank_P}).
\]

\[
\text{triple_result}(\text{Blank_P},"\text{foaf:name}"),N) \leftarrow \\
\text{answer}(N,P), \text{&SK[ "#genid_P",N,P]}(\text{Blank_P}).
\]
Demo Translation

Data in myPubl.rdf:
:p1 a :Publ.
:p1 dc:author "Axel Polleres".
:p1 dc:author "Francois Scharffe".
:p1 dc:author "Roman Schindlauer".
...

Result:

triple_result(":me","foaf:knows","#genid_P('Francois Scharffe',:p1)")
triple_result("#genid_P('Francois Scharffe',:p1)","foaf:name","Francois Scharffe")
triple_result(":me","foaf:knows","#genid_P('Roman Schindlauer',:p1)")
triple_result("#genid_P('Roman Schindlauer',:p1)","foaf:name","Roman Schindlauer")
Demo Translation

Data in myPubl.rdf:
:p1 a :Publ.
:p1 dc:author "Axel Polleres".
:p1 dc:author "Francois Scharffe".
:p1 dc:author "Roman Schindlauer".
...

Result:

```
triple_result(":me","foaf:knows","#genid_P('Francois Scharffe',:p1)")
triple_result("#genid_P('Francois Scharffe',:p1)","foaf:name","Francois Scharffe")
triple_result(":me","foaf:knows","#genid_P('Roman Schindlauer',:p1)")
triple_result("#genid_P('Roman Schindlauer',:p1)","foaf:name","Roman Schindlauer")
```

Can in turn be translated back to RDF Triples:

```
:me foaf:knows _:b1.
_:b1 foaf:name "Francois Scharffe".
:me foaf:knows _:b2.
_:b2 foaf:name "Roman Schindlauer".
```
Aggregates Translation:

CONSTRUCT { ?P os:latestRelease
WHERE { ?P rdf:type doap:Project . }

will become:

triple_result(P,os:latestRelease,Va) :- MAX[auxa,P,mask](Va),
            triple(P,rdf:type,doap:Project,def).
answera(P,R,V) :- triple(P,doap:release R,def),
               triple(R,doap:revision,V,def).
Aggregates Translation:

CONSTRUCT { ?P os:latestRelease
WHERE { ?P rdf:type doap:Project . } 

will become:

triple_result(P,os:latestRelease,Va) :- MAX[auxa,P,mask](Va),
    triple(P,rdf:type,doap:Project,def).


answera(P,R,V) :- triple(P,doap:release R,def),
    triple(R,doap:revision,V,def).
Aggregates Translation:

CONSTRUCT { ?P os:latestRelease
WHERE { ?P rdf:type doap:Project . }

will become:

triple_result(P,os:latestRelease,Va) :- MAX[auxa,P,mask](Va),
    triple(P,rdf:type,doap:Project,def).


answera(P,R,V) :- triple(P,doap:release R,def),
    triple(R,doap:revision,V,def).
Aggregates Translation:

CONSTRUCT { ?P os:latestRelease
WHERE { ?P rdf:type doap:Project . }
Aggregates Translation:

CONSTRUCT { ?P os:latestRelease
WHERE { ?P rdf:type doap:Project . }

will become:

\[
\text{triple}\_\text{result}(P, \text{os:latestRelease}, V_a) \leftarrow \text{MAX}[\text{aux}_a(P, \text{mask})](V_a), \quad \\
\text{triple}(P, \text{rdf:type}, \text{doap:Project}, \text{def}).
\]

\[
\text{aux}_a (P, V) \leftarrow \text{answer}_a (P, R, V).
\]

\[
\text{answer}_a (P, R, V) \leftarrow \text{triple}(P, \text{doap:release} R, \text{def}), \\
\text{triple}(R, \text{doap:revision} V, \text{def}).
\]
Aggregates Translation:

CONSTRUCT { ?P os:latestRelease
WHERE { ?P rdf:type doap:Project . } 

will become:

triple_result(P,os:latestRelease,V_a) :- MAX[aux_a,P,mask](V_a), 
    triple(P,rdf:type,doap:Project,def). 

aux_a(P,V) :- answer_a(P,R,V). 

answer_a(P,R,V) :- triple(P,doap:release R,def), 
    triple(R,doap:revision,V,def).
Aggregates Translation:

CONSTRUCT { ?P os:latestRelease
WHERE { ?P rdf:type doap:Project . }

will become:

triple_result(P,os:latestRelease,V_a) :- MAX[aux_a(P,mask)](V_a),
   triple(P,rdf:type,doap:Project,def).
aux_a(P,V) :- answer_a(P,R,V).
answer_a(P,R,V) :- triple(P,doap:release R,def),
   triple(R,doap:revision,V,def).
**Aggregates Translation:**

```
CONSTRUCT { ?P os:latestRelease
WHERE { ?P rdf:type doap:Project . }
```

will become:

```
triple_result(P,os:latestRelease,Va) :- MAX[auxa,P,mask](Va),
    triple(P,rdf:type,doap:Project,def).
answera(P,R,V) :- triple(P,doap:release R,def),
    triple(R,doap:revision,V,def).
```

`aux` predicate used for projection; result of automatic translation.
CONSTRUCT { ?P os:latestRelease
WHERE { ?P rdf:type doap:Project . }

will become:

triple_result(P,os:latestRelease,V) :- MAX[aux_a,P,mask](V),
    triple(P,rdf:type,doap:Project,def).

aux_a(P,V) :- answer_a(P,R,V).
answer_a(P,R,V) :- triple(P,doap:release R,def),
    triple(R,doap:revision,V,def).

aux predicate used for for projection; result of automatic translation.

Find more details on the translation in the paper.
RDFS Inference:

- RDFS Semantics can be expressed in Rules

  So, it is expressible as CONSTRUCT queries

  ```sparql
  CONSTRUCT {?A :subPropertyOf ?C}
  WHERE {?A :subPropertyOf ?B. ?B :subPropertyOf ?C.}
  
  CONSTRUCT {?A :subClassOf ?C}
  
  CONSTRUCT {?X ?B ?Y}
  
  CONSTRUCT {?X rdf:type ?B}
  
  CONSTRUCT {?X rdf:type ?B}
  
  CONSTRUCT {?Y rdf:type ?B}
  
  CONSTRUCT {?X rdf:type ?B}
  
  CONSTRUCT {?Y rdf:type ?B}
  ```

- Simply add these to your extended graph, if RDFS needed. Will be evaluated (recursively) by our translation.
RDFS Inference:

- RDFS Semantics can be expressed in Rules
- So, it is expressible as CONSTRUCT queries

```
CONSTRUCT {?A :subPropertyOf ?C}
  WHERE {?A :subPropertyOf ?B. ?B :subPropertyOf ?C.}
CONSTRUCT {?A :subClassOf ?C}
CONSTRUCT {?X ?B ?Y}
CONSTRUCT {?X rdf:type ?B}
CONSTRUCT {?X rdf:type ?B}
CONSTRUCT {?Y rdf:type ?B}
CONSTRUCT {?X rdf:type ?B}
CONSTRUCT {?Y rdf:type ?B}
```

- Simply add these to you extended graph, if RDFS needed. Will be evaluated (recursively) by our translation.
RDFS Inference:

- RDFS Semantics can be expressed in Rules
  
- So, it is expressible as CONSTRUCT queries

```sparql
CONSTRUCT {?A :subPropertyOf ?C}
  WHERE {?A :subPropertyOf ?B. ?B :subPropertyOf ?C.}
CONSTRUCT {?A :subClassOf ?C}
CONSTRUCT {?X ?B ?Y}
CONSTRUCT {?X rdf:type ?B}
CONSTRUCT {?X rdf:type ?B}
CONSTRUCT {?Y rdf:type ?B}
CONSTRUCT {?X rdf:type ?B}
CONSTRUCT {?Y rdf:type ?B}
```

- Simply add these to your extended graph, if RDFS needed. Will be evaluated (recursively) by our translation.
Outline

From SPARQL to LP
  Basic Graph Patterns
  GRAPH Patterns
  FILTERs
  UNION Patterns
  OPTIONAL and Negation as failure

Full SPARQL-Spec compliance
  ORDER BY, LIMIT, OFFSET
  Multi-set semantics
  FILTERs in OPTIONALs

SPARQL++ for Ontology alignment
  Mapping by SPARQL
  Examples
  Implementation
  Example Translation
  RDFS

Wrap-up
Summary

Take-home message:

- SPARQL can be translated to Logic Programs.
- Application ontology mappings: Current standards don’t provide the right “ingredients” to describe the necessary mappings.
- Extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of “Extended Graphs”, i.e. Mappings+RDF Data in one file, similar to “Networked Graphs” [Schenk and Staab, 2007]

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A “safety condition” for recursive mappings with bnodes and value-generating CONSTRUCTs.

\[^{3}\text{diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins}\]
Summary

**Take-home message:**

- **SPARQL can be translated to Logic Programs.**
- Application ontology mappings: Current standards don’t provide the right “ingredients” to describe the necessary mappings
- extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of “Extended Graphs”, i.e. Mappings+RDF Data in one file, similar to “Networked Graphs” [Schenk and Staab, 2007]

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A “safety condition” for recursive mappings with bnodes and value-generating CONSTRUCTs.

---

\[\text{diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins}\]
Summary

Take-home message:

- SPARQL can be translated to Logic Programs.
- Application ontology mappings: Current standards don’t provide the right “ingredients” to describe the necessary mappings
- extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of “Extended Graphs”, i.e. Mappings+RDF Data in one file, similar to “Networked Graphs” [Schenk and Staab, 2007]

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A “safety condition” for recursive mappings with bnodes and value-generating CONSTRUCTs.

\[diff: \] stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins
Summary

Take-home message:

▶ SPARQL can be translated to Logic Programs.
▶ Application ontology mappings: Current standards don’t provide the right “ingredients” to describe the necessary mappings
▶ extended version of SPARQL, SPARQL++, fills this gap and adds more...
▶ SPARQL++ allows the definition of “Extended Graphs”, i.e. Mappings+RDF Data in one file, similar to “Networked Graphs” [Schenk and Staab, 2007]

Find more details in [Polleres et al., 2007]:
▶ Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
▶ A “safety condition” for recursive mappings with bnodes and value-generating CONSTRUCTs.

\[^{3}\text{diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins}\]
Take-home message:

- SPARQL can be translated to Logic Programs.
- Application ontology mappings: Current standards don’t provide the right “ingredients” to describe the necessary mappings.
- Extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of “Extended Graphs”, i.e. Mappings+RDF Data in one file, similar to “Networked Graphs” [Schenk and Staab, 2007].

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A “safety condition” for recursive mappings with bnodes and value-generating CONSTRUCTs.

\[^3\]diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins
Summary

Take-home message:

- SPARQL can be translated to Logic Programs.
- Application ontology mappings: Current standards don’t provide the right “ingredients” to describe the necessary mappings.
- Extended version of SPARQL, SPARQL++, fills this gap and adds more...
- SPARQL++ allows the definition of “Extended Graphs”, i.e. Mappings+RDF Data in one file, similar to “Networked Graphs” [Schenk and Staab, 2007].

Find more details in [Polleres et al., 2007]:

- Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
- A “safety condition” for recursive mappings with bnodes and value-generating CONSTRUCTs.

\[^{3}\text{diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins}\]
Summary

Take-home message:

▶ SPARQL can be translated to Logic Programs.
▶ Application ontology mappings: Current standards don’t provide the right “ingredients” to describe the necessary mappings
▶ extended version of SPARQL, SPARQL++, fills this gap and adds more...
▶ SPARQL++ allows the definition of “Extended Graphs”, i.e. Mappings+RDF Data in one file, similar to “Networked Graphs” [Schenk and Staab, 2007]³

Find more details in [Polleres et al., 2007]:

▶ Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
▶ A “safety condition” for recursive mappings with bnodes and value-generating CONSTRUCTs.

³diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins
Summary

Take-home message:

▶ SPARQL can be translated to Logic Programs.
▶ Application ontology mappings: Current standards don’t provide the right “ingredients” to describe the necessary mappings
▶ extended version of SPARQL, SPARQL++, fills this gap and adds more...
▶ SPARQL++ allows the definition of “Extended Graphs”, i.e. Mappings+RDF Data in one file, similar to “Networked Graphs” [Schenk and Staab, 2007]

Find more details in [Polleres et al., 2007]:

▶ Formal Semantics of Extended Graphs, based on Stable Model Semantics for HEX-Programs.
▶ A “safety condition” for recursive mappings with bnodes and value-generating CONSTRUCTs.

\(^3\) diff: stable vs. well-founded semantics, safe value-generation allowed, aggregates, built-ins
Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic Web...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)

Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

A. Polleres
Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic Web...
  - ... i.e. allow to publish mappings and implicit RDF data on the Web.
  - As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
  - Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
  - Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
  - Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)


Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

A. Polleres
Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic Web...
  
- ... i.e. allow to publish mappings and implicit RDF data on the Web.

- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.

- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.

- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.

- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)


Stay Tuned: http://www.polleres.net/dlvhex-sparql

Thanks! Questions please! :-(
Next Steps

▶ SPARQL++, Extended Graphs are intended as a means to weave the Semantic Web...
▶ ... i.e. allow to publish mappings and implicit RDF data on the Web.
▶ As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
▶ Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
▶ Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
▶ Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)

Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

A. Polleres
Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic Web...

- ... i.e. allow to publish mappings and implicit RDF data on the Web.

- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.

- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.

- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.

- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)

- More related efforts on the way, e.g.

Stay Tuned: http://www.polleres.net/dlvhex-sparql

Thanks! Questions please! :-)
Next Steps

▶ SPARQL++, Extended Graphs are intended as a means to weave the Semantic Web...
▶ ... i.e. allow to publish mappings and implicit RDF data on the Web.
▶ As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
▶ Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
▶ Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
▶ Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)

Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

A. Polleres
Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic Web...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)


Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

A. Polleres
Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic Web...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)

- More related efforts on the way, e.g.

Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

A. Polleres
Next Steps

- SPARQL++, Extended Graphs are intended as a means to weave the Semantic Web...
- ... i.e. allow to publish mappings and implicit RDF data on the Web.
- As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
- Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
- Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
- Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)
- More related efforts on the way, e.g.

Stay Tuned: http://www.polleres.net/dlvhex-sparql

Thanks! Questions please! :-)

A. Polleres
Next Steps

▶ SPARQL++, Extended Graphs are intended as a means to weave the Semantic Web...
▶ ... i.e. allow to publish mappings and implicit RDF data on the Web.
▶ As the community picks up SPARQL, people will be able to publish mappings for free, without having to learn a new syntax.
▶ Necessary next step: Optimization of distributed querying: We conceive a Linked Open Data Web rather a network of SPARQL++ endpoints than a network of RDF files.
▶ Full SPARQL spec compliance is tedious, as SPARQL semantics is not purely declarative.
▶ Ontological inference beyond RDFS, or OWL Horst at max. unlikely. (Personal opinion: Higher expressivity languages rather important for TBox only, than for instance semantics and query answering)

Stay Tuned: http://www.polleres.net/dlvhex-sparql
Thanks! Questions please! :-)

A. Polleres
References I

Answer set programming for the semantic web.
Tutorial at the European Semantic Web Conference (ESWC), see http://asptut.gibbi.com/.

Semantics and complexity of sparql.

From SPARQL to rules (and back).
In Proceedings of the 16th World Wide Web Conference (WWW2007), Banff, Canada.

SPARQL++ for mapping between RDF vocabularies.

Networked rdf graph networked rdf graphs.