SPARQL-DL
Implementation Experience

Petr Křemen\(^1\) Evren Sirin\(^2\)

\(^1\)Czech Technical University in Prague (CZ)

\(^2\)Clark & Parsia (US)

April 2008
What is SPARQL-DL
  Different Perspectives
  SPARQL-DL constructs

SPARQL-DL Evaluation
  Preprocessing
  Evaluation Strategies
  Optimizations

Examples
SPARQL-DL vs. Conjunctive Queries

- query language for OWL-DL ontologies.
SPARQL-DL vs. Conjunctive Queries

- query language for OWL-DL ontologies.
- mixed ABox / TBox queries:

Conjunctive ABox Queries.
"Get all teachers and their students."

Employee

AssociateProfessor

AssistantProfessor

AssociationProfessor1

AssistantProfessor7

Course17

Course47

UndergraduateStudent4

UndergraduateStudent20

teachOf

takesCourse
SPARQL-DL vs. Conjunctive Queries

- query language for OWL-DL ontologies.
- mixed ABox / TBox queries:

Mixed TBox/ABox Queries.
"Get all teachers and their students."
"... together with the type of the teachers."
SPARQL-DL vs. SPARQL

- SPARQL-DL uses SPARQL syntax

Example (SPARQL-DL)

```
Type (?,?,?), SubClassOf (?,Employee), PropertyValue (?,teacherOf:a), PropertyValue (?,takesCourse:a).
```

Example (SPARQL)

```
SELECT ?t ?x ?y
WHERE {
?x rdf:type ?t .
?t rdfs:subClassOf :Employee.
}
```
SPARQL-DL vs. SPARQL

- SPARQL-DL uses SPARQL syntax
- SPARQL-DL provides OWL-DL semantics for SPARQL basic graph patterns:
SPARQL-DL vs. SPARQL

- SPARQL-DL uses SPARQL syntax
- SPARQL-DL provides OWL-DL semantics for SPARQL basic graph patterns:

**Example (SPARQL-DL)**

```
Type(?x, ?t), SubClassOf(?t, Employee), PropertyValue(?x, teacherOf, _ : a), PropertyValue(?y, takesCourse, _ : a).
```
SPARQL-DL vs. SPARQL

- SPARQL-DL uses SPARQL syntax
- SPARQL-DL provides OWL-DL semantics for SPARQL basic graph patterns:

**Example (SPARQL-DL)**

```
Type(?x, ?t), SubClassOf(?t, Employee), PropertyValue(?x, teacherOf, _ : a), PropertyValue(?y, takesCourse, _ : a).
```

**Example (SPARQL)**

```
SELECT ?t ?x ?y
WHERE {
  ?x rdf:type ?t .
  ?t rdfs:subClassOf :Employee.
}
```
SPARQL-DL Constructs

SPARQL-DL query is a conjunction of atoms:

\text{Type}(i, c), \text{PropertyValue}(i, p, j) \quad \text{– conjunctive query atoms allowing distinguished variables in } c/ p \text{ positions}
SPARQL-DL Constructs

SPARQL-DL query is a conjunction of atoms:
Type\((i, c)\), PropertyValue\((i, p, j)\) – conjunctive query atoms allowing distinguished variables in \(c/ p\) positions
SameAs\((i, j)\), DifferentFrom\((i, j)\) – OWL individual axiom patterns.
SPARQL-DL Constructs

SPARQL-DL query is a conjunction of atoms:

- **Type**($i, c$), **PropertyValue**($i, p, j$) – conjunctive query atoms allowing distinguished variables in $c/p$ positions
- **SameAs**($i, j$), **DifferentFrom**($i, j$) – OWL individual axiom patterns.
- **SubClassOf**($c, d$), **EquivalentClass**($c, d$), **DisjointWith**($c, d$) – OWL class axiom patterns
SPARQL-DL Constructs

SPARQL-DL query is a conjunction of atoms:

- **Type**\((i, c)\), **PropertyValue**\((i, p, j)\) – conjunctive query atoms allowing distinguished variables in \(c/ p\) positions
- **SameAs**\((i, j)\), **DifferentFrom**\((i, j)\) – OWL individual axiom patterns.
- **SubClassOf**\((c, d)\), **EquivalentClass**\((c, d)\), **DisjointWith**\((c, d)\) – OWL class axiom patterns
- **ComplementOf**\((c, d)\) – pattern for matching class complement (the only class description construct)
SPARQL-DL Constructs

SPARQL-DL query is a conjunction of atoms:

- **Type**(*i, c)*, **PropertyValue**(*i, p, j*) – conjunctive query atoms allowing distinguished variables in *c/ p* positions
- **SameAs**(*i, j*), **DifferentFrom**(*i, j*) – OWL individual axiom patterns.
- **SubClassOf**(*c, d*), **EquivalentClass**(*c, d*), **DisjointWith**(*c, d*) – OWL class axiom patterns
- **ComplementOf**(*c, d*) – pattern for matching class complement (the only class description construct)
- **SubPropertyOf**(*p, q*), **EquivalentProperty**(*p, q*), **InverseOf**(*p, q*)
SPARQL-DL Constructs

SPARQL-DL query is a conjunction of atoms:

Type\((i, c)\), PropertyValue\((i, p, j)\) – conjunctive query atoms allowing distinguished variables in \(c/p\) positions

SameAs\((i, j)\), DifferentFrom\((i, j)\) – OWL individual axiom patterns.

SubClassOf\((c, d)\), EquivalentClass\((c, d)\), DisjointWith\((c, d)\) – OWL class axiom patterns

ComplementOf\((c, d)\) – pattern for matching class complement (the only class description construct)

SubPropertyOf\((p, q)\), EquivalentProperty\((p, q)\), InverseOf\((p, q)\)

ObjectProperty\((p)\), DataProperty\((p)\), FunctionalProperty\((p)\)
SPARQL-DL Constructs

SPARQL-DL query is a conjunction of atoms:

- **Type**\((i, c)\), **PropertyValue**\((i, p, j)\) – conjunctive query atoms allowing distinguished variables in \(c/p\) positions
- **SameAs**\((i, j)\), **DifferentFrom**\((i, j)\) – OWL individual axiom patterns.
- **SubClassOf**\((c, d)\), **EquivalentClass**\((c, d)\), **DisjointWith**\((c, d)\) – OWL class axiom patterns
- **ComplementOf**\((c, d)\) – pattern for matching class complement (the only class description construct)
- **SubPropertyOf**\((p, q)\), **EquivalentProperty**\((p, q)\), **InverseOf**\((p, q)\)
- **ObjectProperty**\((p)\), **DataProperty**\((p)\), **FunctionalProperty**\((p)\)
- **InverseFunctional**\((p)\), **Symmetric**\((p)\), **Transitive**\((p)\) – OWL property axiom patterns
SPARQL-DL Constructs

SPARQL-DL query is a conjunction of atoms:

Type\((i, c)\), PropertyValue\((i, p, j)\)  – conjunctive query atoms allowing distinguished variables in \(c/p\) positions

SameAs\((i, j)\), DifferentFrom\((i, j)\)  – OWL individual axiom patterns.

SubClassOf\((c, d)\), EquivalentClass\((c, d)\), DisjointWith\((c, d)\)  – OWL class axiom patterns

ComplementOf\((c, d)\)  – pattern for matching class complement (the only class description construct)

SubPropertyOf\((p, q)\), EquivalentProperty\((p, q)\), InverseOf\((p, q)\)

ObjectProperty\((p)\), DataProperty\((p)\), FunctionalProperty\((p)\)

InverseFunctional\((p)\), Symmetric\((p)\), Transitive\((p)\)  – OWL property axiom patterns

Annotation\((i, p, j)\)  – ground atom for matching OWL annotations
SPARQL-DL Constructs

SPARQL-DL query is a conjunction of atoms:

- **Type** \((i, c)\), **PropertyValue** \((i, p, j)\) – conjunctive query atoms allowing distinguished variables in \(c/p\) positions
- **SameAs** \((i, j)\), **DifferentFrom** \((i, j)\) – OWL individual axiom patterns.
- **SubClassOf** \((c, d)\), **EquivalentClass** \((c, d)\), **DisjointWith** \((c, d)\) – OWL class axiom patterns
- **ComplementOf** \((c, d)\) – pattern for matching class complement (the only class description construct)
- **SubPropertyOf** \((p, q)\), **EquivalentProperty** \((p, q)\), **InverseOf** \((p, q)\)
- **ObjectProperty** \((p)\), **DataProperty** \((p)\), **FunctionalProperty** \((p)\)
- **InverseFunctional** \((p)\), **Symmetric** \((p)\), **Transitive** \((p)\) – OWL property axiom patterns
- **Annotation** \((i, p, j)\) – ground atom for matching OWL annotations

... non-monotonic extension – **DirectType** \((i, c)\), **DirectSubClassOf** \((c, d)\), **StrictSubClassOf** \((c, d)\), **DirectSubProperty** \((p, q)\), **StrictSubProperty** \((p, q)\).
Preprocessing

- getting rid of all SameAs atoms with undistinguished variables
Preprocessing

- getting rid of all `SameAs` atoms with undistinguished variables

Example

$q_1(\text{?}x) \rightarrow \text{SameAs(_: b, ?x), Type(_: b, Person)}$
turns
$q_2(\text{?}x) \rightarrow \text{Type(?x, Person)}$
Preprocessing

- getting rid of all `SameAs` atoms with undistinguished variables

Example

<table>
<thead>
<tr>
<th>$q_1(?x)$</th>
<th>$\rightarrow$</th>
<th><code>SameAs(_ : b, ?x), Type(_ : b, Person)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>turns</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$q_2(?x)$</th>
<th>$\rightarrow$</th>
<th><code>Type(?x, Person)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>turns</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- BUT, we cannot perform this simplification for distinguished variable in `SameAs` atoms, since there can be several individuals in KB that are stated to be same.
Preprocessing

- getting rid of all `SameAs` atoms with undistinguished variables

**Example**

\[ q_1(?x) \rightarrow \text{SameAs}(\_: b, ?x), \text{Type}(\_: b, \text{Person}) \]

\[ q_2(?x) \rightarrow \text{Type}(?x, \text{Person}) \]

- BUT, we cannot perform this simplification for distinguished variable in `SameAs` atoms, since there can be several individuals in KB that are stated to be same.

- removing trivially satisfied atoms is valuable w.r.t. the cost based reordering
Preprocessing

- getting rid of all \texttt{SameAs} atoms with undistinguished variables

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
Example \tabularnewline
\hline
$q_1(?x) \rightarrow \texttt{SameAs(_ : b, ?x), Type(_ : b, \texttt{Person})}$ \tabularnewline
turns \tabularnewline
$q_2(?x) \rightarrow \texttt{Type(?x, \texttt{Person})}$ \tabularnewline
\hline
\end{tabular}
\end{table}

- BUT, we cannot perform this simplification for distinguished variable in \texttt{SameAs} atoms, since there can be several individuals in KB that are stated to be same.

- removing trivially satisfied atoms is valuable w.r.t. the cost based reordering

- splitting the query into connected components in order to avoid computing cross-products of their results.
Preprocessing

- getting rid of all \texttt{SameAs} atoms with undistinguished variables

\textbf{Example}

\begin{verbatim}
q_1(?x) \rightarrow \texttt{SameAs(\_: b, ?x)}, \texttt{Type(\_: b, Person)}
\end{verbatim}

turns

\begin{verbatim}
q_2(?x) \rightarrow \texttt{Type(?x, Person)}
\end{verbatim}

- BUT, we cannot perform this simplification for distinguished variable in \texttt{SameAs} atoms, since there can be several individuals in KB that are stated to be same.

- removing trivially satisfied atoms is valuable w.r.t. the cost based reordering

- splitting the query into connected components in order to avoid computing cross-products of their results.

- queries without \texttt{DifferentFrom} atoms with undistinguished variables.
Separated vs. Mixed Evaluation

**separated** partitioning a SPARQL-DL query $Q$ into the ABox part $Q_c$ (**Type** and **PropertyValue**) and Schema part $Q_s$.
Separated vs. Mixed Evaluation

Separated partitioning a SPARQL-DL query $Q$ into the ABox part $Q_c$ (Type and PropertyValue) and Schema part $Q_s$

- augmenting $Q_s$ with $\text{SubClassOf}(\?x, \?x)$, resp. $\text{SubPropertyOf}(\?x, \?x)$ for all $\?x$ in $\text{Type}(\bullet, \?x)$, resp. $\text{PropertyValue}(\bullet, \?x, \bullet)$ atoms that do not appear in $Q_s$. 

better performance w.r.t. the query reordering.

only SPARQL-DL queries with distinguished variables
Separated vs. Mixed Evaluation

**Separated** partitioning a SPARQL-DL query $Q$ into the ABox part $Q_c$ (**Type** and **PropertyValue**) and Schema part $Q_s$

- augmenting $Q_s$ with $\text{SubClassOf}(\text{?x}, \text{?x})$, resp. $\text{SubPropertyOf}(\text{?x}, \text{?x})$ for all $\text{?x}$ in $\text{Type}(\text{?}, \text{?x})$, resp. $\text{PropertyValue}(\text{?}, \text{?x}, \text{?})$ atoms that do not appear in $Q_s$.
- evaluate first $Q_s$ using the SPARQL-DL engine and for each binding found evaluate $Q_c$ part using the existing ABox query engine,

Better performance w.r.t. the query reordering.
Separated vs. Mixed Evaluation

partitioning a SPARQL-DL query $Q$ into the ABox part $Q_c$ (Type and PropertyValue) and Schema part $Q_s$

- augmenting $Q_s$ with $\text{SubClassOf}(?x, ?x)$, resp. $\text{SubPropertyOf}(?x, ?x)$ for all $?x$ in $\text{Type}(\bullet, ?x)$, resp. $\text{PropertyValue}(\bullet, ?x, \bullet)$ atoms that do not appear in $Q_s$.
- evaluate first $Q_s$ using the SPARQL-DL engine and for each binding found evaluate $Q_c$ part using the existing ABox query engine

mixed using SPARQL-DL evaluation for all atoms
Separated vs. Mixed Evaluation

**Separated** partitioning a SPARQL-DL query $Q$ into the ABox part $Q_c$ (*Type* and *PropertyValue*) and Schema part $Q_s$

- augmenting $Q_s$ with $\text{SubClassOf}(?x, ?x)$, resp. $\text{SubPropertyOf}(?x, ?x)$ for all $?x$ in $\text{Type}(\bullet, ?x)$, resp. $\text{PropertyValue}(\bullet, ?x, \bullet)$ atoms that do not appear in $Q_s$.

- evaluate first $Q_s$ using the SPARQL-DL engine and for each binding found evaluate $Q_c$ part using the existing ABox query engine

**Mixed** using SPARQL-DL evaluation for all atoms

😊 better performance w.r.t. the query reordering.
Separated vs. Mixed Evaluation

**separated** partitioning a SPARQL-DL query $Q$ into the ABox part $Q_c$ (**Type** and **PropertyValue**) and Schema part $Q_s$

- augmenting $Q_s$ with $\text{SubClassOf}(?x, ?x)$, resp. $\text{SubPropertyOf}(?x, ?x)$ for all $?x$ in $\text{Type}(\bullet, ?x)$, resp. $\text{PropertyValue}(\bullet, ?x, \bullet)$ atoms that do not appear in $Q_s$.
- evaluate first $Q_s$ using the SPARQL-DL engine and for each binding found evaluate $Q_c$ part using the existing ABox query engine

**mixed** using SPARQL-DL evaluation for all atoms

- better performance w.r.t. the query reordering.
- only SPARQL-DL queries with distinguished variables
computes cheapest atom ordering in advance. We choose ordering $p^* = \arg \min cost(p, 0)$, where:

\[
\begin{align*}
    cost(p, length(p)) &= 1 \\
    cost(p, i) &= cost_{KB}(p[i]) + B(p[i]) \times cost(p, i + 1)
\end{align*}
\]
computes cheapest atom ordering in advance. We choose ordering \( p^* = \arg \min cost(p, 0) \), where:

\[
\begin{align*}
    cost(p, length(p)) & = 1 \\
    cost(p, i) & = cost_{KB}(p[i]) + B(p[i]) \times cost(p, i + 1)
\end{align*}
\]

\( cost_{KB} \) estimates cost for the dominant KB operation required to evaluate the atom: \textit{noSat}, \textit{oneSat}, \textit{classify}, \textit{realize}. 

\( B \) estimates number of branches generated by the atom using various KB characteristics, for example:

\[
\text{cost}_{KB}(\text{SubclassOf}(?x, \text{Person})) = \text{classify}(\text{SubclassOf}(?x, \text{Person})) = \#\text{toldSubclasses}(\text{Person})
\]
computes cheapest atom ordering in advance. We choose ordering \( p^* = \arg \min \text{cost}(p, 0) \), where:

\[
\text{cost}(p, \text{length}(p)) = 1
\]
\[
\text{cost}(p, i) = \text{cost}_{KB}(p[i]) + B(p[i]) \times \text{cost}(p, i + 1)
\]

\( \text{cost}_{KB} \) estimates cost for the dominant KB operation required to evaluate the atom: \textit{noSat}, \textit{oneSat}, \textit{classify}, \textit{realize}.

\( B \) estimates number of branches generated by the atom using various KB characteristics, for example:
computes cheapest atom ordering in advance. We choose ordering \( p^* = \arg \min \text{cost}(p, 0) \), where:

\[
\begin{align*}
\text{cost}(p, \text{length}(p)) &= 1 \\
\text{cost}(p, i) &= \text{cost}_{\text{KB}}(p[i]) + B(p[i]) \times \text{cost}(p, i + 1)
\end{align*}
\]

\( \text{cost}_{\text{KB}} \) ... estimates cost for the dominant KB operation required to evaluate the atom: \text{noSat}, \text{oneSat}, \text{classify}, \text{realize}.

\( B \) ... estimates number of branches generated by the atom using various KB characteristics, for example:

**Example**

\[
\begin{align*}
\text{cost}_{\text{KB}}(\text{SubclassOf}(?x, \text{Person})) &= \text{classify} \\
B(\text{SubclassOf}(?x, \text{Person})) &= \#\text{toldSubclasses}(\text{Person})
\end{align*}
\]
Static Query Reordering (2)

😊 for short queries is fast and precise enough,
Static Query Reordering (2)

- for short queries is fast and precise enough,
- as it needs to generate and evaluate all query atom orderings, and thus all permutations, it is useless for queries longer than as few as 10 atoms,
Static Query Reordering (2)

- for short queries is fast and precise enough,
- as it needs to generate and evaluate all query atom orderings, and thus all permutations, it is useless for queries longer than as few as 10 atoms,
- cost evaluation of each query ordering is linear in the query length, but its quality decreases with the number of distinguished variables.
Down-monotonic Variables (mixed queries)

TBox

[Diagram showing a hierarchy of classes: Person, Student, Employee, UndergraduateStudent, ResearchAssistant]

Query

\[ \ldots, \ SubClassOf(?x, \ Person), \ \ldots, \ Type(\bullet, ?x), \ \ldots \]
Down-monotonic Variables (mixed queries)

TBox

\[ \text{\( n > 0 \) results} \]

\[ ?x \]

\[ \text{Person} \]

\[ \text{Student} \]

\[ \text{Employee} \]

\[ \text{UndergraduateStudent} \]

\[ \text{ResearchAssistant} \]

Query

\[ \ldots, \ \text{SubClassOf}(?x, \text{Person}), \ \ldots, \ \text{Type}(\bullet, ?x), \ \ldots \]
Down-monotonic Variables (mixed queries)

TBox

0 results

Query

... , SubClassOf(\(?x\), Person), ... , Type(\(\bullet\), ?x), ...

\(\triangleleft\)
Down-monotonic Variables (mixed queries)

TBox

Query

\[ \ldots, \text{SubClassOf}(?x, \text{Person}), \ldots, \text{Type}(\bullet, ?x), \ldots \]
during query execution *down-monotonic variable* is a class/property variable \(?x\) occurring in a later \(\text{Type}(\bullet, ?x)\), or \(\text{PropertyValue}(\bullet, ?x, \bullet)\) atom.
during query execution *down-monotonic variable* is a class/property variable $?x$ occurring in a later Type(●, $?x$), or PropertyValue(●, $?x$, ●) atom.

if subsequent execution finds no results for class $C$ (bound for $?x$) we can safely avoid exploring its subs.
during query execution *down-monotonic variable* is a class/property variable \(?x\) occurring in a later \(\text{Type}(\bullet, ?x)\), or \(\text{PropertyValue}(\bullet, ?x, \bullet)\) atom.

- if subsequent execution finds no results for class \(C\) (bound for \(?x\)) we can safely avoid exploring its subs.

- reverse implication does not hold: Finding a binding \(C\) for (down-monotonic) \(?x\) we can not take all subs of \(C\) as valid bindings, for instance:

\[
\text{SubClassOf}(?x, C), \text{Type}(i, ?x), \text{ComplementOf}(?x, \text{not}(C))
\]
during query execution *down-monotonic variable* is a class/property variable $?x$ occurring in a later $\text{Type}(\bullet, ?x)$, or $\text{PropertyValue}(\bullet, ?x, \bullet)$ atom.

if subsequent execution finds no results for class $C$ (bound for $?x$) we can safely avoid exploring its subs.

reverse implication does not hold: Finding a binding $C$ for (down-monotonic) $?x$ we can not take all subs of $C$ as valid bindings, for instance:

$$\text{SubClassOf}(?x, C), \text{Type}(i, ?x), \text{ComplementOf}(?x, not(C))$$

useful for ontologies with rich taxonomies.
### Example (Q1 – Variables in property position)

Find all the graduate students that are related to a course and find what kind of relationship (e.g. \textit{takesCourse}):

\[
\text{Type}(?x, \text{GraduateStudent}), \text{PropertyValue}(?x, ?y, ?z), \text{Type}(?z, \text{Course})
\]

### Example (Q2 – Mixed ABox/TBox query)

Find all the students who are also employees and find what kind of employee (e.g. \textit{ResearchAssistant}):

\[
\text{Type}(?x, \text{Student}), \text{Type}(?x, ?C), \text{SubClassOf}(?C, \text{Employee})
\]

### Example (Q3 – Mixed ABox/RBox query)

Find all the members of \textit{Dept0} and what kind of membership (e.g. \textit{worksFor, headOf}):

\[
\text{Type}(?x, \text{Person}), \text{PropertyValue}(?x, ?y, \text{Dept0}), \text{SubPropertyOf}(?y, \text{memberOf})
\]
Experiments (results for LUBM(1))
What Has Been Done - Summary

SPARQL-DL implementation *without bnodes in DifferentFrom atoms to appear in the next Pellet release*

- simple preprocessing – getting rid of *SameAs* atoms with bnodes
What Has Been Done - Summary

SPARQL-DL implementation *without bnodes in `DifferentFrom` atoms* to appear in the next Pellet release

- simple preprocessing – getting rid of `SameAs` atoms with bnodes
- two evaluation strategies – using an existing CAQ engine / mixed evaluation
SPARQL-DL implementation *without bnodes in DifferentFrom atoms* to *appear in the next Pellet release*

- simple preprocessing – getting rid of *SameAs* atoms with bnodes
- two evaluation strategies – using an existing CAQ engine / mixed evaluation
- optimizations – static query reordering, down-monotonic variables
What to Do Next?

- evaluation and optimization of bnodes using the mixed engine
What to Do Next?

- evaluation and optimization of bnodes using the mixed engine
- moving towards OWL 1.1
What to Do Next?

- evaluation and optimization of bnodes using the mixed engine
- moving towards OWL 1.1
- different cost functions
What to Do Next?

- evaluation and optimization of bnodes using the mixed engine
- moving towards OWL 1.1
- different cost functions
- dynamic query reordering
What to Do Next?

- evaluation and optimization of bnodes using the mixed engine
- moving towards OWL 1.1
- different cost functions
- dynamic query reordering
- more SPARQL stuff – optimized implementation of SPARQL algebra, like UNION, OPTIONAL, FILTER, etc...
What to Do Next?

- evaluation and optimization of bnodes using the mixed engine
- moving towards OWL 1.1
- different cost functions
- dynamic query reordering
- more SPARQL stuff – optimized implementation of SPARQL algebra, like UNION, OPTIONAL, FILTER, etc.
- ... and much more