SPARQL Extensions and Outlook

Axel Polleres[1]

[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 Π_q the logic program obtained by the translation sketched in the previous Unit, where we give the auxiliary predicates non-ambiguous names, i.e. answeri_q.

Then, the extension of the predicate answer1$_q$ 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 Π_q the logic program obtained by the translation sketched in the previous Unit, where we give the auxiliary predicates non-ambiguous names, i.e. $\texttt{answer}i_q$.

Then, the extension of the predicate $\texttt{answer1}_q$ 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 Π_q the logic program obtained by the translation sketched in the previous Unit, where we give the auxiliary predicates non-ambiguous names, i.e. $\texttt{answer}i_q$.

Then, the extension of the predicate $\texttt{answer1}_q$ 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_q1) :- rdf["alice.org"](S,P,O).
answer1_q1(E,N,default_q1) :- triple(X,":name",N,default_q1),
                              triple(X,":email",E,default_q1).
answer1_q1(null,N,default_q1) :- triple(X,":name",N,default_q1),
                                 not answer2_q1(X).
answer2_q1(X) :- triple(X,":email",E,default_q1).
```

More complex queries are decomposed recursively introducing more auxiliary predicates for nested sub-patterns: $\texttt{answer2}_{q}$, $\texttt{answer3}_{q}$, $\texttt{answer4}_{q_1}$, $\texttt{answer5}_{q_1}$, ...

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 } }
```

$\Pi_{q_1} =$

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

More complex queries are decomposed recursively introducing more auxiliary predicates for nested sub-patterns: $\texttt{answer2}_{q}$, $\texttt{answer3}_{q}$, $\texttt{answer4}_{q_1}$, $\texttt{answer5}_{q_1}$, ...

Example: $q_1 = (\{?E, ?N\},$
$(((?X \ \ :name\ ?N)\ \mathrm{OPT}\ (?X \ \ :email\ ?E))),$
$(\{http://alice.org\}, \emptyset)\ )$

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

$\Pi_{q_1} =$

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

More complex queries are decomposed recursively introducing more auxiliary predicates for nested sub-patterns: $\texttt{answer2}_{q}$, $\texttt{answer3}_{q}$, $\texttt{answer4}_{q_1}$, $\texttt{answer5}_{q_1}$, ...

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_q1) :- rdf["alice.org"](S,P,O).
answer1_q1(E,N,default_q1) :- triple(X,":name",N,default_q1),
                              triple(X,":email",E,default_q1).
answer1_q1(null,N,default_q1) :- triple(X,":name",N,default_q1),
                                 not answer2_q1(X).
answer2_q1(X) :- triple(X,":email",E,default_q1).
```

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

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.

Lessons to be learned from SQL: Nested ASK queries (1/2)

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!

Lessons to be learned from SQL: Nested ASK queries (1/2)

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!

Lessons to be learned from SQL: Nested ASK queries (1/2)

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!

Lessons to be learned from SQL: Nested ASK queries (1/2)

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!

Lessons to be learned from SQL: Nested ASK queries (1/2)

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!

Lessons to be learned from SQL: Nested ASK queries (2/2)

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

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

Example:

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

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

Π_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 $(P_1 \text{ FILTER } (\text{ASK } q_{\text{ASK}}))$ and $q_{\text{ASK}} = (\emptyset, P_{\text{ASK}}, DS_{\text{ASK}})$:

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

Example:

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

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

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

Π_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
$(P_1$ FILTER (ASK $q_{\text{ASK}}))$ and $q_{\text{ASK}} = (\emptyset, P_{\text{ASK}}, DS_{\text{ASK}})$:

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

Example:

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

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

Π_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
$(P_1$ FILTER (ASK $q_{\text{ASK}}))$ and $q_{\text{ASK}} = (\emptyset, P_{\text{ASK}}, DS_{\text{ASK}})$:

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

Example:

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

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

Π_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 $(P_1 \;\text{FILTER (ASK}\; q_{\text{ASK}}))$ and $q_{\text{ASK}} = (\emptyset, P_{\text{ASK}}, DS_{\text{ASK}})$:

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

Example:

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

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

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

Π_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 $(P_1 \text{ FILTER } (\text{ASK } q_{\text{ASK}}))$ and $q_{\text{ASK}} = (\emptyset, P_{\text{ASK}}, DS_{\text{ASK}})$:

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

Example:

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

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

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

Π_q:

$\text{answer1}_{q'}(\text{X})$:- triple(X,":email",E, default).

$\text{answer1}_{q}(\text{N})$:- triple(X,":name",N, default), not $\text{answer1}_{q'}(\text{X})$.

Lessons to be learned from SQL: Aggregates (1/4)

Example Count:

```
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)

Lessons to be learned from SQL: Aggregates (1/4)

Example Count:

```
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)

Lessons to be learned from SQL: Aggregates (1/4)

Example Count:

```
SELECT ?X
FROM <http://example.org/lotsOfFOAFData.rdf>
WHERE { ?X a person .
        ?X foaf:knows ?Y1 , ?Y2, ?Y3 .
        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)

Lessons to be learned from SQL: Aggregates (2/4)

Aggregates: A mockup syntax proposal:

- **Symbolic Set:** Expression

  $$\{\mathit{Vars} : \mathit{Pattern}\}$$

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

- **Aggregate Function:** Expression

  $$f\,\{\mathit{Vars} : \mathit{Pattern}\}$$

  where

  - $f \in \{COUNT, MIN, MAX, SUM, TIMES\}$, and
  - $\{\mathit{Vars} : \mathit{Pattern}\}$ 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

  - $f \in \{COUNT, MIN, MAX, SUM, TIMES\}$, and
  - $\{Vars : Pattern\}$ is a symbolic set
    (e.g. `COUNT{ ?K : ?X foaf:knows ?K }`)

Lessons to be learned from SQL: Aggregates (2/4)

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 }`)

Lessons to be learned from SQL: Aggregates (2/4)

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 }`)

Lessons to be learned from SQL: Aggregates (2/4)

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 }`)

Lessons to be learned from SQL: Aggregates (2/4)

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 }`)

- **Aggregate Atom:** Expression

$$
\begin{array}{rll}
Agg\_Atom ::= & val \ \circledcirc \ f\,\{Vars : Pattern\} \\
| & f\,\{Vars : Conj\} \ \circledcirc \ val \\
| & val_l \ \circledcirc_l \ f\,\{Vars : Pattern\} \ \circledcirc_r \ val_u
\end{array}
$$

where

- val, val_l, val_u are constants or variables,
- $\circledcirc \in \{<, >, \leq, \geq, =\}$,
- $\circledcirc_l, \circledcirc_r \in \{<, \leq\}$, and
- $f\,\{Vars : Pattern\}$ is an aggregate function (e.g. `COUNT{ ?K : ?X foaf:knows ?K }< 3`)

Lessons to be learned from SQL: Aggregates (3/4)

- **Aggregate Atom:** Expression

$$
\begin{array}{rcl}
Agg\_Atom ::= & & val \ \circledcirc \ f\,\{Vars : Pattern\} \\
 & | & f\,\{Vars : Conj\} \ \circledcirc \ val \\
 & | & val_l \ \circledcirc_l \ f\,\{Vars : Pattern\} \ \circledcirc_r \ val_u
\end{array}
$$

where

- val, val_l, val_u are constants or variables,
- $\circledcirc \in \{<, >, \leq, \geq, =\}$,
- $\circledcirc_l, \circledcirc_r \in \{<, \leq\}$, and
- $f\,\{Vars : Pattern\}$ is an aggregate function
  (e.g. `COUNT{ ?K : ?X foaf:knows ?K }< 3`)

Lessons to be learned from SQL: Aggregates (3/4)

- **Aggregate Atom:** Expression

$$
\begin{array}{rcl}
Agg\_Atom ::= & & val \ \circledcirc \ f\{Vars : Pattern\} \\
& \mid & f\{Vars : Conj\} \ \circledcirc \ val \\
& \mid & val_l \ \circledcirc_l f\{Vars : Pattern\} \ \circledcirc_r \ val_u
\end{array}
$$

  where

  - val, val_l, val_u are constants or variables,
  - $\circledcirc \in \{<, >, \leq, \geq, =\}$,
  - $\circledcirc_l, \circledcirc_r \in \{<, \leq\}$, and
  - $f\{Vars : Pattern\}$ is an aggregate function (e.g. `COUNT{ ?K : ?X foaf:knows ?K }< 3`)

Lessons to be learned from SQL: Aggregates (3/4)

- **Aggregate Atom:** Expression

$$
\begin{array}{rcl}
Agg\_Atom ::= & & val \ \circledcirc \ f \, \{ Vars : Pattern \} \\
& | & f \, \{ Vars : Conj \} \ \circledcirc \ val \\
& | & val_l \ \circledcirc_l \, f \, \{ Vars : Pattern \} \ \circledcirc_r \ val_u
\end{array}
$$

  where

  - val, val_l, val_u are constants or variables,
  - $\circledcirc \in \{ <, >, \leq, \geq, = \}$,
  - $\circledcirc_l, \circledcirc_r \in \{ <, \leq \}$, and
  - $f \, \{ Vars : Pattern \}$ is an aggregate function (e.g. `COUNT{ ?K : ?X foaf:knows ?K }< 3`)

- **Aggregate Atom:** Expression

$$
\begin{array}{rcl}
Agg\_Atom ::= & & val \ \circledcirc \ f\,\{Vars : Pattern\} \\
& | & f\,\{Vars : Conj\} \ \circledcirc \ val \\
& | & val_l \ \circledcirc_l \, f\,\{Vars : Pattern\} \ \circledcirc_r \ val_u
\end{array}
$$

  where

  - val, val_l, val_u are constants or variables,
  - $\circledcirc \in \{<, >, \leq, \geq, =\}$,
  - $\circledcirc_l, \circledcirc_r \in \{<, \leq\}$, and
  - $f\,\{Vars : Pattern\}$ is an aggregate function (e.g. `COUNT{ ?K : ?X foaf:knows ?K }< 3`)

Lessons to be learned from SQL: Aggregates (3/4)

- **Aggregate Atom:** Expression

$$\begin{array}{rcl} Agg\_Atom ::= & & val \;\circledcirc\; f\,\{Vars : Pattern\} \\ & | & f\,\{Vars : Conj\} \;\circledcirc\; val \\ & | & val_l \;\circledcirc_l\; f\,\{Vars : Pattern\} \;\circledcirc_r\; val_u \end{array}$$

  where

  - val, val_l, val_u are constants or variables,
  - $\circledcirc \in \{<, >, \leq, \geq, =\}$,
  - $\circledcirc_l, \circledcirc_r \in \{<, \leq\}$, and
  - $f\,\{Vars : Pattern\}$ is an aggregate function (e.g. `COUNT{ ?K : ?X foaf:knows ?K }< 3`)

Lessons to be learned from SQL: Aggregates (4/4)

Examples of usage:

- Aggregate atoms in FILTERs:

```
SELECT ?X
WHERE { ?X a foaf:Person .
        FILTER ( COUNT{ ?K : ?X foaf:knows ?K } }< 3 )
```

- Aggregate atoms in result forms:

```
SELECT ?X COUNT{ ?K : ?X foaf:knows ?K } }
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]

Lessons to be learned from SQL: Aggregates (4/4)

Examples of usage:

- Aggregate atoms in FILTERs:

```
SELECT ?X
WHERE { ?X a foaf:Person .
        FILTER ( COUNT{ ?K : ?X foaf:knows ?K } }< 3 )
```

- Aggregate atoms in result forms:

```
SELECT ?X COUNT{ ?K : ?X foaf:knows ?K } }
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]

Lessons to be learned from SQL: Aggregates (4/4)

Examples of usage:

- Aggregate atoms in FILTERs:

```
SELECT ?X
WHERE { ?X a foaf:Person .
        FILTER ( COUNT{ ?K : ?X foaf:knows ?K } }< 3 )
```

- Aggregate atoms in result forms:

```
SELECT ?X COUNT{ ?K : ?X foaf:knows ?K } }
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]

Lessons to be learned from SQL: Aggregates (4/4)

Examples of usage:

- Aggregate atoms in FILTERs:

```
SELECT ?X
WHERE { ?X a foaf:Person .
        FILTER ( COUNT{ ?K : ?X foaf:knows ?K } }< 3 )
```

- Aggregate atoms in result forms:

```
SELECT ?X COUNT{ ?K : ?X foaf:knows ?K } }
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]

Lessons to be learned from SQL: Aggregates (4/4)

Examples of usage:

- Aggregate atoms in FILTERs:

```
SELECT ?X
WHERE { ?X a foaf:Person .
        FILTER ( COUNT{ ?K : ?X foaf:knows ?K } }< 3 )
```

- Aggregate atoms in result forms:

```
SELECT ?X COUNT{ ?K : ?X foaf:knows ?K } }
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]

CONSTRUCT 1/3

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

CONSTRUCT 1/3

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

CONSTRUCT 1/3

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

CONSTRUCT 2/3

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[1]:

foafWithImplicitdData.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"].
```

[1] see e.g. RIF use case 2.10, http://www.w3.org/TR/rif-ucr/

CONSTRUCT 2/3

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[1]:

foafWithImplicitdData.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"].
```

[1] see e.g. RIF use case 2.10, http://www.w3.org/TR/rif-ucr/

CONSTRUCT 2/3

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[1]:

foafWithImplicitdData.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"].
```

[1] see e.g. RIF use case 2.10, http://www.w3.org/TR/rif-ucr/

CONSTRUCT 3/3

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!

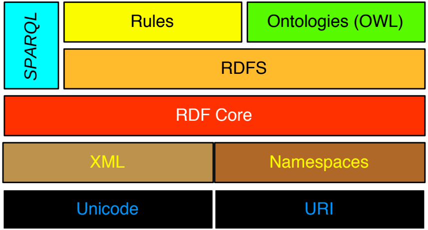
etc., etc.

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!

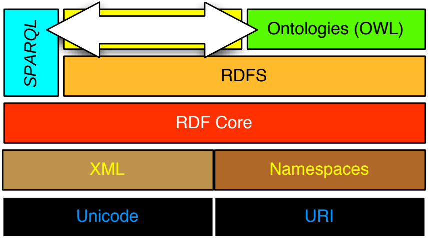
Now let's get back to the next logical step. . .



etc., etc.

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

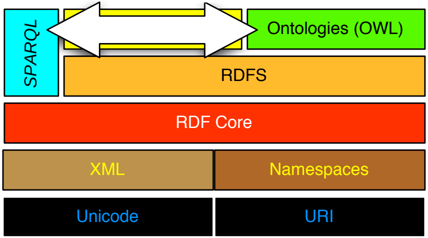
Now let's get back to the next logical step...
...how to combine with OWL and RDFS?



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

Faber, W., Leone, N., and Pfeifer, G. (2004).
Recursive aggregates in disjunctive logic programs: Semantics and complexity.
In Alferes, J. J. and Leite, J., editors, *Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004)*, number 3229 in Lecture Notes in AI (LNAI), pages 200–212. Springer Verlag.

Polleres, A. (2007).
From SPARQL to rules (and back).
In *Proceedings of the 16th World Wide Web Conference (WWW2007)*, Banff, Canada.
Extended technical report version available at
http://www.polleres.net/publications/GIA-TR-2006-11-28.pdf.

Schenk, S. and Staab, S. (2007).
Networked rdf graph networked rdf graphs.
Technical Report 3/2007, Universsity of Koblenz.
available at http://www.uni-koblenz.de/~sschenk/publications/2006/ngtr.pdf.