# dlvhex-sparql: A SPARQL-compliant Query Engine based on dlvhex

Axel Polleres[1]

[1]Digital Enterprise Research Institute (DERI), National University of Ireland, Galway
axel.polleres@deri.org

ALPSWS 2007

joint work with **Roman Schindlauer**
Univ. della Calabria, Rende, Italy and Vienna Univ. of Technology, Austria

Supported by EU FP6 project inContext, and SFI Lion project

# Outline

# Outline

# dlvhex

- a flexible plugin-framework for the DLV engine
- extends Answer Set Programming by external atoms
- implemented plugins
  - for importing Semantic Web data (RDF)
  - for calling DL reasoners (OWL)
  - etc.

# dlvhex Syntax

- *external atoms*

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m)$$

where $Y_1, \ldots, Y_n$ are "input" parameters and $(X_1, \ldots, X_m)$ is the output tuple.

- Rules:

$$h \;\; :- \;\; b_1, \ldots, b_m, not\; b_{m+1}, \ldots not\; b_n.$$

where $h$ and $b_i$ $(1 \leq i \leq n)$ are atoms, $b_k$ $(1 \leq k \leq m)$ either atoms or external atoms

# dlvhex Syntax

- *external atoms*

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m)$$

  where $Y_1, \ldots, Y_n$ are “input” parameters and $(X_1, \ldots, X_m)$ is the output tuple.

- Rules:

$$h \text{ :- } b_1, \ldots, b_m, \mathit{not}\ b_{m+1}, \ldots \mathit{not}\ b_n.$$

  where $h$ and $b_i$ $(1 \leq i \leq n)$ are atoms, $b_k$ $(1 \leq k \leq m)$ either atoms or external atoms

# dlvhex Semantics

- semantics of dlvhex generalizes the answer-set semantics
- external predicates similar to function calls, but can have multiple "return" tuples
- We use particularly 2 external predicates in this work:
  - `&rdf`$[i](s, p, o)$ is true if $(s, p, o)$ is an RDF triple *entailed* by the RDF graph which is accessibly at IRI $i$.
  - `&sk`$[id, v_1, \ldots, v_n](sk_{n+1})$ computes a unique, new "Skolem"-like term $id(v_1, \ldots, v_n)$, from its input parameters.

# dlvhex Semantics

- semantics of dlvhex generalizes the answer-set semantics
- external predicates similar to function calls, but can have multiple "return" tuples
- We use particularly 2 external predicates in this work:
  - `&rdf`$[i](s, p, o)$ is true if $(s, p, o)$ is an RDF triple *entailed* by the RDF graph which is accessibly at IRI $i$.
  - `&sk`$[id, v_1, \ldots, v_n](sk_{n+1})$ computes a unique, new "Skolem"-like term $id(v_1, \ldots, v_n)$, from its input parameters.

# dlvhex Semantics

- semantics of dlvhex generalizes the answer-set semantics
- external predicates similar to function calls, but can have multiple “return” tuples
- We use particularly 2 external predicates in this work:
  - &rdf$[i](s, p, o)$ is true if $(s, p, o)$ is an RDF triple *entailed* by the RDF graph which is accessibly at IRI $i$.
  - &sk$[id, v_1, \ldots, v_n](sk_{n+1})$ computes a unique, new “Skolem”-like term $id(v_1, \ldots, v_n)$, from its input parameters.

# dlvhex Semantics

- semantics of dlvhex generalizes the answer-set semantics
- external predicates similar to function calls, but can have multiple "return" tuples
- We use particularly 2 external predicates in this work:
  - $\texttt{\&rdf}[i](s, p, o)$ is true if $(s, p, o)$ is an RDF triple *entailed* by the RDF graph which is accessibly at IRI $i$.
  - $\texttt{\&sk}[id, v_1, \ldots, v_n](sk_{n+1})$ computes a unique, new "Skolem"-like term $id(v_1, \ldots, v_n)$, from its input parameters.

# dlvhex Semantics

- semantics of dlvhex generalizes the answer-set semantics
- external predicates similar to function calls, but can have multiple "return" tuples
- We use particularly 2 external predicates in this work:
  - `&rdf`$[i](s, p, o)$ is true if $(s, p, o)$ is an RDF triple *entailed* by the RDF graph which is accessibly at IRI $i$.
  - `&sk`$[id, v_1, \ldots, v_n](sk_{n+1})$ computes a unique, new "Skolem"-like term $id(v_1, \ldots, v_n)$, from its input parameters.

# dlvhex Semantics

- semantics of dlvhex generalizes the answer-set semantics
- external predicates similar to function calls, but can have multiple “return” tuples
- We use particularly 2 external predicates in this work:
  - `&rdf`$[i](s, p, o)$ is true if $(s, p, o)$ is an RDF triple *entailed* by the RDF graph which is accessibly at IRI $i$.
  - `&sk`$[id, v_1, \ldots, v_n](sk_{n+1})$ computes a unique, new “Skolem”-like term $id(v_1, \ldots, v_n)$, from its input parameters.

# SQL and Datalog

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

  Example: Two tables containing adressbooks
  `myAddr(Name, Street, City, Telephone)`
  `yourAddr(Name, Address)`

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

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

- That was easy... Now what about SPARQL?

# SQL and Datalog

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

Example: Two tables containing adressbooks

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

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

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

- That was easy... Now what about SPARQL?

# SQL and Datalog

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

  Example: Two tables containing adressbooks
  `myAddr(Name, Street, City, Telephone)`
  `yourAddr(Name, Address)`

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

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

- That was easy... Now what about SPARQL?

# SQL and Datalog

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

  Example: Two tables containing adressbooks
  `myAddr(Name, Street, City, Telephone)`
  `yourAddr(Name, Address)`

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

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

- That was easy... Now what about SPARQL?

# SQL and Datalog

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

  Example: Two tables containing adressbooks
  `myAddr(Name, Street, City, Telephone)`
  `yourAddr(Name, Address)`

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

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

  ```
  ?- answer1(Name).
  ```

- That was easy... Now what about SPARQL?

# RDF

- SPARQL (W3C Candidate Recommentation), a query language for RDF
- RDF is sets of (S, P, O) triples, often written in the following notation:

```
<axel> <foaf:knows> _:x .
_:x foaf:name "Roman" .
<axel> <rdf:type> <foaf:Person> .
<axel> <:age> "33"^^<xsd:integer> .
```

- special thing: “blank” nodes (_:x) are kind of existential variables in the data, to represent incomplete data, may be read:

$$\exists X.triple(axel, foaf{:}knows, X) \wedge triple(X, foaf{:}name, "Roman") \wedge \ldots$$

- this is somewhat different from SQL.
- How to get RDF data into dlvhex? We use the &rdf external atom:

```
{triple(S,P,O) :- &rdf["http://ex.org/bob.rdf"](S,P,O).}
```

# RDF

- SPARQL (W3C Candidate Recommentation), a query language for RDF
- RDF is sets of (S, P, O) triples, often written in the following notation:

```
<axel> <foaf:knows> _:x .
_:x foaf:name "Roman" .
<axel> <rdf:type> <foaf:Person> .
<axel> <:age> "33"^^<xsd:integer> .
```

- special thing: "blank" nodes (_:x) are kind of existential variables in the data, to represent incomplete data, may be read:

$$\exists X.triple(axel, foaf{:}knows, X) \wedge triple(X, foaf{:}name, "Roman") \wedge \ldots$$

- this is somewhat different from SQL.
- How to get RDF data into dlvhex? We use the &rdf external atom:

```
{triple(S,P,O) :- &rdf["http://ex.org/bob.rdf"](S,P,O).}
```

# RDF

- SPARQL (W3C Candidate Recommentation), a query language for RDF
- RDF is sets of (S, P, O) triples, often written in the following notation:

```
<axel> <foaf:knows> _:x .
_:x foaf:name "Roman" .
<axel> <rdf:type> <foaf:Person> .
<axel> <:age> "33"^^<xsd:integer> .
```

- special thing: "blank" nodes (`_:x`) are kind of existential variables in the data, to represent incomplete data, may be read:

$$\exists X.triple(axel, foaf{:}knows, X) \wedge triple(X, foaf{:}name, \text{"}Roman\text{"}) \wedge \ldots$$

- this is somewhat different from SQL.
- How to get RDF data into dlvhex? We use the `&rdf` external atom:

```
{triple(S,P,O) :- &rdf["http://ex.org/bob.rdf"](S,P,O).}
```

# Outline

# From SPARQL to dlvhex: Basic Graph Patterns

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

Basic Graph patterns = simple conjunctive queries:

*"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 . }
```

```
triple(S,P,O,def) :- &rdf["http://ex.org/bob"](S,P,O).
triple(S,P,O,def) :- &rdf["http://alice.org"](S,P,O).
answer1(X,Y,def)  :- triple(X,"rdf:type","foaf:Person",def),
                     triple(X,"foaf:name",Y,def).

?- answer1(X,Y,def).
```

# From SPARQL to dlvhex: Basic Graph Patterns

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

Basic Graph patterns = simple conjunctive queries:

*"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 . }
```

```
triple(S,P,O,def) :- &rdf["http://ex.org/bob"](S,P,O).
triple(S,P,O,def) :- &rdf["http://alice.org"](S,P,O).
answer1(X,Y,def)  :- triple(X,"rdf:type","foaf:Person",def),
                     triple(X,"foaf:name",Y,def).

?- answer1(X,Y,def).
```

# From SPARQL to dlvhex: Basic Graph Patterns

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

Basic Graph patterns = simple conjunctive queries:

*"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 . }
```

```
triple(S,P,O,def) :- &rdf["http://ex.org/bob"](S,P,O).
triple(S,P,O,def) :- &rdf["http://alice.org"](S,P,O).
answer1(X,Y,def)  :- triple(X,"rdf:type","foaf:Person",def),
                     triple(X,"foaf:name",Y,def).

?- answer1(X,Y,def).
```

# From SPARQL to dlvhex: Basic Graph Patterns

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

Basic Graph patterns = simple conjunctive queries:

*"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 . }
```

```
triple(S,P,O,def) :- &rdf["http://ex.org/bob"](S,P,O).
triple(S,P,O,def) :- &rdf["http://alice.org"](S,P,O).
answer1(X,Y,def)  :- triple(X,"rdf:type","foaf:Person",def),
                     triple(X,"foaf:name",Y,def).

?- answer1(X,Y,def).
```

# From SPARQL to dlvhex: Basic Graph Patterns

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

Basic Graph patterns = simple conjunctive queries:

*"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 . }
```

```
triple(S,P,O,def) :- &rdf["http://ex.org/bob"](S,P,O).
triple(S,P,O,def) :- &rdf["http://alice.org"](S,P,O).
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>
WHERE { ?G foaf:maker ?X .
         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),
                    triple(X,"foaf:knows",Y,G).
```

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>
WHERE { ?G foaf:maker ?X .
        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),
                    triple(X,"foaf:knows",Y,G).
```

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>
WHERE { ?G foaf:maker ?X .
         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),
                    triple(X,"foaf:knows",Y,G).
```

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>
WHERE { ?G foaf:maker ?X .
         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),
                    triple(X,"foaf:knows",Y,G).
```

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>
WHERE { ?G foaf:maker ?X .
        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),
                    triple(X,"foaf:knows",Y,G).
```

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>
WHERE { ?G foaf:maker ?X .
         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),
                    triple(X,"foaf:knows",Y,G).
```

For legibility we left out the `http://` prefix

# From SPARQL to dlvhex: FILTERs

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

```
SELECT ?X
FROM ...
WHERE { ?X foaf:mbox ?M . ?X :age ?Age .
        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.

# From SPARQL to dlvhex: FILTERs

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

```
SELECT ?X
FROM ...
WHERE { ?X foaf:mbox ?M . ?X :age ?Age .
          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.

# From SPARQL to dlvhex: FILTERs

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

```
SELECT ?X
FROM ...
WHERE { ?X foaf:mbox ?M .
          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.

# From SPARQL to dlvhex: UNION Patterns 1/2

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).
```

# From SPARQL to dlvhex: UNION Patterns 1/2

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).
```

# From SPARQL to dlvhex: UNION Patterns 1/2

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).
```

# From SPARQL to dlvhex: UNION Patterns 2/2

What if variables of the of 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" .
<ex.org/bob#me> foaf:nick "Bobby".
```

Result:

| ?X | ?Y | ?Z |
|---|---|---|
| <alice.org#me> | "Alice" | |
| <ex.org/bob#me> | "Bob" | |
| <ex.org/bob#me> | | "Bobby" |

# From SPARQL to dlvhex: UNION Patterns 2/2

What if variables of the of 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" .
<ex.org/bob#me> foaf:nick "Bobby".
```

Result:

| ?X | ?Y | ?Z |
|---|---|---|
| <alice.org#me> | "Alice" | |
| <ex.org/bob#me> | "Bob" | |
| <ex.org/bob#me> | | "Bobby" |

# From SPARQL to dlvhex: UNION Patterns 2/2

What if variables of the of 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" .
<ex.org/bob#me> foaf:nick "Bobby".
```

Result:

| ?X | ?Y | ?Z |
|---|---|---|
| <alice.org#me> | "Alice" | null |
| <ex.org/bob#me> | "Bob" | null |
| <ex.org/bob#me> | null | "Bobby" |

# From SPARQL to dlvhex: UNION Patterns 2/2

What if variables of the of 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).
answer1(X,null,Z,def) :- triple(X,"foaf:nick",Z,def).
```

# From SPARQL to dlvhex: UNION Patterns 2/2

What if variables of the of 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).
answer1(X,null,Z,def) :- triple(X,"foaf:nick",Z,def).
```

# From SPARQL to dlvhex: *OPTIONAL* Patterns

"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**:

$\{P_1$ `OPTIONAL` $\{P_2\}\}$: $\quad M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

where $M_1$ and $M_2$ are variable binding for $P_1$ and $P_2$, resp.

# From SPARQL to dlvhex: *OPTIONAL* Patterns

"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**:

$\{P_1$ `OPTIONAL` $\{P_2\}\}$: $\quad M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

where $M_1$ and $M_2$ are variable binding for $P_1$ and $P_2$, resp.

# From SPARQL to dlvhex: *OPTIONAL* Patterns

"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**:

$\{P_1 \texttt{ OPTIONAL } \{P_2\}\}$: $\quad M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

where $M_1$ and $M_2$ are variable binding for $P_1$ and $P_2$, resp.

# SPARQL's OPTIONAL has "negation as failure", hidden:

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

"select all persons *without* an email address"

```
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 OPTIONAL and !bound

"select all persons *without* an email address"

```
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 OPTIONAL and !bound

"select all persons *without* an email address"

```
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".

# From SPARQL to dlvhex: OPTIONAL Patterns

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

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

```
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.

Note: Additional machinery needed for special OPTIONAL queries... out of scope of this short paper, see [Polleres, WWW2007].

# From SPARQL to dlvhex: OPTIONAL Patterns

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

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

```
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.

Note: Additional machinery needed for special OPTIONAL queries... out of scope of this short paper, see [Polleres, WWW2007].

# From SPARQL to dlvhex: OPTIONAL Patterns

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

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

```
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.

Note: Additional machinery needed for special OPTIONAL queries... out of scope of this short paper, see [Polleres, WWW2007]

# From SPARQL to dlvhex: OPTIONAL Patterns

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

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

```
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.

Note: Additional machinery needed for special OPTIONAL queries... out of scope of this short paper, see [Polleres, WWW2007].

# From SPARQL to dlvhex: OPTIONAL Patterns

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

Recall: ($P_1$ OPT $P_2$): $M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

```
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.

Note: Additional machinery needed for special OPTIONAL queries... out of scope of this short paper, see [Polleres, WWW2007]

# From SPARQL to dlvhex: OPTIONAL Patterns

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

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

```
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.

Note: Additional machinery needed for special OPTIONAL queries... out of scope of this short paper, see [Polleres, WWW2007]

# From SPARQL to dlvhex: OPTIONAL Patterns

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

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

```
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.

Note: Additional machinery needed for special OPTIONAL queries... out of scope of this short paper, see [Polleres, WWW2007]

# Outline

# SPARQL Specification compliance

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

Some peculiarities are hidden in the SPARL specification document

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.

# SPARQL Specification compliance

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

Some peculiarities are hidden in the SPARL specification document

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.

# SPARQL Specification compliance

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

Some peculiarities are hidden in the SPARL specification document

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.

# SPARQL Specification compliance

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

Some peculiarities are hidden in the SPARL specification document

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.

# SPARQL Specification compliance

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

Some peculiarities are hidden in the SPARL specification document

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.

# SPARQL Specification compliance

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

Some peculiarities are hidden in the SPARL specification document

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.

# 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 (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 (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 (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 (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:

```
:bob foaf:name "Bob" .  :bob foaf:nick "Bobby" .
: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:

```
:bob foaf:name "Bob" .  :bob foaf:nick "Bobby" .
: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:

```
:bob foaf:name "Bob" .  :bob foaf:nick "Bobby" .
: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:

```
:bob foaf:name "Bob" .  :bob foaf:nick "Bobby" .
: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:

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

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

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

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

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

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

# SPARQL Specification: FILTER expressions in OPTIONAL patterns

*"select names and email addresses only of those older than 30"*

```
SELECT ?N ?M WHERE { ?X foaf:name ?N . ?X :age ?Age .
                     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 { ?X foaf:name ?N . ?X :age ?Age .
                     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 { ?X foaf:name ?N . ?X :age ?Age .
                     OPTIONAL { ?X foaf:mbox ?M . 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).
```

# SPARQL Specification: CONSTRUCT queries and blank nodes

How to deal with this one?

```
CONSTRUCT  _:b a foaf:Agent. _:b foaf:name ?N. ?Doc foaf:maker _:b.  FROM ...
    WHERE  ?Doc dc:creator ?N.
```

CONSTRUCT queries create new triples (similar to views in Rel. DBs).

For blank nodes in CONSTRUCTs, we need Skolem terms as blank node identifiers!

```
answer1(Doc,N,def)  :-  triple_Q(Doc,dc:creator,N,def).
triple_Res(BLANK_b,rdf:type,foaf:Agent,res)  :-  answer1(Doc,N,def),
                                                 &sk[b,Doc,N](BLANK_b).
triple_Res(BLANK_b,foaf:name,N,res)  :-  answer1(Doc,N,def),
                                         &sk[b,Doc,N](BLANK_b).
triple_Res(Doc,foaf:maker,BLANK_b,res)  :-  answer1(Doc,N,def),
                                            &sk[b,Doc,N](BLANK_b).
```

# SPARQL Specification: CONSTRUCT queries and blank nodes

How to deal with this one?

```
CONSTRUCT  _:b a foaf:Agent. _:b foaf:name ?N. ?Doc foaf:maker _:b.  FROM ...
    WHERE  ?Doc dc:creator ?N.
```

CONSTRUCT queries create new triples (similar to views in Rel. DBs).

For blank nodes in CONSTRUCTs, we need Skolem terms as blank node identifiers!

```
answer1(Doc,N,def)  :-  triple_Q(Doc,dc:creator,N,def).
triple_Res(BLANK_b,rdf:type,foaf:Agent,res)  :-  answer1(Doc,N,def),
                                                  &sk[b,Doc,N](BLANK_b).
triple_Res(BLANK_b,foaf:name,N,res)  :-  answer1(Doc,N,def),
                                          &sk[b,Doc,N](BLANK_b).
triple_Res(Doc,foaf:maker,BLANK_b,res)  :-  answer1(Doc,N,def),
                                             &sk[b,Doc,N](BLANK_b).
```

# SPARQL Specification: CONSTRUCT queries and blank nodes

How to deal with this one?

```
CONSTRUCT  _:b a foaf:Agent. _:b foaf:name ?N. ?Doc foaf:maker _:b.  FROM ...
    WHERE  ?Doc dc:creator ?N.
```

CONSTRUCT queries create new triples (similar to views in Rel. DBs).

For blank nodes in CONSTRUCTs, we need Skolem terms as blank node identifiers!

```
answer1(Doc,N,def)  :-  triple_Q(Doc,dc:creator,N,def).
triple_Res(BLANK_b,rdf:type,foaf:Agent,res)  :-  answer1(Doc,N,def),
                                                 &sk[b,Doc,N](BLANK_b).
triple_Res(BLANK_b,foaf:name,N,res)  :-  answer1(Doc,N,def),
                                         &sk[b,Doc,N](BLANK_b).
triple_Res(Doc,foaf:maker,BLANK_b,res)  :-  answer1(Doc,N,def),
                                            &sk[b,Doc,N](BLANK_b).
```

# SPARQL Specification: CONSTRUCT queries and blank nodes

How to deal with this one?

```
CONSTRUCT  _:b a foaf:Agent. _:b foaf:name ?N. ?Doc foaf:maker _:b.  FROM ...
    WHERE  ?Doc dc:creator ?N.
```

CONSTRUCT queries create new triples (similar to views in Rel. DBs).

For blank nodes in CONSTRUCTs, we need Skolem terms as blank node identifiers!

```
answer1(Doc,N,def)  :-  triple_Q(Doc,dc:creator,N,def).
triple_Res(BLANK_b,rdf:type,foaf:Agent,res)  :-  answer1(Doc,N,def),
                                                  &sk[b,Doc,N](BLANK_b).
triple_Res(BLANK_b,foaf:name,N,res)  :-  answer1(Doc,N,def),
                                          &sk[b,Doc,N](BLANK_b).
triple_Res(Doc,foaf:maker,BLANK_b,res)  :-  answer1(Doc,N,def),
                                             &sk[b,Doc,N](BLANK_b).
```

# Summary:

- SPARQL to Datalog seems easy
- Actual implementation raises some issues ... not SOOO easy.
- We have implemented a recursive translation from arbitrarily nested SPARQL queries to dlvhex
- We further are working towards a full implementation of SPARQL on dlvhex
- Why do we do that?
  - dlvhex is a qood platform for extensions (aggregates), additional built-in functions
  - CONSTRUCTs may be viewed as rules them selves, useful for defining implicit, interlinked metadata in RDF. ⇒ We can implement such an extension to RDF right away.
  - combination with RDFS inference rules
  - Recent results on dlv-db for RDF give us confidence that this is not only a “toy” implementation of SPARQL, but could in fact lead to a competitive RDF-Store
  - We are currently implementing these extensions to SPARQL!
- Ask me in the coffee break for a DEMO!

# Summary:

- SPARQL to Datalog seems easy
- Actual implementation raises some issues ... not SOOO easy.
- We have implemented a recursive translation from arbitrarily nested SPARQL queries to dlvhex
- We further are working towards a full implementation of SPARQL on dlvhex
- Why do we do that?
  - dlvhex is a good platform for extensions (aggregates), additional built-in functions
  - CONSTRUCTs may be viewed as rules them selves, useful for defining implicit, interlinked metadata in RDF. ⇒ We can implement such an extension to RDF right away.
  - combination with RDFS inference rules
  - Recent results on dlv-db for RDF give us confidence that this is not only a "toy" implementation of SPARQL, but could in fact lead to a competitive RDF-Store
  - We are currently implementing these extensions to SPARQL!
- Ask me in the coffee break for a DEMO!

# Summary:

- SPARQL to Datalog seems easy
- Actual implementation raises some issues ... not SOOO easy.
- We have implemented a recursive translation from arbitrarily nested SPARQL queries to dlvhex
- We further are working towards a full implementation of SPARQL on dlvhex
- Why do we do that?
  - dlvhex is a good platform for extensions (aggregates), additional built-in functions
  - CONSTRUCTs may be viewed as rules them selves, useful for defining implicit, interlinked metadata in RDF. ⇒ We can implement such an extension to RDF right away.
  - combination with RDFS inference rules
  - Recent results on dlv-db for RDF give us confidence that this is not only a "toy" implementation of SPARQL, but could in fact lead to a competitive RDF-Store
  - We are currently implementing these extensions to SPARQL!
- Ask me in the coffee break for a DEMO!

# Summary:

- SPARQL to Datalog seems easy
- Actual implementation raises some issues ... not SOOO easy.
- We have implemented a recursive translation from arbitrarily nested SPARQL queries to dlvhex
- We further are working towards a full implementation of SPARQL on dlvhex
- Why do we do that?
  - dlvhex is a good platform for extensions (aggregates), additional built-in functions
  - CONSTRUCTs may be viewed as rules them selves, useful for defining implicit, interlinked metadata in RDF. ⇒ We can implement such an extension to RDF right away.
  - combination with RDFS inference rules
  - Recent results on dlv-db for RDF give us confidence that this is not only a "toy" implementation of SPARQL, but could in fact lead to a competitive RDF-Store
  - We are currently implementing these extensions to SPARQL!
- Ask me in the coffee break for a DEMO!

# Summary:

- SPARQL to Datalog seems easy
- Actual implementation raises some issues ... not SOOO easy.
- We have implemented a recursive translation from arbitrarily nested SPARQL queries to dlvhex
- We further are working towards a full implementation of SPARQL on dlvhex
- Why do we do that?
  - dlvhex is a good platform for extensions (aggregates), additional built-in functions
  - CONSTRUCTs may be viewed as rules them selves, useful for defining implicit, interlinked metadata in RDF. ⇒ We can implement such an extension to RDF right away.
  - combination with RDFS inference rules
  - Recent results on dlv-db for RDF give us confidence that this is not only a "toy" implementation of SPARQL, but could in fact lead to a competitive RDF-Store
  - We are currently implementing these extensions to SPARQL!
- Ask me in the coffee break for a DEMO!

# Summary:

- SPARQL to Datalog seems easy
- Actual implementation raises some issues ... not SOOO easy.
- We have implemented a recursive translation from arbitrarily nested SPARQL queries to dlvhex
- We further are working towards a full implementation of SPARQL on dlvhex
- Why do we do that?
  - dlvhex is a qood platform for extensions (aggregates), additional built-in functions
  - CONSTRUCTs may be viewed as rules them selves, useful for defining implicit, interlinked metadata in RDF. $\Rightarrow$ We can implement such an extension to RDF right away.
  - combination with RDFS inference rules
  - Recent results on dlv-db for RDF give us confidence that this is not only a "toy" implementation of SPARQL, but could in fact lead to a competitive RDF-Store
  - We are currently implementing these extensions to SPARQL!
- Ask me in the coffee break for a DEMO!

# Summary:

- SPARQL to Datalog seems easy
- Actual implementation raises some issues ... not SOOO easy.
- We have implemented a recursive translation from arbitrarily nested SPARQL queries to dlvhex
- We further are working towards a full implementation of SPARQL on dlvhex
- Why do we do that?
  - dlvhex is a qood platform for extensions (aggregates), additional built-in functions
  - CONSTRUCTs may be viewed as rules them selves, useful for defining implicit, interlinked metadata in RDF. ⇒ We can implement such an extension to RDF right away.
  - combination with RDFS inference rules
  - Recent results on dlv-db for RDF give us confidence that this is not only a “toy” implementation of SPARQL, but could in fact lead to a competitive RDF-Store
  - We are currently implementing these extensions to SPARQL!
- Ask me in the coffee break for a DEMO!

# Summary:

- SPARQL to Datalog seems easy
- Actual implementation raises some issues ... not SOOO easy.
- We have implemented a recursive translation from arbitrarily nested SPARQL queries to dlvhex
- We further are working towards a full implementation of SPARQL on dlvhex
- Why do we do that?
  - dlvhex is a qood platform for extensions (aggregates), additional built-in functions
  - CONSTRUCTs may be viewed as rules them selves, useful for defining implicit, interlinked metadata in RDF. $\Rightarrow$ We can implement such an extension to RDF right away.
  - combination with RDFS inference rules
  - Recent results on dlv-db for RDF give us confidence that this is not only a "toy" implementation of SPARQL, but could in fact lead to a competitive RDF-Store
  - We are currently implementing these extensions to SPARQL!
- Ask me in the coffee break for a DEMO!

# Summary:

- SPARQL to Datalog seems easy
- Actual implementation raises some issues ... not SOOO easy.
- We have implemented a recursive translation from arbitrarily nested SPARQL queries to dlvhex
- We further are working towards a full implementation of SPARQL on dlvhex
- Why do we do that?
  - dlvhex is a qood platform for extensions (aggregates), additional built-in functions
  - CONSTRUCTs may be viewed as rules them selves, useful for defining implicit, interlinked metadata in RDF. ⇒ We can implement such an extension to RDF right away.
  - combination with RDFS inference rules
  - Recent results on dlv-db for RDF give us confidence that this is not only a "toy" implementation of SPARQL, but could in fact lead to a competitive RDF-Store
  - We are currently implementing these extensions to SPARQL!
- Ask me in the coffee break for a DEMO!

# Summary:

- SPARQL to Datalog seems easy
- Actual implementation raises some issues ... not SOOO easy.
- We have implemented a recursive translation from arbitrarily nested SPARQL queries to dlvhex
- We further are working towards a full implementation of SPARQL on dlvhex
- Why do we do that?
  - dlvhex is a qood platform for extensions (aggregates), additional built-in functions
  - CONSTRUCTs may be viewed as rules them selves, useful for defining implicit, interlinked metadata in RDF. ⇒ We can implement such an extension to RDF right away.
  - combination with RDFS inference rules
  - Recent results on dlv-db for RDF give us confidence that this is not only a "toy" implementation of SPARQL, but could in fact lead to a competitive RDF-Store
  - We are currently implementing these extensions to SPARQL!
- Ask me in the coffee break for a DEMO!