SPARQL Extensions and Outlook

Axel Polleres\textsuperscript{1}

\textsuperscript{1}DERI Galway, National University of Ireland, Galway
axel.polleres@deri.org

European Semantic Web Conference 2007
Outline

Translation to LP, a bit more formal

Next steps? Some possible Examples

Lessons to be learned from SQL?
  Nested queries – Nesting ASK
  Aggregates

Lessons to be learned from Datalog, Rules Languages, etc. ?
  Use SPARQL as rules
  Mixing data and rules – Recursion?
Translation to LP, a bit more formal

Given a query $q = (V, P, DS), DS = (G, G_N)$

```
SELECT V
FROM G
FROM NAMED G_N
WHERE P
```

we denote by $\Pi_q$ the logic program obtained by the translation sketched in the previous Unit, where we give the auxiliary predicates non-ambiguous names, i.e. $\text{answer}_{i_q}$.

Then, the extension of the predicate $\text{answer}_{1_q}$ contains all answer substitutions for $q$. 
Translation to LP, a bit more formal

Given a query $q = (V, P, DS)$, $DS = (G, G_N)$

```
SELECT V
FROM G
FROM NAMED G_N
WHERE P
```

we denote by $\Pi_q$ the logic program obtained by the translation sketched in the previous Unit, where we give the auxiliary predicates non-ambiguous names, i.e. $\text{answer}_i q$.

Then, the extension of the predicate $\text{answer}_1 q$ contains all answer substitutions for $q$. 

Given a query \( q = (V, P, DS), \ DS = (G, G_N) \)

\[
\text{SELECT } V \\
\text{FROM } G \\
\text{FROM NAMED } G_N \\
\text{WHERE } P
\]

we denote by \( \Pi_q \) the logic program obtained by the translation sketched in the previous Unit, where we give the auxiliary predicates non-ambiguous names, i.e. \( \text{answer}_i q \).

Then, the extension of the predicate \( \text{answer}_1 q \) contains all answer substitutions for \( q \).
Example: $q_1 = ( \{ ?E, ?N \},$

$((?X : name ?N) \text{ OPT } (?X : email ?E)),$

$(\{http://alice.org\}, \emptyset))$

SELECT ?N ?E
FROM <http://alice.org>
WHERE { ?X :name ?N
    OPTIONAL {?X :email ?E } }

$\Pi_{q_1} =$

triple(S,P,O,default$_{q_1}$) :- rdf["alice.org"](S,P,O).
answer1$_{q_1}$(E,N,default$_{q_1}$) :- triple(X,":name",N,default$_{q_1}$),
    triple(X,":email",E,default$_{q_1}$).
answer1$_{q_1}$(null,N,default$_{q_1}$) :- triple(X,":name",N,default$_{q_1}$),
    not answer2$_{q_1}$(X).
answer2$_{q_1}$(X) :- triple(X,":email",E,default$_{q_1}$).

More complex queries are decomposed recursively introducing more auxiliary predicates for nested sub-patterns: answer2$_q$, answer3$_q$, answer4$_{q_1}$, answer5$_{q_1}$, ...
Example: $q_1 = ( \{ ?E, ?N \},$

$(((?X : name ?N) \text{ OPT } (?X : email ?E))),$

$(\{ \text{http://alice.org}, \emptyset \})$)

\[
\begin{array}{l}
\text{SELECT } ?N \ ?E \\
\text{FROM } \text{<http://alice.org> } \\
\text{WHERE } \{ \ ?X : \text{name} \ ?N \\
\quad \text{OPTIONAL } \{ \ ?X : \text{email} \ ?E \ \} \ \}
\end{array}
\]

\[\Pi_{q_1} =
\begin{align*}
\text{triple}(S,P,O,\text{default}_{q_1}) & :- \text{rdf["alice.org"]}(S,P,O). \\
\text{answer1}_{q_1}(E,N,\text{default}_{q_1}) & :- \text{triple}(X,"\text{name}",N,\text{default}_{q_1}), \\
& \quad \text{triple}(X,"\text{email}",E,\text{default}_{q_1}). \\
\text{answer1}_{q_1}(\text{null},N,\text{default}_{q_1}) & :- \text{triple}(X,"\text{name}",N,\text{default}_{q_1}), \\
& \quad \text{not answer2}_{q_1}(X). \\
\text{answer2}_{q_1}(X) & :- \text{triple}(X,"\text{email}",E,\text{default}_{q_1}).
\end{align*}
\]

More complex queries are decomposed recursively introducing more auxiliary predicates for nested sub-patterns: \text{answer2}_{q}, \text{answer3}_{q}, \text{answer4}_{q_1}, \text{answer5}_{q_1}, \ldots
Example: \( q_1 = ( \{ ?E, ?N \}, \)
\(( ((?X : name ?N) \text{ OPT } (?X : email ?E)) ), \)
\(( \{ \text{http://alice.org} \}, \emptyset \} ) \)

\[
\begin{align*}
\text{SELECT } ?N \ ?E \\
\text{FROM } \text{<http://alice.org>} \\
\text{WHERE } \{ \ ?X : name ?N \\
\quad \text{OPTIONAL } \{ ?X : email ?E \} \} \\
\end{align*}
\]

\[\Pi_{q_1} = \]
\[
\begin{align*}
\text{triple}(S,P,O,\text{default}_{q_1}) & : - \text{rdf["alice.org"]}(S,P,O). \\
\text{answer1}_{q_1}(E,N,\text{default}_{q_1}) & : - \text{triple}(X,":name",N,\text{default}_{q_1}), \\
& \quad \text{triple}(X,":email",E,\text{default}_{q_1}). \\
\text{answer1}_{q_1}(\text{null},N,\text{default}_{q_1}) & : - \text{triple}(X,":name",N,\text{default}_{q_1}), \\
& \quad \text{not answer2}_{q_1}(X). \\
\text{answer2}_{q_1}(X) & : - \text{triple}(X,":email",E,\text{default}_{q_1}).
\end{align*}
\]

More complex queries are decomposed recursively introducing more auxiliary predicates for nested sub-patterns: \text{answer2}_{q}, \text{answer3}_{q}, \text{answer4}_{q_1}, \text{answer5}_{q_1}, ...
Example: \( q_1 = ( \{ ?E, ?N \}, \langle ((?X : \text{name} ?N) \text{ OPT} (?X : \text{email} ?E)) \rangle, \langle \{ \text{http:/\!/alice.org} \}, \emptyset \} ) \)

SELECT ?N ?E
FROM <http://alice.org>
WHERE { ?X :name ?N
    OPTIONAL {?X :email ?E }
}

\[ \Pi_{q_1} = \]

triple(S,P,O,default_{q_1}) :- rdf["alice.org"](S,P,O).
answer1_{q_1}(E,N,default_{q_1}) :- triple(X,"\text{name}",N,default_{q_1}),
    triple(X,"\text{email}",E,default_{q_1}).
answer1_{q_1}(\text{null},N,default_{q_1}) :- triple(X,"\text{name}",N,default_{q_1}),
    not answer2_{q_1}(X).
answer2_{q_1}(X) :- triple(X,"\text{email}",E,default_{q_1}).

More complex queries are decomposed recursively introducing more auxiliary predicates for nested sub-patterns: answer2_{q}, answer3_{q}, answer4_{q_1}, answer5_{q_1},...
Next steps?

Disclaimer: What follows in this unit is a speculative outlook and does not necessarily reflect the SPARQL working group’s agenda. We discuss in this unit two starting points for such extensions:

▶ Lessons to be learned from SQL
▶ Lessons to be learned from Datalog

Both these partially overlap, and we will discuss how they integrate with the current SPARQL spec by using the translation from the previous unit.
Nested queries are very common in SQL e.g.

```
SELECT ... FROM WHERE EXISTS ( SELECT ... 
```

a simple and very useful extension for SPARQL could be nesting of boolean queries (ASK) in FILTERS:

```
SELECT ... FROM WHERE { P FILTER (ASK P\_ASK) }
```

So, how could we implement e.g.

```
SELECT ?N
FROM <http://alice.org>
WHERE { ?X :name ?N
    FILTER (!((ASK {?X :email ?E })) ) }
```

Note that this give a more elegant solution for “set difference” queries avoiding the OPTIONAL/\!bound combination!
Nested queries are very common in SQL e.g.

```
SELECT ... FROM WHERE EXISTS ( SELECT ... )
```

A simple and very useful extension for SPARQL could be nesting of boolean queries (ASK) in FILTERS:

```
SELECT ... FROM WHERE { P FILTER (ASK P ASK) }
```

So, how could we implement e.g.

```
SELECT ?N
FROM <http://alice.org>
WHERE { ?X :name ?N
    FILTER (!(ASK {?X :email ?E}) ) }
```

Note that this gives a more elegant solution for “set difference” queries avoiding the OPTIONAL/!bound combination!
Nested queries are very common in SQL e.g.

```sql
SELECT ... FROM WHERE EXISTS ( SELECT ...
```

A simple and very useful extension for SPARQL could be nesting of boolean queries (ASK) in FILTERS:

```sparql
SELECT ... FROM WHERE { P FILTER (ASK P ASK) }
```

So, how could we implement e.g.

```sparql
SELECT ?N
FROM <http://alice.org>
WHERE { ?X :name ?N
  FILTER (!(ASK {?X :email ?E}))
}
```

Note that this gives a more elegant solution for "set difference" queries avoiding the OPTIONAL/!bound combination!
Nested queries are very common in SQL e.g.

```
SELECT ... FROM WHERE EXISTS ( SELECT ... 

a simple and very useful extension for SPARQL could be nesting of boolean queries (ASK) in FILTERS:

```

```
SELECT ... FROM WHERE ( P FILTER (ASK P ) )
```

So, how could we implement e.g.

```
SELECT ?N
FROM <http://alice.org>
WHERE { ?X :name ?N
      FILTER (! (ASK {?X :email ?E }) ) } }
```

Note that this gives a more elegant solution for “set difference” queries avoiding the OPTIONAL/!bound combination!
Nested queries are very common in SQL e.g.

```
SELECT ... FROM WHERE EXISTS ( SELECT ... 

```

a simple and very useful extension for SPARQL could be nesting of boolean queries (ASK) in FILTERS:

```
SELECT ... FROM WHERE {
    P FILTER (ASK P

```

So, how could we implement e.g.

```
SELECT ?N
FROM <http://alice.org>
WHERE {
    ?X :name ?N
    FILTER (! (ASK {?X :email ?E })) }

```

Note that this give a more elegant solution for “set difference” queries avoiding the OPTIONAL/!bound combination!
Lessons to be learned from SQL: Nested ASK queries (2/2)

Given query $q = (V, P, DS)$, with sub-pattern $(P_1 \text{ FILTER } (\text{ASK } q_{\text{ASK}}))$ and $q_{\text{ASK}} = (\emptyset, P_{\text{ASK}}, DS_{\text{ASK}})$:

- modularly translate such sub-queries by extending $\Pi_q$ with $\Pi_{q'}$ where $q' = (\text{vars}(P_1) \cap \text{vars}(P_{\text{ASK}}), P_{\text{ASK}}, DS_{\text{ASK}})$
- let $DS_{\text{ASK}}$ default to $DS$ if not specified otherwise.

Example:

```
SELECT ?N
FROM <http://alice.org>
WHERE { ?X :name ?N
    FILTER ( !(\text{ASK} {?X :email ?E }) ) }
```

$\text{vars}(P_1) \cap \text{vars}(P_{\text{ASK}}) = \{X\}$
$q' = (\{?X\}, (?X : email ?E), (\{http : //alice.org\}, \emptyset) )$

$\Pi_q$:

$\text{answer1}_{q'}(X) :- \text{triple}(X, " : email", E, \text{default}).$
$\text{answer1}_{q}(N) :- \text{triple}(X, " : name", N, \text{default}), \text{not } \text{answer1}_{q'}(X).$
Given query $q = (V, P, DS)$, with sub-pattern

$$(P_1 \text{ FILTER (ASK } q_{\text{ASK}}))$$

and $q_{\text{ASK}} = (\emptyset, P_{\text{ASK}}, DS_{\text{ASK}})$:

- modularly translate such sub-queries by extending $\Pi_q$ with $\Pi_{q'}$
  where $q' = (\text{vars}(P_1) \cap \text{vars}(P_{\text{ASK}}), P_{\text{ASK}}, DS_{\text{ASK}})$

- let $DS_{\text{ASK}}$ default to $DS$ if not specified otherwise.

Example:

```sparql
SELECT ?N
FROM <http://alice.org>
WHERE { ?X :name ?N
  FILTER ( !(ASK {?X :email ?E }) ) }
```

\[
\text{vars}(P_1) \cap \text{vars}(P_{\text{ASK}}) = \{X\}
\]

$q' = \{ ?X \}, (?X :email ?E),(\{http://alice.org\}, \emptyset) \}$

$$\Pi_q:
answer1_{q'}(X) :- \text{triple}(X,"\text{:email"}, E, \text{default}).
answer1_{q}(N) :- \text{triple}(X,"\text{:name"},N, \text{default}), \text{not answer1}_{q'}(X).$$
Given query \( q = (V, P, DS) \), with sub-pattern 
\( (P_1 \ \text{FILTER} \ (\text{ASK} \ q_{\text{ASK}})) \) and 
\( q_{\text{ASK}} = (\emptyset, P_{\text{ASK}}, DS_{\text{ASK}}) \):

▶ modularly translate such sub-queries by extending \( \Pi_q \) with \( \Pi_{q'} \) where \( q' = (\text{vars}(P_1) \cap \text{vars}(P_{\text{ASK}}), P_{\text{ASK}}, DS_{\text{ASK}}) \)
▶ let \( DS_{\text{ASK}} \) default to \( DS \) if not specified otherwise.

Example:

```
SELECT ?N
FROM <http://alice.org>
WHERE { ?X :name ?N
       FILTER ( !(ASK {?X :email ?E }) ) }
```

\[ \text{vars}(P_1) \cap \text{vars}(P_{\text{ASK}}) = \{ X \} \]
\[ q' = ( \{ ?X \}, (?X : email ?E), (\{http : //alice.org\}, \emptyset) ) \]

\( \Pi_q : \)
\( \text{answer1}_{q'}(X) :- \text{triple}(X,"\text{name}" ,N, default). \)
\( \text{answer1}_q(N) :- \text{triple}(X,"\text{name}" ,N, default), \text{not answer1}_{q'}(X). \)
Lessons to be learned from SQL: Nested ASK queries (2/2)

Given query \( q = (V, P, DS) \), with sub-pattern
\( (P_1 \text{ FILTER (ASK } q_{\text{ASK}})) \) and \( q_{\text{ASK}} = (\emptyset, P_{\text{ASK}}, DS_{\text{ASK}}) \):

- modularly translate such sub-queries by extending \( \Pi_q \) with \( \Pi_{q'} \) where \( q' = (\text{vars}(P_1) \cap \text{vars}(P_{\text{ASK}}), P_{\text{ASK}}, DS_{\text{ASK}}) \)
- let \( DS_{\text{ASK}} \) default to \( DS \) if not specified otherwise.

Example:

```
SELECT ?N
FROM <http://alice.org>
WHERE { ?X :name ?N
  FILTER ( !(ASK {?X :email ?E }) ) }
```

\( \text{vars}(P_1) \cap \text{vars}(P_{\text{ASK}}) = \{X\} \)

\( q' = ( \{?X\}, (?X : email ?E), ({http://alice.org}, \emptyset) ) \)

\( \Pi_q: \)

```
answer1_{q'}(X) :- triple(X,":email",E, default).
answer1_{q}(N) :- triple(X,":name",N, default), not answer1_{q'}(X).
```
Lessons to be learned from SQL: Nested ASK queries (2/2)

Given query \( q = (V, P, DS) \), with sub-pattern
\[
(P_1 \text{ FILTER (ASK } q_{ASK}) \text{)) and } q_{ASK} = (\emptyset, P_{ASK}, DS_{ASK}):
\]

- modularly translate such sub-queries by extending \( \Pi_q \) with \( \Pi_{q'} \) where \( q' = (\text{vars}(P_1) \cap \text{vars}(P_{ASK}), P_{ASK}, DS_{ASK}) \)
- let \( DS_{ASK} \) default to \( DS \) if not specified otherwise.

Example:

\[
\text{SELECT ?N}
\text{FROM <http://alice.org>}
\text{WHERE { ?X :name ?N}
FILTER ( !(ASK {?X :email ?E }) )}}
\]

\( \text{vars}(P_1) \cap \text{vars}(P_{ASK}) = \{X\} \)

\( q' = ( \{?X\},(?X : email ?E),(\{http : //alice.org\},\emptyset) ) \)

\( \Pi_q: \)

\( \text{answer1}_{q'}(X) :- \text{triple}(X,"\text{:email"},E, \text{default}). \)
\( \text{answer1}_q(N) :- \text{triple}(X,"\text{:name"},N, \text{default}), \text{not answer1}_{q'}(X). \)
Given query \( q = (V, P, DS) \), with sub-pattern 
\((P_1 \text{ FILTER } (\text{ASK } q_{\text{ASK}})) \) and 
\( q_{\text{ASK}} = (\emptyset, P_{\text{ASK}}, DS_{\text{ASK}}) \):

- modularly translate such sub-queries by extending \( \Pi_q \) with \( \Pi_{q'} \) where 
  \( q' = (\text{vars}(P_1) \cap \text{vars}(P_{\text{ASK}}), P_{\text{ASK}}, DS_{\text{ASK}}) \)

- let \( DS_{\text{ASK}} \) default to \( DS \) if not specified otherwise.

Example:

```sql
SELECT ?N
FROM <http://alice.org>
WHERE { ?X :name ?N
       FILTER ( !(\text{ASK } {?X :email ?E }) ) }
```

\( \text{vars}(P_1) \cap \text{vars}(P_{\text{ASK}}) = \{X\} \)

\( q' = (\{?X\}, (?X : email ?E), (\{http : //alice.org\}, \emptyset)) \)

\( \Pi_q: \)

\( \text{answer1}_{q'}(X) :- \text{triple}(X,"\text{:email"}, E, \text{default}). \)
\( \text{answer1}_{q}(N) :- \text{triple}(X,"\text{:name"}, N, \text{default}), \neg \text{answer1}_{q'}(X). \)
Example Count:

```sql
SELECT ?X
FROM <http://example.org/lotsOfFOAFData.rdf>
WHERE { ?X a person .

    FILTER (
    )
}
```

- Possible argument against:
  - UNA, closed world!
  - Implementation needs to take special care for counting semantics (duplicates)

- Arguments in favor:
  - COUNT is already expressible!
  - closed world is already there! (OPTIONAL+!bound)
Example Count:

```sql
SELECT ?X
FROM <http://example.org/lotsOfFOAFData.rdf>
WHERE { ?X a person .

    FILTER ( COUNT{ ?Y : ?X foaf:knows ?Y} > 3 )
}
```

- Possible argument against:
  - UNA, closed world!
  - Implementation needs to take special care for counting semantics (duplicates)
- Arguments in favor:
  - COUNT is already expressible!
  - closed world is already there! (OPTIONAL+!bound)
Example Count:

```sql
SELECT ?X
FROM <http://example.org/lotsOfFOAFData.rdf>
WHERE { ?X a person .
    FILTER (! ( ?Y1 = ?Y2 ) AND
             ! ( ?Y1 = ?Y3 ) AND
             ! ( ?Y2 = ?Y3 ) )
}
```

- Possible argument against:
  - UNA, closed world!
  - Implementation needs to take special care for counting semantics (duplicates)

- Arguments in favor:
  - COUNT is already expressible!
  - closed world is already there! (OPTIONAL+!bound)
Aggregates: A mockup syntax proposal:

- **Symbolic Set**: Expression
  \[ \{ Vars : Pattern \} \]
  of a list \( Vars \) of variables and a pattern \( P \)
  (e.g. \( \{ ?K : ?X \text{ foaf:knows } ?K \} \)).

- **Aggregate Function**: Expression
  \[ f \{ Vars : Pattern \} \]
  where
  - \( f \in \{ \text{COUNT}, \text{MIN}, \text{MAX}, \text{SUM}, \text{TIMES} \} \), and
  - \( \{ Vars : Pattern \} \) is a symbolic set
    (e.g. \( \text{COUNT} \{ ?K : ?X \text{ foaf:knows } ?K \} \)).
Aggregates: A mockup syntax proposal:

- **Symbolic Set:** Expression

  \{ \text{Vars} : \text{Pattern} \}

  of a list \text{Vars} of variables and a pattern \text{P}
  (e.g. \{ ?K : ?X foaf:knows ?K \}).

- **Aggregate Function:** Expression

  \( f \{ \text{Vars} : \text{Pattern} \} \)

  where

  - \( f \in \{ \text{COUNT}, \text{MIN}, \text{MAX}, \text{SUM}, \text{TIMES} \} \), and
  - \{ \text{Vars} : \text{Pattern} \} \) is a symbolic set
    (e.g. \text{COUNT} \{ ?K : ?X foaf:knows ?K \}).
Aggregates: A mockup syntax proposal:

- **Symbolic Set**: Expression
  \[
  \{ Vars : Pattern \}
  \]
  of a list \( Vars \) of variables and a pattern \( P \)
  (e.g. \{ ?K : ?X foaf:knows ?K \}).

- **Aggregate Function**: Expression
  \[
  f \{ Vars : Pattern \}
  \]
  where
  - \( f \in \{ COUNT, MIN, MAX, SUM, TIMES \} \), and
  - \{ \( Vars \) : \( Pattern \) \} is a symbolic set
    (e.g. \( COUNT\{ ?K : ?X foaf:knows ?K \} \)).
Aggregates: A mockup syntax proposal:

❖ **Symbolic Set:** Expression

\{ Vars : Pattern \}

of a list Vars of variables and a pattern P
(e.g. \{ ?K : ?X foaf:knows ?K \}).

❖ **Aggregate Function:** Expression

\( f \{ Vars : Pattern \} \)

where

❖ \( f \in \{ \text{COUNT, MIN, MAX, SUM, TIMES} \} \), and

❖ \( \{ Vars : Pattern \} \) is a symbolic set
(e.g. \( \text{COUNT}\{ ?K : ?X foaf:knows ?K \} \))
Aggregates: A mockup syntax proposal:

- **Symbolic Set:** Expression
  
  \[ \{ \text{Vars} : \text{Pattern} \} \]

  of a list \text{Vars} of variables and a pattern \( P \)
  
  (e.g. \{ ?K : ?X foaf:knows ?K \}).

- **Aggregate Function:** Expression
  
  \[ f \{ \text{Vars} : \text{Pattern} \} \]

  where
  
  - \( f \in \{ \text{COUNT, MIN, MAX, SUM, TIMES} \} \), and
  
  - \{ \text{Vars} : \text{Pattern} \} \) is a symbolic set
  
  (e.g. \text{COUNT}\{ ?K : ?X foaf:knows ?K \})
Aggregates: A mockup syntax proposal:

- **Symbolic Set:** Expression
  \[
  \{ \text{Vars : Pattern} \}
  \]
of a list \text{Vars} of variables and a pattern \text{P}
(e.g. \{ ?K : ?X foaf:knows ?K \}).

- **Aggregate Function:** Expression
  \[ f \{ \text{Vars : Pattern} \} \]
where
  - \( f \in \{ \text{COUNT, MIN, MAX, SUM, TIMES} \} \), and
  - \{ \text{Vars : Pattern} \} is a symbolic set
    (e.g. \text{COUNT}\{ ?K : ?X foaf:knows ?K \})
Aggregate Atom: Expression

\[ \text{Agg Atom ::= val } \circ f \{ \text{Vars : Pattern} \} \]
\[ | f \{ \text{Vars : Conj} \} \circ val \]
\[ | val_l \circ_l f \{ \text{Vars : Pattern} \} \circ_r val_u \]

where

- \( \text{val}, \text{val}_l, \text{val}_u \) are constants or variables,
- \( \circ \in \{ <, >, \leq, \geq, = \} \),
- \( \circ_l, \circ_r \in \{ <, \leq \} \), and
- \( f \{ \text{Vars : Pattern} \} \) is an aggregate function (e.g. \( \text{COUNT} \{ ?K : \text{?X foaf:knows ?K} \} < 3 \))
Aggregate Atom: Expression

\[ \text{Agg Atom ::= } \text{val } \odot \ f \{ \text{Vars : Pattern} \} \]
\[ \mid f \{ \text{Vars : Conj} \} \odot \text{val} \]
\[ \mid \text{val}_l \odot_l f \{ \text{Vars : Pattern} \} \odot_r \text{val}_u \]

where

- \( \text{val}, \text{val}_l, \text{val}_u \) are constants or variables,
- \( \odot \in \{<, >, \leq, \geq, =\} \),
- \( \odot_l, \odot_r \in \{<, \leq\} \), and
- \( f \{ \text{Vars : Pattern} \} \) is an aggregate function
  (e.g. \text{COUNT}\{ ?K : ?X foaf:knows ?K \} < 3 \)

A. Polleres – SPARQL Extensions and Outlook 10 / 16
Aggregate Atom: Expression

\[\text{Agg Atom ::= val} \odot f \{\text{Vars : Pattern}\} \]
\[\mid f \{\text{Vars : Conj}\} \odot \text{val} \]
\[\mid \text{val}_l \odot_l f \{\text{Vars : Pattern}\} \odot_r \text{val}_u\]

where

- \(\text{val, val}_l, \text{val}_u\) are constants or variables,
- \(\odot\) \(\in\) \{<,>, \leq, \geq, =\},
- \(\odot_l, \odot_r\) \(\in\) \{<, \leq\}, and
- \(f \{\text{Vars : Pattern}\}\) is an aggregate function
  (e.g. \(\text{COUNT}\{?K : ?X \text{ foaf:knows} ?K\} < 3\))
Lessons to be learned from SQL: Aggregates (3/4)

- **Aggregate Atom**: Expression

\[
\text{Agg\_Atom ::= } \text{val} \odot f \{ \text{Vars} : \text{Pattern} \} \\
\| \text{f} \{ \text{Vars} : \text{Conj} \} \odot \text{val} \\
\| \text{val}_{l} \odot_{l} f \{ \text{Vars} : \text{Pattern} \} \odot_{r} \text{val}_{u}
\]

where

- \( \text{val, val}_{l}, \text{val}_{u} \) are constants or variables,
- \( \odot \in \{ <, >, \leq, \geq, = \} \),
- \( \odot_{l}, \odot_{r} \in \{ <, \leq \} \), and
- \( f \{ \text{Vars} : \text{Pattern} \} \) is an aggregate function
  (e.g. \( \text{COUNT}\{ \ ?K : \ ?X \text{ foaf:knows} \ ?K \ \} < 3 \))
Aggregate Atom: Expression

\[ Agg\_Atom ::= val \odot f \{ Vars : Pattern \} \]
\[ \quad | \quad f \{ Vars : Conj \} \odot val \]
\[ \quad | \quad val_l \odot_l f \{ Vars : Pattern \} \odot_r val_u \]

where

- \( val, val_l, val_u \) are constants or variables,
- \( \odot, \odot_l, \odot_r \in \{ <, >, \leq, \geq, = \} \),
- \( \odot_l, \odot_r \in \{ <, \leq \} \), and
- \( f \{ Vars : Pattern \} \) is an aggregate function
  (e.g. \( \text{COUNT}\{ ?K : ?X \text{ foaf:knows} ?K \} \) < 3)
Aggregate Atom: Expression

\[ \text{Agg Atom ::= val} \odot f \{ \text{Vars} : \text{Pattern} \} \]
\[ | f \{ \text{Vars} : \text{Conj} \} \odot \text{val} \]
\[ | \text{val}_{l} \odot_{l} f \{ \text{Vars} : \text{Pattern} \} \odot_{r} \text{val}_{u} \]

where

- \( \text{val}, \text{val}_l, \text{val}_u \) are constants or variables,
- \( \odot \in \{ <, >, \leq, \geq, = \} \),
- \( \odot_{l}, \odot_{r} \in \{ <, \leq \} \), and
- \( f \{ \text{Vars} : \text{Pattern} \} \) is an aggregate function

(e.g. \( \text{COUNT}\{ ?K : ?X \text{ foaf:knows } ?K \} \) \( < 3 \))
Examples of usage:

- **Aggregate atoms in FILTERs:**
  
  ```sql
  SELECT ?X
  WHERE {
    ?X a foaf:Person .
  ```

- **Aggregate atoms in result forms:**
  
  ```sql
  WHERE { ?X a foaf:Person . }
  ```

**Implementation:**

- The aggregate syntax chosen here is a straightforward extension of the aggregate syntax of DLV → implementation possible by a slight extension of the LP translation with DLV’s aggregates.

**Semantics:**

- Semantics of Aggregates in LP, even possibly involving recursive rules agreed [Faber et al., 2004]
Lessons to be learned from SQL: Aggregates (4/4)

Examples of usage:

- Aggregate atoms in FILTERs:
  ```sql
  SELECT ?X
  WHERE {
    ?X a foaf:Person .
  ```

- Aggregate atoms in result forms:
  ```sql
  WHERE { ?X a foaf:Person . )
  ```

Implementation:

- The aggregate syntax chosen here is a straight-forward extension of the aggregate syntax of DLV → implementation possible by a slight extension of the LP translation with DLV’s aggregates.

Semantics:

- Semantics of Aggregates in LP, even possibly involving recursive rules agreed [Faber et al., 2004]
Examples of usage:

- **Aggregate atoms in FILTERs:**
  ```sparql
  SELECT ?X
  WHERE
  { ?X a foaf:Person .
  ```

- **Aggregate atoms in result forms:**
  ```sparql
  WHERE { ?X a foaf:Person .}
  ```

**Implementation:**

- The aggregate syntax chosen here is a straightforward extension of
  the aggregate syntax of DLV → implementation possible by a slight
  extension of the LP translation with DLV’s aggregates.

**Semantics:**

- Semantics of Aggregates in LP, even possibly involving recursive
  rules agreed [Faber et al., 2004]
Examples of usage:

- **Aggregate atoms in FILTERs:**
  
  ```sql
  SELECT ?X
  WHERE {
  ?X a foaf:Person .
  }
  ```

- **Aggregate atoms in result forms:**
  
  ```sql
  WHERE {
  ?X a foaf:Person .
  }
  ```

**Implementation:**

- The aggregate syntax chosen here is a straight-forward extension of the aggregate syntax of DLV → implementation possible by a slight extension of the LP translation with DLV’s aggregates.

**Semantics:**

- Semantics of Aggregates in LP, even possibly involving recursive rules agreed [Faber et al., 2004]
Examples of usage:

- Aggregate atoms in FILTERs:
  ```sql
  SELECT ?X
  WHERE { ?X a foaf:Person .
  ```

- Aggregate atoms in result forms:
  ```sql
  WHERE { ?X a foaf:Person . )
  ```

Implementation:

- The aggregate syntax chosen here is a straightforward extension of the aggregate syntax of DLV → implementation possible by a slight extension of the LP translation with DLV’s aggregates.

Semantics:

- Semantics of Aggregates in LP, even possibly involving recursive rules agreed [Faber et al., 2004]
CONSTRUCTs themselves may be viewed as rules over RDF. How to handle CONSTRUCT in the outlined translation to LP?

CONSTRUCT { ?X foaf:name ?Y . ?X a foaf:Person . }  
WHERE { ?X vCard:FN ?Y }.

For blanknode-free CONSTRUCTs our translation can be simply extended:

\[
\text{triple}(X,\text{foaf:name},Y,\text{constructed}) :- \\
\text{triple}(X,\text{rdf:type},\text{foaf:Person},\text{default}).
\]

and export the RDF triples from predicate

\[
\text{triple}(S,P,O,\text{constructed})
\]

in post-processing to get the constructed RDF graph.
CONSTRUCTs themselves may be viewed as rules over RDF. How to handle CONSTRUCT in the outlined translation to LP?

```
CONSTRUCT { ?X foaf:name ?Y . ?X a foaf:Person . }
WHERE { ?X vCard:FN ?Y }.
```

For blanknode-free CONSTRUCTs our translation can be simply extended:

```
triple(X,foaf:name,Y,constructed) :-
    triple(X,rdf:type,foaf:Person,default).
```

and export the RDF triples from predicate

```
triple(S,P,O,constructed)
```

in post-processing to get the constructed RDF graph.
CONSTRUCTs themselves may be viewed as rules over RDF. How to handle CONSTRUCT in the outlined translation to LP?

CONSTRUCT { ?X foaf:name ?Y . ?X a foaf:Person . } 
WHERE { ?X vCard:FN ?Y }.

For blanknode-free CONSTRUCTs our translation can be simply extended:

\[
\text{triple}(X, \text{foaf:name}, Y, \text{constructed}) :\leftarrow \text{triple}(X, \text{rdf:type}, \text{foaf:Person}, \text{default}).
\]

and export the RDF triples from predicate

\[
\text{triple}(S, P, O, \text{constructed})
\]

in post-processing to get the constructed RDF graph.
More interesting: With this translation, we get for free a way to process mixed RDF and SPARQL CONSTRUCTs in ONE file.

Mock-up syntax, mixing TURTLE and SPARQL to describe implicit data or mappings within RDF:

```
foafWithImplicitData.rdf

:me a foaf:Person.
:me foaf:name "Axel Polleres".
CONSTRUCT{ :me foaf:knows ?X }
FROM <http://www.deri.ie/about/team>
WHERE { ?X a foaf:Person. }
:me foaf:knows [foaf:name "Marcelo Arenas"],
[foaf:name "Claudio Gutierrez"],
[foaf:name "Bijan Parsia"],
[foaf:name "Jorge Perez"],
[foaf:name "Andy Seaborne"].
```

\footnote{see e.g. RIF use case 2.10, http://www.w3.org/TR/rif-ucr/}
More interesting: With this translation, we get for free a way to process mixed RDF and SPARQL CONSTRUCTs in ONE file.

Mock-up syntax, mixing TURTLE and SPARQL to describe implicit data or mappings within RDF:\footnote{see e.g. RIF use case 2.10, http://www.w3.org/TR/rif-ucr/}:

```
foafWithImplicitData.rdf

:me a foaf:Person.
:me foaf:name "Axel Polleres".
CONSTRUCT{ :me foaf:knows ?X }
FROM <http://www.deri.ie/about/team>
WHERE { ?X a foaf:Person. }
:me foaf:knows [foaf:name "Marcelo Arenas"],
               [foaf:name "Claudio Gutierrez"],
               [foaf:name "Bijan Parsia"],
               [foaf:name "Jorge Perez"],
               [foaf:name "Andy Seaborne"].
```
More interesting: With this translation, we get for free a way to process mixed RDF and SPARQL CONSTRUCTs in ONE file.

Mock-up syntax, mixing TURTLE and SPARQL to describe implicit data or mappings within RDF¹:

```
foafWithImplicitData.rdf

:me a foaf:Person.
:me foaf:name "Axel Polleres".
CONSTRUCT{ :me foaf:knows ?X } FROM <http://www.deri.ie/about/team> WHERE { ?X a foaf:Person. } :me foaf:knows [foaf:name "Marcelo Arenas"],
[foaf:name "Claudio Gutierrez"],
[foaf:name "Bijan Parsia"],
[foaf:name "Jorge Perez"],
[foaf:name "Andy Seaborne"].
```

¹see e.g. RIF use case 2.10, http://www.w3.org/TR/rif-ucr/
Attention! If you apply the translation to LP and two RDF+CONSTRUCT files refer mutually to each other, you might get a recursive program!

- even non-stratified negation as failure!
- two basic semantics for such “networked RDF graphs” possible:
  - well-founded [Schenk and Staab, 2007]
  - stable [Polleres, 2007]
etc., etc.

These were just some ideas for useful extensions!
More to come! Up to you!
Opens up interesting research directions!
etc., etc.

These were just some ideas for useful extensions!
More to come! Up to you!
Opens up interesting research directions!
etc., etc.

These were just some ideas for useful extensions!
More to come! Up to you!
Opens up interesting research directions!
etc., etc.

These were just some ideas for useful extensions!
More to come! Up to you!
Opens up interesting research directions!

Now let’s get back to the next logical step...
etc., etc.

These were just some ideas for useful extensions!
More to come! Up to you!
Opens up interesting research directions!

Now let’s get back to the next logical step... . . .
...how to combine with OWL and RDFS?
etc., etc.

These were just some ideas for useful extensions!
More to come! Up to you!
Opens up interesting research directions!

Now let’s get back to the next logical step... 
...how to combine with OWL and RDFS?

As it turns out, not so simple! Bijan, the stage is yours!
References

