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

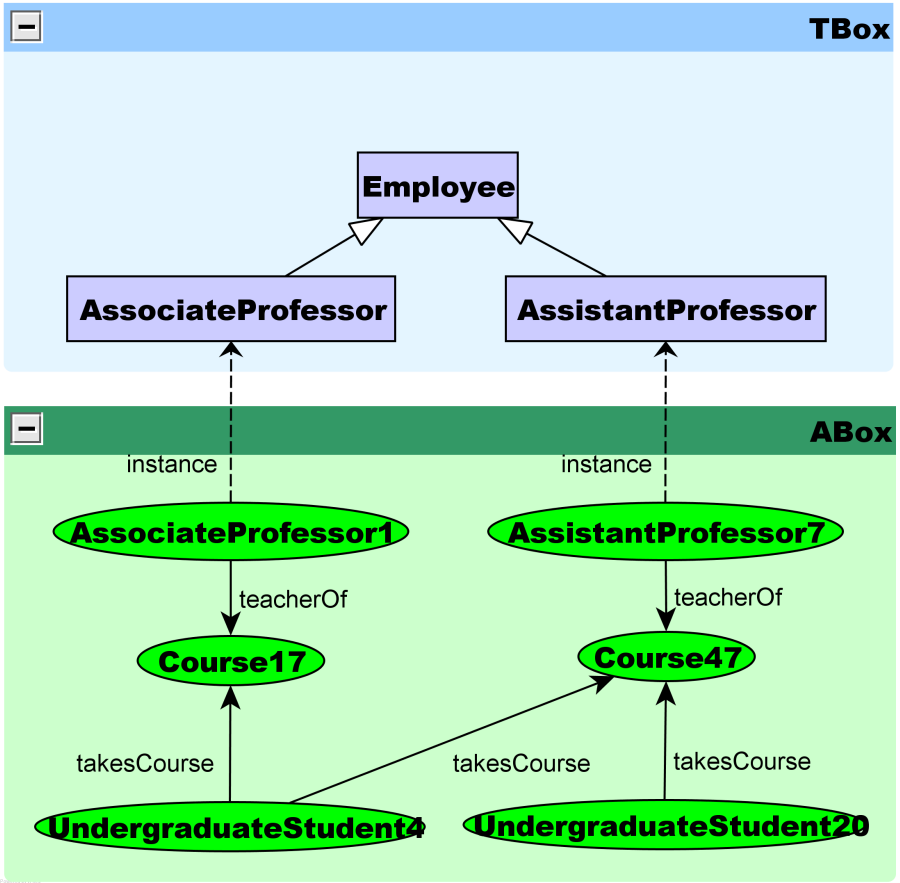
Examples

SPARQL-DL vs. Conjunctive Queries

- query language for OWL-DL ontologies.

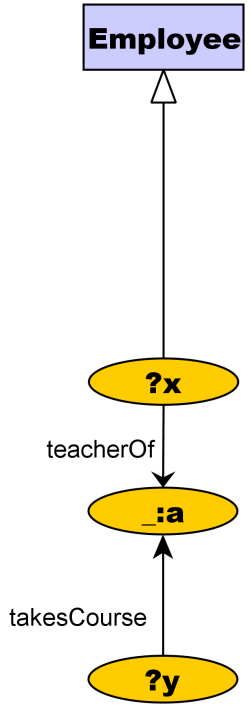
SPARQL-DL vs. Conjunctive Queries

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



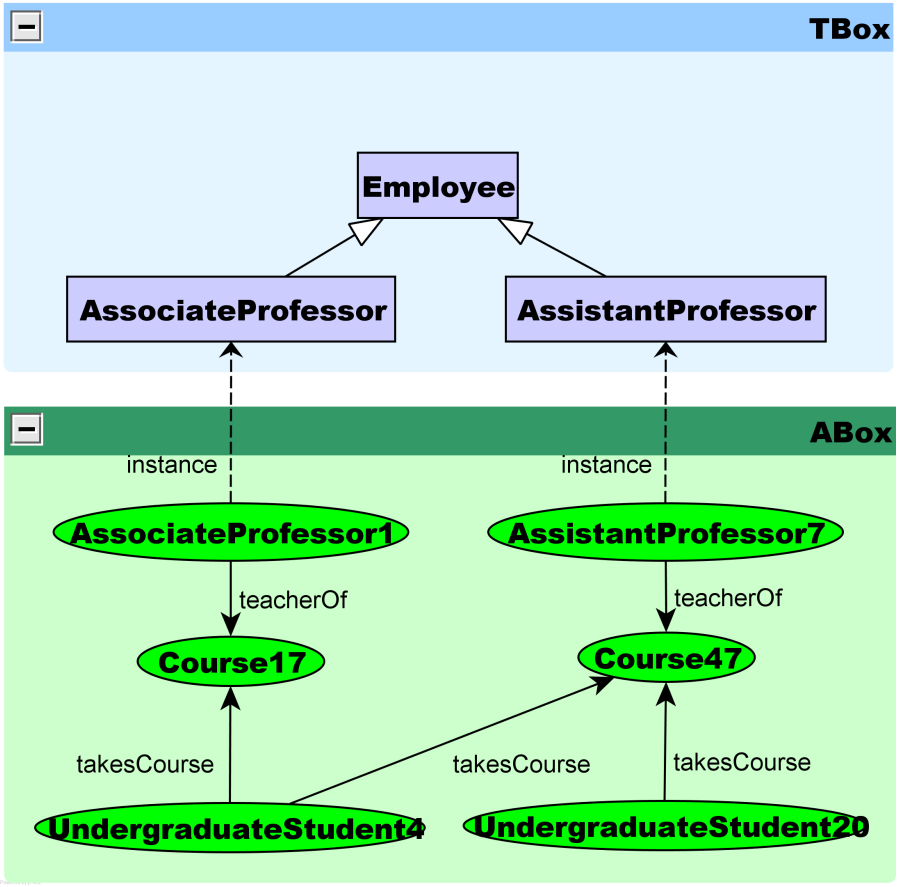
Conjunctive ABox Queries.

"Get all teachers and their students."



SPARQL-DL vs. Conjunctive Queries

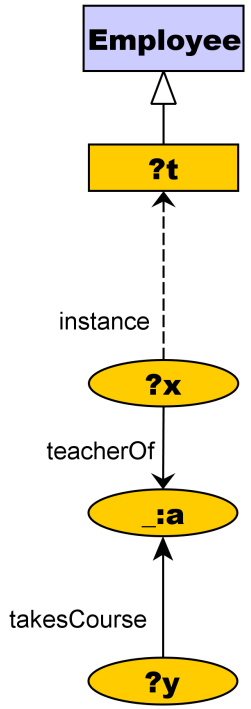
- query language for OWL-DL ontologies.
- mixed ABox / TBox queries :



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

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)

$\mathrm{Type}(?x, ?t), \mathrm{SubClassOf}(?t, \mathit{Employee}),$
$\mathrm{PropertyValue}(?x, \mathit{teacherOf}, \_ : a), \mathrm{PropertyValue}(?y, \mathit{takesCourse}, \_ : a).$

SPARQL-DL vs. SPARQL

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

Example (SPARQL-DL)

$\mathrm{Type}(?x, ?t), \mathrm{SubClassOf}(?t, \mathit{Employee}),$
$\mathrm{PropertyValue}(?x, \mathit{teacherOf}, \_ : a), \mathrm{PropertyValue}(?y, \mathit{takesCourse}, \_ : a).$

Example (SPARQL)

```
SELECT   ?t ?x ?y
WHERE    {
         ?x rdf:type ?t .
         ?t rdfs:subClassOf :Employee.
         ?x :teacherOf _:a .
         ?y :takesCourse _:a .
}
```

SPARQL-DL Constructs

SPARQL-DL query is a conjunction of atoms:

$\text{Type}(i, c), \text{PropertyValue}(i, p, j)$ – conjunctive query atoms allowing distinguished variables in c/ p 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), StrictSubPropertyOf(p, q).

Preprocessing

- getting rid of all SameAs atoms with undistinguished variables

Preprocessing

- getting rid of all SameAs atoms with undistinguished variables

Example

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

Preprocessing

- getting rid of all SameAs atoms with undistinguished variables

Example

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

- 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.

Preprocessing

- getting rid of all SameAs atoms with undistinguished variables

Example

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

- 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 SameAs atoms with undistinguished variables

Example

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

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

Preprocessing

- getting rid of all SameAs atoms with undistinguished variables

Example

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

- 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
- splitting the query into connected components in order to avoid computing cross-products of their results.
- queries without 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 SubClassOf$(?x, ?x)$, resp. SubPropertyOf$(?x, ?x)$ for all $?x$ in Type$(\bullet, ?x)$, resp. PropertyValue$(\bullet, ?x, \bullet)$ atoms that do not appear in 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

- augumenting Q_s with SubClassOf$(?x, ?x)$, resp. SubPropertyOf$(?x, ?x)$ for all $?x$ in Type$(\bullet, ?x)$, resp. 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

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

- augumenting Q_s with SubClassOf$(?x, ?x)$, resp. SubPropertyOf$(?x, ?x)$ for all $?x$ in Type$(\bullet, ?x)$, resp. 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

- augumenting Q_s with SubClassOf$(?x, ?x)$, resp. SubPropertyOf$(?x, ?x)$ for all $?x$ in Type$(\bullet, ?x)$, resp. 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

- augumenting Q_s with SubClassOf$(?x, ?x)$, resp. SubPropertyOf$(?x, ?x)$ for all $?x$ in Type$(\bullet, ?x)$, resp. 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

Static Query Reordering

- **computes cheapest atom ordering in advance**. We choose ordering $p^* = \arg\min cost(p, 0)$, where :

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

Static Query Reordering

- **computes cheapest atom ordering in advance**. We choose ordering $p^* = \arg\min cost(p, 0)$, where :

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

$cost_{KB}$ . . . estimates cost for the dominant KB operation required to evaluate the atom: *noSat*, *oneSat*, *classify*, *realize*.

Static Query Reordering

- **computes cheapest atom ordering in advance**. We choose ordering $p^* = \arg\min cost(p, 0)$, where :

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

$cost_{KB}$ . . . estimates cost for the dominant KB operation required to evaluate the atom: *noSat*, *oneSat*, *classify*, *realize*.

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

Static Query Reordering

- **computes cheapest atom ordering in advance**. We choose ordering $p^* = \arg\min cost(p, 0)$, where :

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

$cost_{KB}$ . . . estimates cost for the dominant KB operation required to evaluate the atom: *noSat*, *oneSat*, *classify*, *realize*.

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

Example

$cost_{KB}(\text{SubclassOf}(?x, Person)) = classify$
$B(\text{SubclassOf}(?x, Person)) = \#toldSubclasses(Person)$

Static Query Reordering (2)

- ☺ for short queries is fast and precise enough,

Static Query Reordering (2)

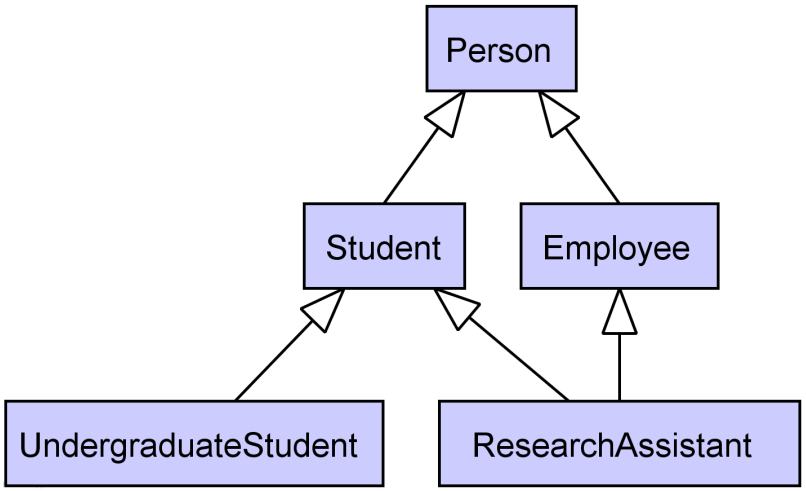
- ☺ for short queries is fast and precise enough,
- ☹ as it needs to generate and evaluate all query atom orderings, and thus all permutations, it is useless for queries longer than as few as 10 atoms,

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



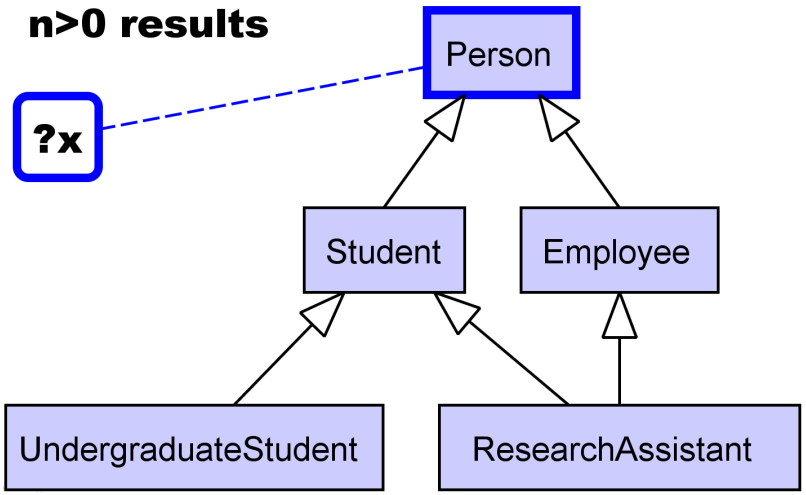
Query

$$\ldots, \quad \mathit{SubClassOf}(?x, \mathit{Person}), \quad \ldots, \quad \mathit{Type}(\bullet, ?x), \quad \ldots$$

▲

Down-monotonic Variables (mixed queries)

TBox



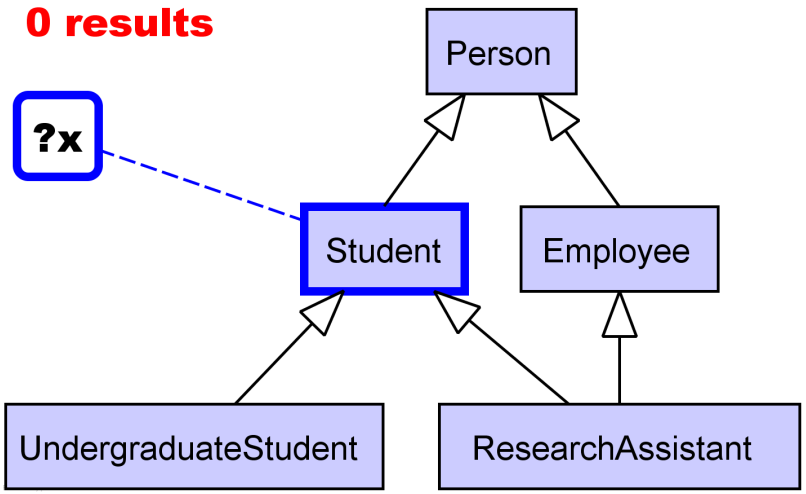
Query

$$\ldots, \quad \mathit{SubClassOf}(?x, \mathit{Person}), \quad \ldots, \quad \mathit{Type}(\bullet, ?x), \quad \ldots$$

▲

Down-monotonic Variables (mixed queries)

TBox



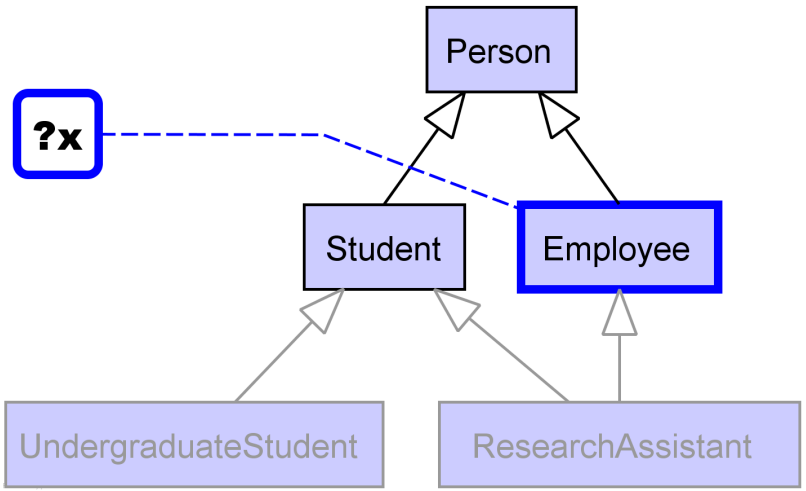
Query

$$\ldots, \quad \mathit{SubClassOf}(?x, \mathit{Person}), \quad \ldots, \quad \mathit{Type}(\bullet, ?x), \quad \ldots$$

▲

Down-monotonic Variables (mixed queries)

TBox



Query

$$\ldots, \quad \mathit{SubClassOf}(?x, \mathit{Person}), \quad \ldots, \quad \mathit{Type}(\bullet, ?x), \quad \ldots$$

▲

Down-monotonic Variables (2)

- during query execution *down-monotonic variable* is a class/property variable $?x$ occuring in a later $\mathrm{Type}(\bullet, ?x)$, or $\mathrm{PropertyValue}(\bullet, ?x, \bullet)$ atom.

Down-monotonic Variables (2)

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

Down-monotonic Variables (2)

- during query execution *down-monotonic variable* is a class/property variable $?x$ occuring in a later $\mathrm{Type}(\bullet, ?x)$, or $\mathrm{PropertyValue}(\bullet, ?x, \bullet)$ 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 :

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

Down-monotonic Variables (2)

- during query execution *down-monotonic variable* is a class/property variable $?x$ occuring in a later $\mathrm{Type}(\bullet, ?x)$, or $\mathrm{PropertyValue}(\bullet, ?x, \bullet)$ 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 :

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

☺ useful for ontologies with rich taxonomies.

Benchmarking with LUBM

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. `takesCourse`):

$$\text{Type}(?x, \texttt{GraduateStudent}), \text{PropertyValue}(?x, ?y, ?z), \text{Type}(?z, \texttt{Course})$$

Example (Q2 – Mixed ABox/TBox query)

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

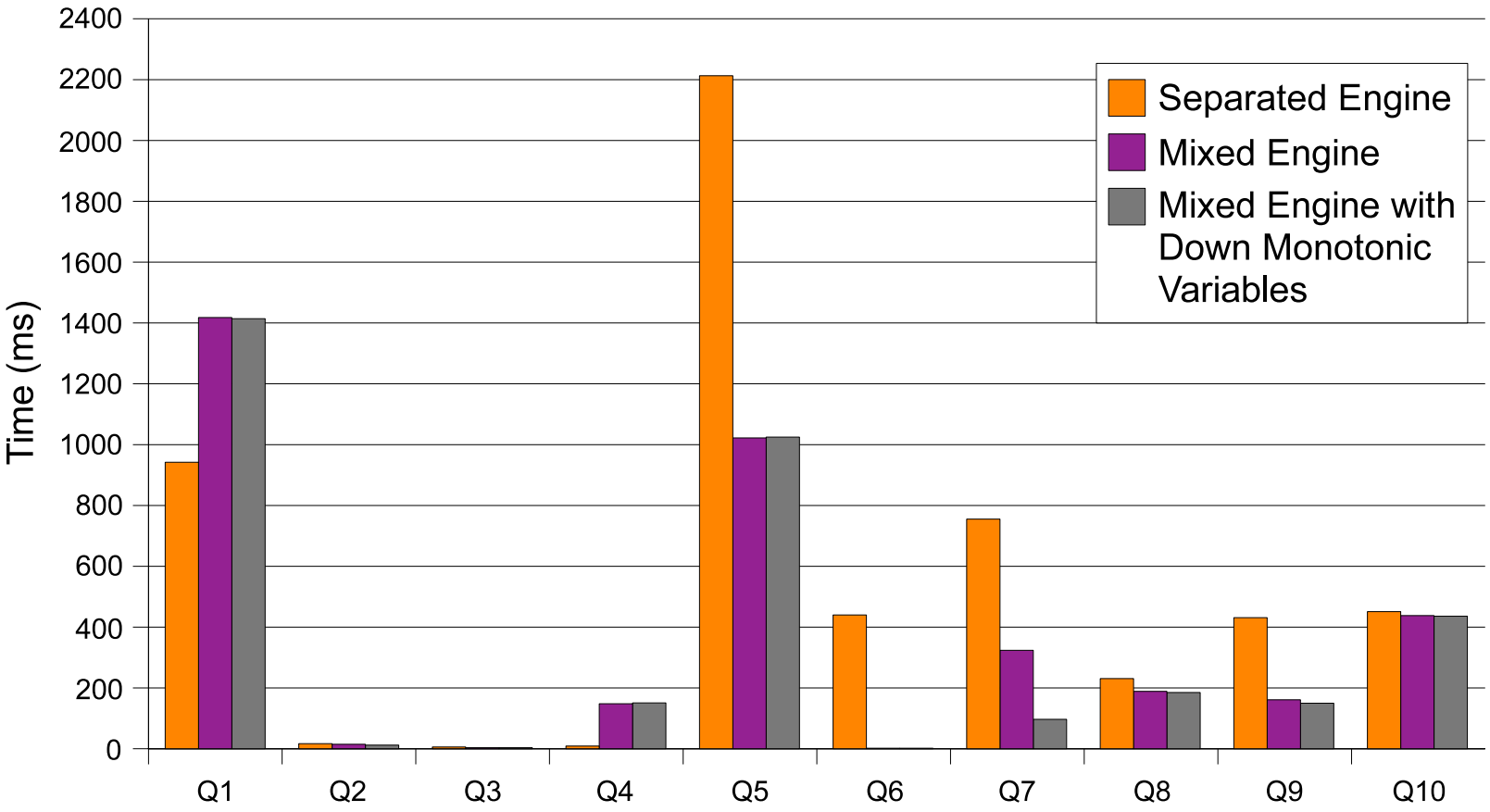
$$\text{Type}(?x, \texttt{Student}), \text{Type}(?x, ?C), \text{SubClassOf}(?C, \texttt{Employee})$$

Example (Q3 – Mixed ABox/RBox query)

Find all the members of `Dept0` and what kind of membership (e.g. `worksFor`, `headOf`):

$$\text{Type}(?x, \texttt{Person}), \text{PropertyValue}(?x, ?y, \texttt{Dept0}), \text{SubPropertyOf}(?y, \texttt{memberOf})$$

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

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
- 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 . . .
- . . . an much more