# SPARQL-DL

Petr Křemen

Czech Technical University in Prague (CZ)

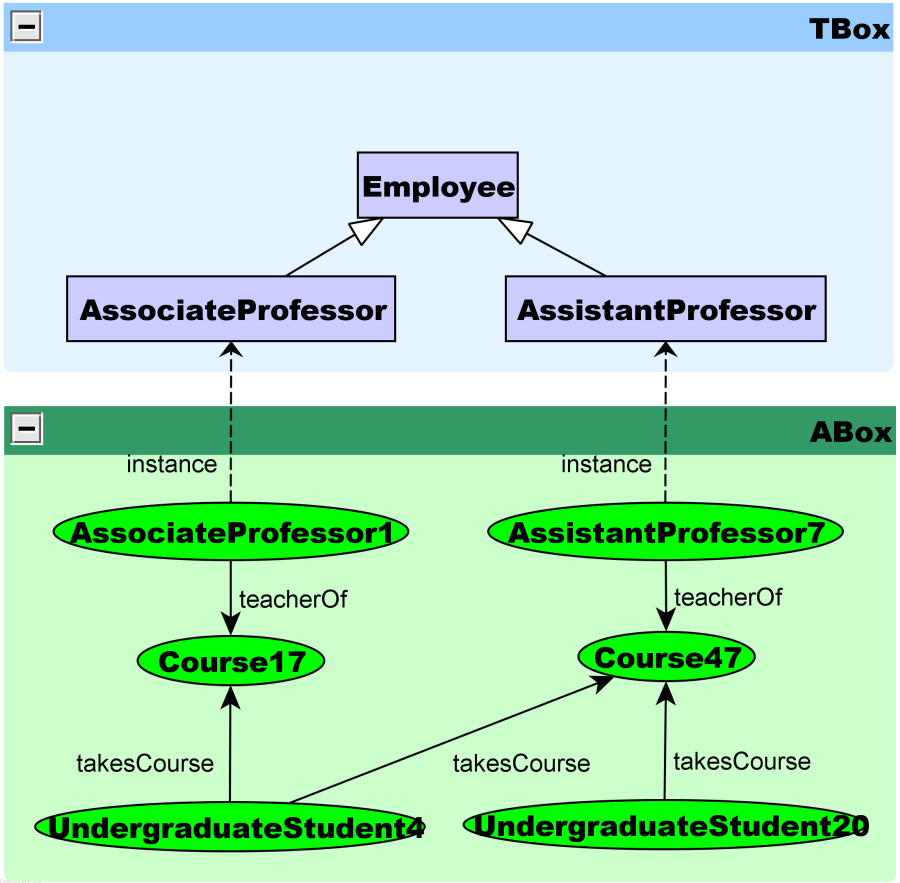
April 2008

# SPARQL-DL vs. Conjunctive Queries

- query language for OWL-DL ontologies.

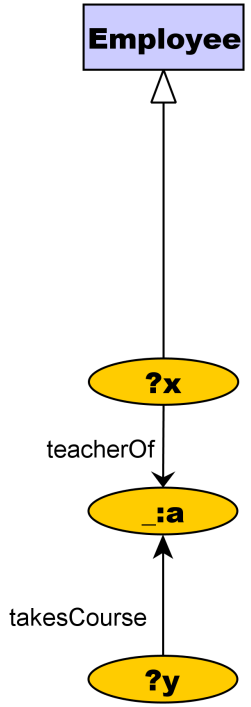
# SPARQL-DL vs. Conjunctive Queries

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



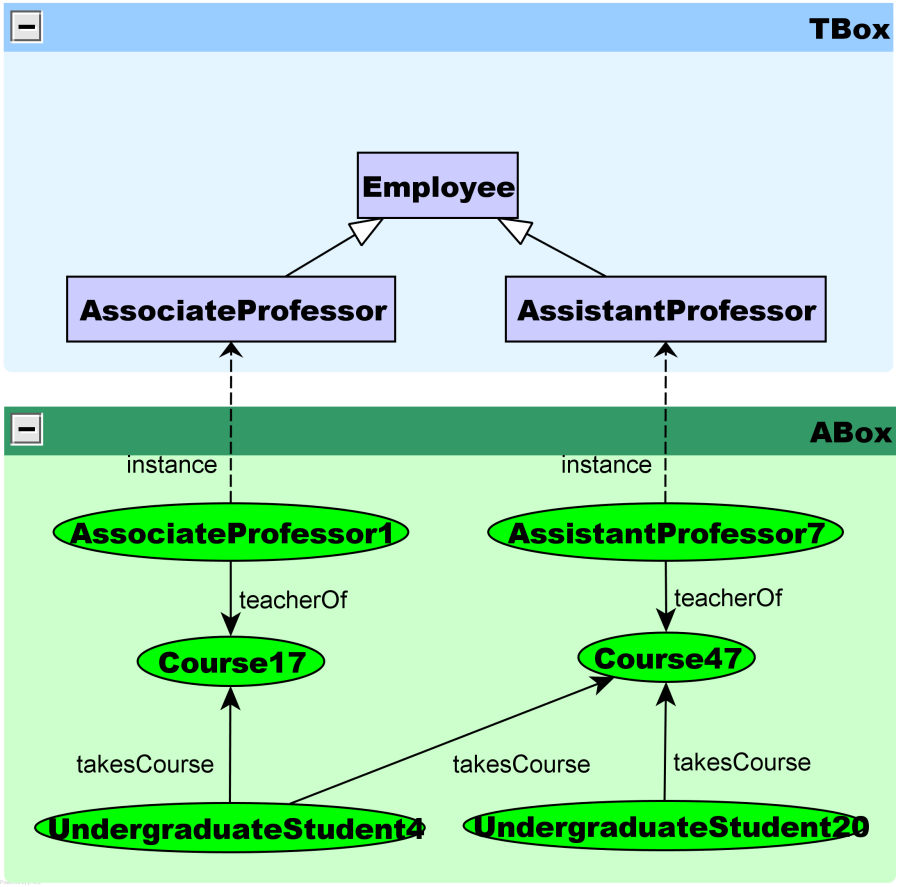

**Conjunctive ABox Queries.**

"Get all teachers and their students."



# SPARQL-DL vs. Conjunctive Queries

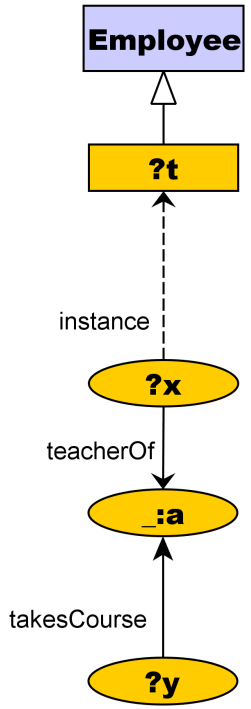
- query language for OWL-DL ontologies.
- mixed ABox / TBox queries :




**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($p$, $q$).

# SoA Pellet Engine Structure

- only conjunctive ABox queries (CAQ) supported :

# SoA Pellet Engine Structure

- only conjunctive ABox queries (CAQ) supported :

  NoDistVarQueryExec for q. without distinguished variables (DV) and at least one constant. The query is rolled-up to the constant and an instance check is performed.

# SoA Pellet Engine Structure

- only conjunctive ABox queries (CAQ) supported :

  NoDistVarQueryExec for q. without distinguished variables (DV) and at least one constant. The query is rolled-up to the constant and an instance check is performed.

  DistVarQueryExec for q. with only DV. Best atom (*Type*, *PropertyValue*) ordering is computed and atoms are evaluated in this ordering one by one.

# SoA Pellet Engine Structure

- only conjunctive ABox queries (CAQ) supported :

  NoDistVarQueryExec for q. without distinguished variables (DV) and at least one constant. The query is rolled-up to the constant and an instance check is performed.

  DistVarQueryExec for q. with only DV. Best atom (*Type*, *PropertyValue*) ordering is computed and atoms are evaluated in this ordering one by one.

  OptimizedQueryExec for q. with at least two DV. Query is rolled-up to each dist. variable. All possible bindings are explored, partial bindings being checked immediately.

# SoA Pellet Engine Structure

- only conjunctive ABox queries (CAQ) supported :

  NoDistVarQueryExec for q. without distinguished variables (DV) and at least one constant. The query is rolled-up to the constant and an instance check is performed.

  DistVarQueryExec for q. with only DV. Best atom (*Type*, *PropertyValue*) ordering is computed and atoms are evaluated in this ordering one by one.

  OptimizedQueryExec for q. with at least two DV. Query is rolled-up to each dist. variable. All possible bindings are explored, partial bindings being checked immediately.

  SimpleQueryExec for q. with at least one DV. Query is rolled-up to each dist. variable and all possible bindings are checked.

# SoA Pellet Engine Structure

- only conjunctive ABox queries (CAQ) supported :

  NoDistVarQueryExec for q. without distinguished variables (DV) and at least one constant. The query is rolled-up to the constant and an instance check is performed.

  DistVarQueryExec for q. with only DV. Best atom (*Type*, *PropertyValue*) ordering is computed and atoms are evaluated in this ordering one by one.

  OptimizedQueryExec for q. with at least two DV. Query is rolled-up to each dist. variable. All possible bindings are explored, partial bindings being checked immediately.

  SimpleQueryExec for q. with at least one DV. Query is rolled-up to each dist. variable and all possible bindings are checked.

# 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.

# Dynamic Query Reordering

- **chooses the cheapest atom in each evaluation step**. At present, *greedy-like* strategy is implemented. In each step cheapest atom $a^* = \arg\min cost(a)$ is chosen according to the following heuristic:

$$cost(a) = cost_{KB}(a) + B(a)\alpha,$$

# Dynamic Query Reordering

- **chooses the cheapest atom in each evaluation step**. At present, *greedy-like* strategy is implemented. In each step cheapest atom $a^* = \arg\min cost(a)$ is chosen according to the following heuristic:

$$cost(a) = cost_{KB}(a) + B(a)\alpha,$$

- $\alpha$ is a parameter – tested with values 0, *oneSat*.

# Dynamic Query Reordering

- **chooses the cheapest atom in each evaluation step**. At present, *greedy-like* strategy is implemented. In each step cheapest atom $a^* = \arg\min cost(a)$ is chosen according to the following heuristic:

$$cost(a) = cost_{KB}(a) + B(a)\alpha,$$

- $\alpha$ is a parameter – tested with values 0, *oneSat*.
- ☺ empirically more scalable than static reordering and comparable with static reordering on short queries.

# Dynamic Query Reordering

- **chooses the cheapest atom in each evaluation step**. At present, *greedy-like* strategy is implemented. In each step cheapest atom $a^* = \arg\min cost(a)$ is chosen according to the following heuristic:

$$cost(a) = cost_{KB}(a) + B(a)\alpha,$$

- $\alpha$ is a parameter – tested with values 0, *oneSat*.
- ☺ empirically more scalable than static reordering and comparable with static reordering on short queries.
- ☹ as any greedy search, it might suggest *really* inefficient strategies.

# Dynamic Query Reordering

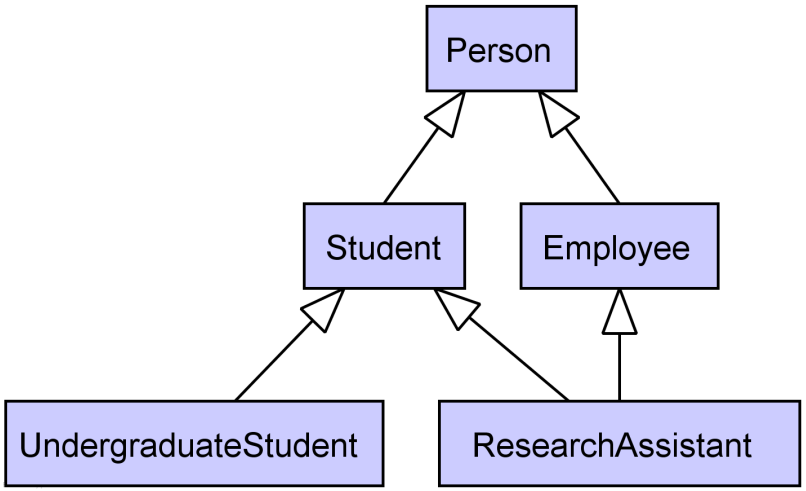
- **chooses the cheapest atom in each evaluation step**. At present, *greedy-like* strategy is implemented. In each step cheapest atom $a^* = \arg\min cost(a)$ is chosen according to the following heuristic:

$$cost(a) = cost_{KB}(a) + B(a)\alpha,$$

- $\alpha$ is a parameter – tested with values 0, *oneSat*.
- ☺ empirically more scalable than static reordering and comparable with static reordering on short queries.
- ☹ as any greedy search, it might suggest *really* inefficient strategies.
- more sofisticated heuristic functions for dynamic reordering will be helpful . . ..

# Down-monotonic Variables (mixed queries)

TBox



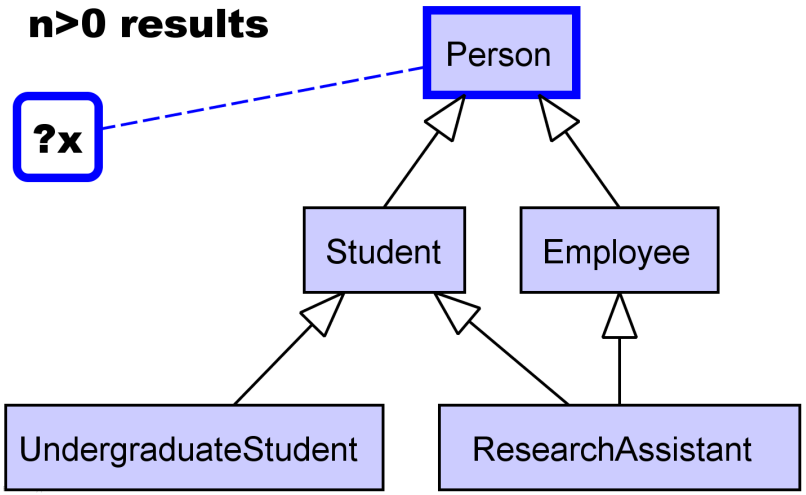

Query

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

▲

# Down-monotonic Variables (mixed queries)

TBox



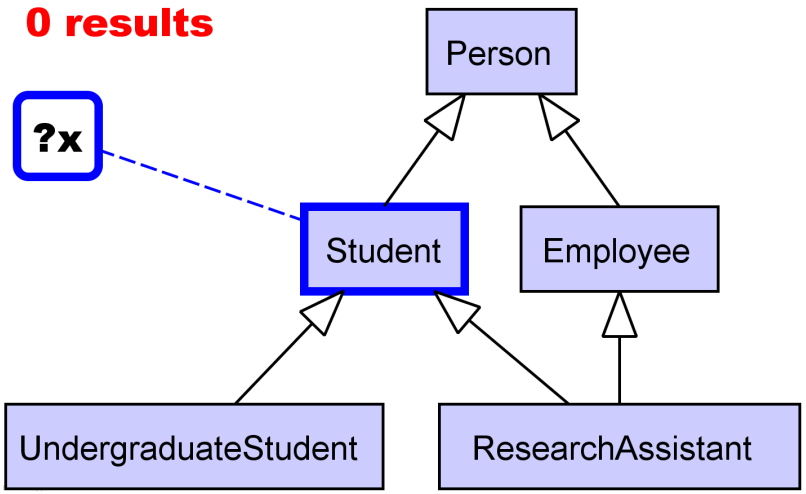

Query

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

▲

# Down-monotonic Variables (mixed queries)

TBox



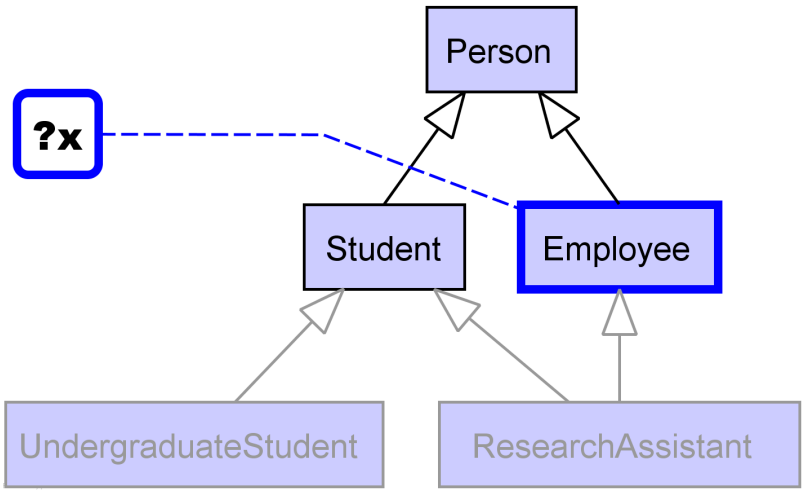

Query

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

▲

# 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.

# Queries with Undistinguished Variables

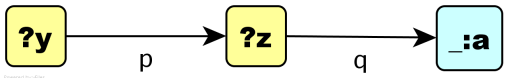
- for simplicity, let's consider just connected conjunctive ABox queries.

# Queries with Undistinguished Variables

- for simplicity, let's consider just connected conjunctive ABox queries.
- SoA Pellet engine simply rolls-up to all distinguished variables (DV) and performs an instance check for each candidate binding, or prunes invalid bindings during the execution.

# Queries with Undistinguished Variables

- for simplicity, let's consider just connected conjunctive ABox queries.
- SoA Pellet engine simply rolls-up to all distinguished variables (DV) and performs an instance check for each candidate binding, or prunes invalid bindings during the execution.
- ☹ which is inefficient in many cases :



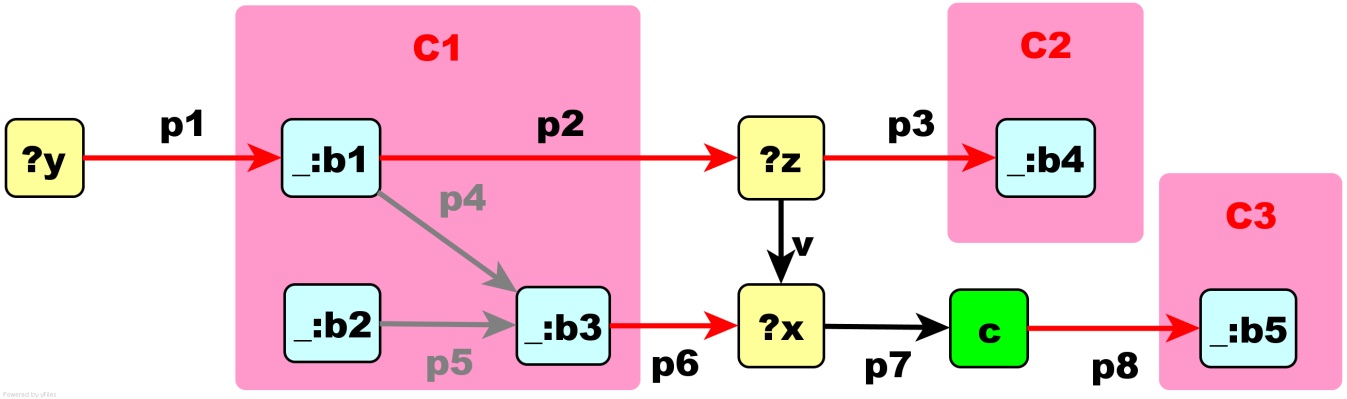

We should get rid of as many consistency checks as possible. The problem is when an edge $\mathit{PropertyValue}(?y, \mathrm{p}, ?z)$ is rolled-up to $\exists p \cdot T$. Retrieving instances of this concept in many cases ends-up finding also a valid binding for $?z$, which is then thrown away – and an additional check is performed later.

# Cores and Neighbourhoods

- Which DVs to roll-up ? How to evaluate them ? :

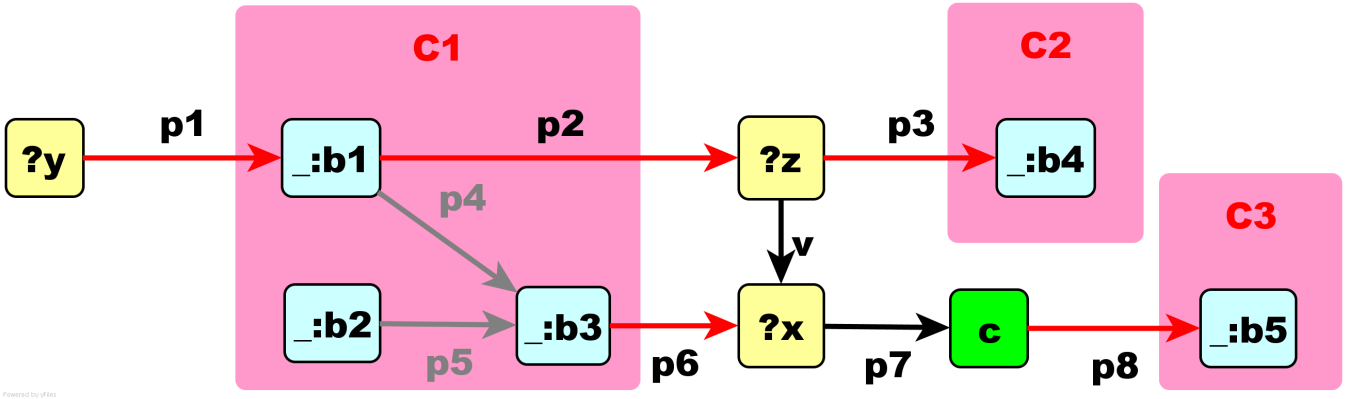
# Cores and Neighbourhoods

- Which DVs to roll-up ? How to evaluate them ? :



# Cores and Neighbourhoods

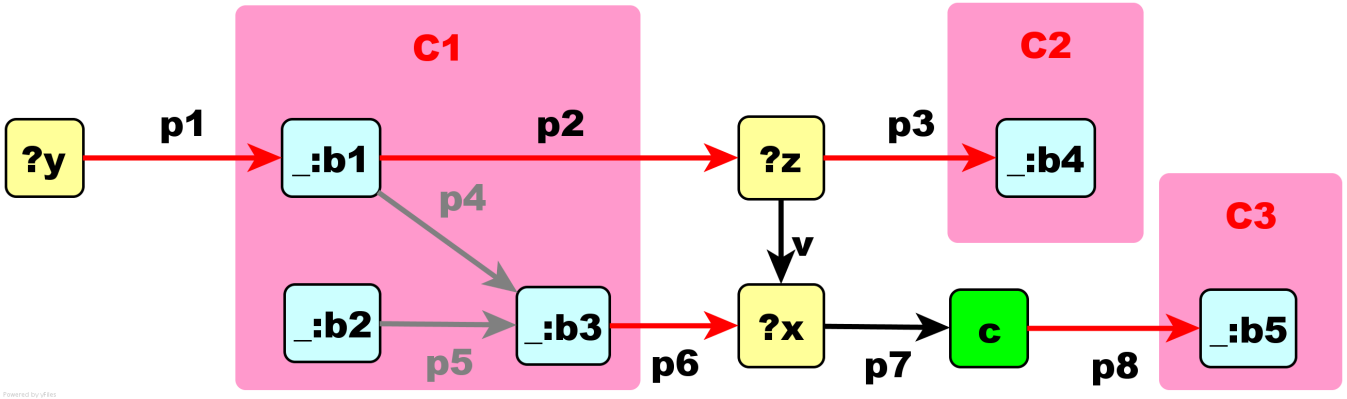
- Which DVs to roll-up ? How to evaluate them ? :




**core** is a maximal connected subgraph of the query induced by undistinguished variables (UV) .

# Cores and Neighbourhoods

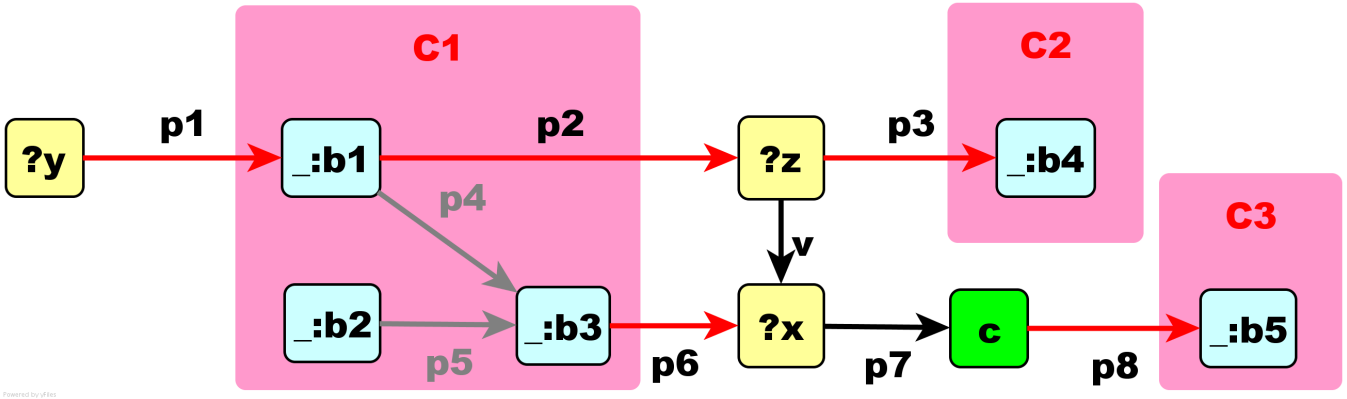
- Which DVs to roll-up ? How to evaluate them ? :




**core** is a maximal connected subgraph of the query induced by undistinguished variables (UV) .
**neighbour** of a core is a DV or constant (C) incident with some core node. There are different types of neighbour sets:

# Cores and Neighbourhoods

- Which DVs to roll-up ? How to evaluate them ? :

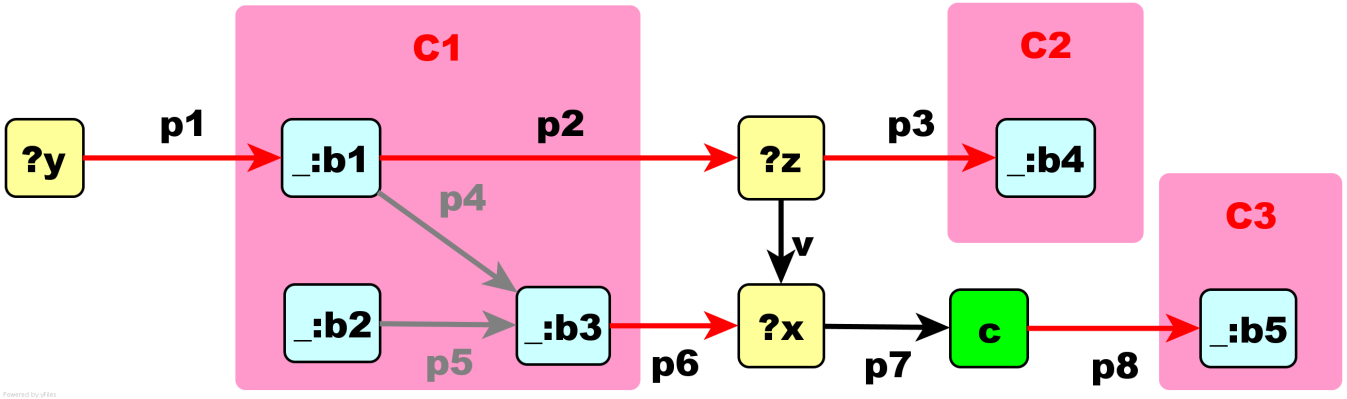



**core** is a maximal connected subgraph of the query induced by undistinguished variables (UV) .
**neighbour** of a core is a DV or constant (C) incident with some core node. There are different types of neighbour sets:

empty – the query is just a core of UV. We need to check that the rolled-up query has non-empty extension in *each* KB model.

# Cores and Neighbourhoods

- Which DVs to roll-up ? How to evaluate them ? :




**core** is a maximal connected subgraph of the query induced by undistinguished variables (UV) .

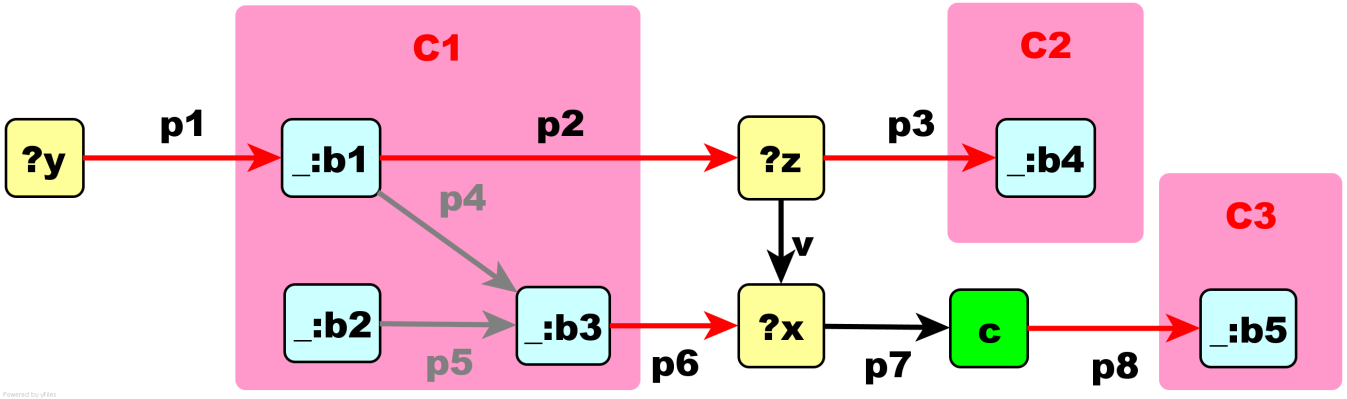
**neighbour** of a core is a DV or constant (C) incident with some core node. There are different types of neighbour sets:

empty – the query is just a core of UV. We need to check that the rolled-up query has non-empty extension in *each* KB model.

no DV, some C – just check that $C$ is an instance of $rollupto(C)$ .

# Cores and Neighbourhoods

- Which DVs to roll-up ? How to evaluate them ? :




**core** is a maximal connected subgraph of the query induced by undistinguished variables (UV) .
**neighbour** of a core is a DV or constant (C) incident with some core node. There are different types of neighbour sets:

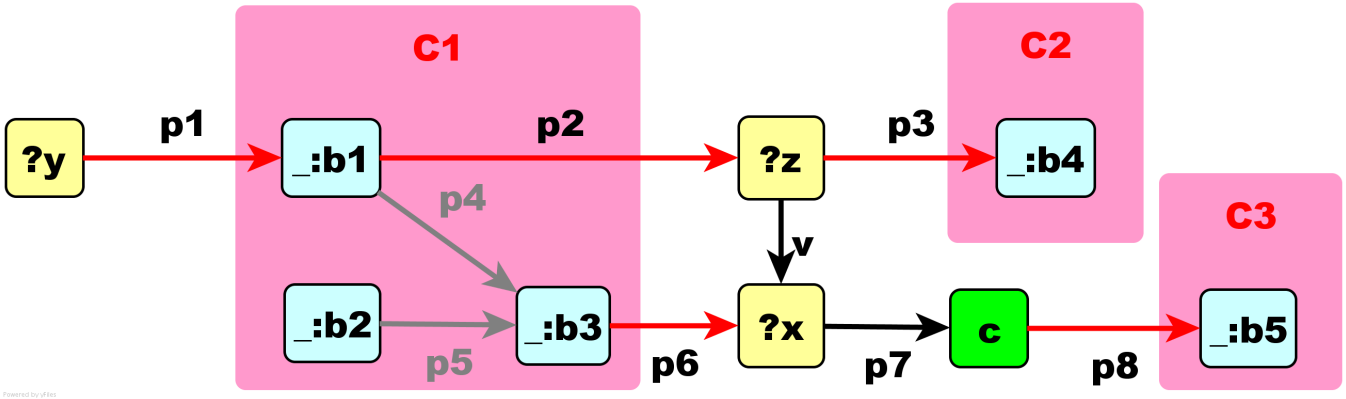
empty – the query is just a core of UV. We need to check that the rolled-up query has non-empty extension in *each* KB model.

no DV, some C – just check that $C$ is an instance of $rollupto(C)$ .

one DV – for each instance of $rollupto(DV)$ check the rest of the query.

# Cores and Neighbourhoods

- Which DVs to roll-up ? How to evaluate them ? :




**core** is a maximal connected subgraph of the query induced by undistinguished variables (UV) .
**neighbour** of a core is a DV or constant (C) incident with some core node. There are different types of neighbour sets:

empty – the query is just a core of UV. We need to check that the rolled-up query has non-empty extension in *each* KB model.

no DV, some C – just check that $C$ is an instance of $rollupto(C)$ .

one DV – for each instance of $rollupto(DV)$ check the rest of the query.

more DV – different evaluation strategies . . . .

# Core Evaluation Strategies

- . . . there are several strategies :

# Core Evaluation Strategies

- . . . there are several strategies :

  **simple** – SoA query engine strategy to evaluate whole queries.

# Core Evaluation Strategies

- . . . there are several strategies :

  **simple** – SoA query engine strategy to evaluate whole queries.

  **all fast** – distinguished variables are evaluated one by one and a candidate set for each DV is obtained by (cheap) filtering out all obvious non-instances of the rolled-up query.

# Properties of Cores

- the above strategies are under testing (but the “all fast” one seems to be most promising).

# Properties of Cores

- the above strategies are under testing (but the “all fast” one seems to be most promising).
- Cores :

# Properties of Cores

- the above strategies are under testing (but the “all fast” one seems to be most promising).
- Cores :
  - ☺ are cheap to compute

# Properties of Cores

- the above strategies are under testing (but the “all fast” one seems to be most promising).
- Cores :
  - ☺ are cheap to compute
  - ☺ allow us turn queries with UVs to queries with DVs. Cores become just extra n-ary atoms.

# Properties of Cores

- the above strategies are under testing (but the “all fast” one seems to be most promising).
- Cores :
  - ☺ are cheap to compute
  - ☺ allow us turn queries with UVs to queries with DVs. Cores become just extra n-ary atoms.
  - ☺ can be included to atom ordering.

# Properties of Cores

- the above strategies are under testing (but the “all fast” one seems to be most promising).
- Cores :
  - ☺ are cheap to compute
  - ☺ allow us turn queries with UVs to queries with DVs. Cores become just extra n-ary atoms.
  - ☺ can be included to atom ordering.
  - ☺ . . . thus can be evaluated using the already implemented strategy for queries with distinguished variables.

# 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`):

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

# Benchmarking with LUBM (2)

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

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

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

## Example (Q7 – Down-monotonic variables)

Find all people that take the same course they teach together with their classification and the course :

$\mathsf{SubClassOf}(?C, \texttt{Thing}), \mathsf{Type}(?x, ?C), \mathsf{PropertyValue}(?x, \texttt{takesCourse}, ?A),$
$\mathsf{PropertyValue}(?x, \texttt{teachingAssistantOf}, ?A)$

# Other Examples

## Example (Q11 – Long conjunctive ABox query)

Find all students and teachers from the same university that have some common publication, the student takes some course taught by the teacher and is a teaching assistent of another course taught by the teacher :

$$\begin{aligned}
&\text{Type}(?w, \texttt{University}), \text{PropertyValue}(?r, \texttt{subOrganizationOf}, ?w),\\
&\text{PropertyValue}(?x, \texttt{memberOf}, ?r), \text{PropertyValue}(?a, \texttt{memberOf}, ?r),\\
&\text{PropertyValue}(?x, \texttt{takesCourse}, ?z), \text{PropertyValue}(?a, \texttt{teacherOf}, ?z),\\
&\text{PropertyValue}(?a, \texttt{teacherOf}, ?z_2), \text{PropertyValue}(?x, \texttt{advisor}, ?b),\\
&\text{PropertyValue}(?x, \texttt{teachingAssistentOf}, ?z_2),\\
&\text{PropertyValue}(?q, \texttt{publicationAuthor}, ?x),\\
&\text{PropertyValue}(?q, \texttt{publicationAuthor}, ?a)
\end{aligned}$$

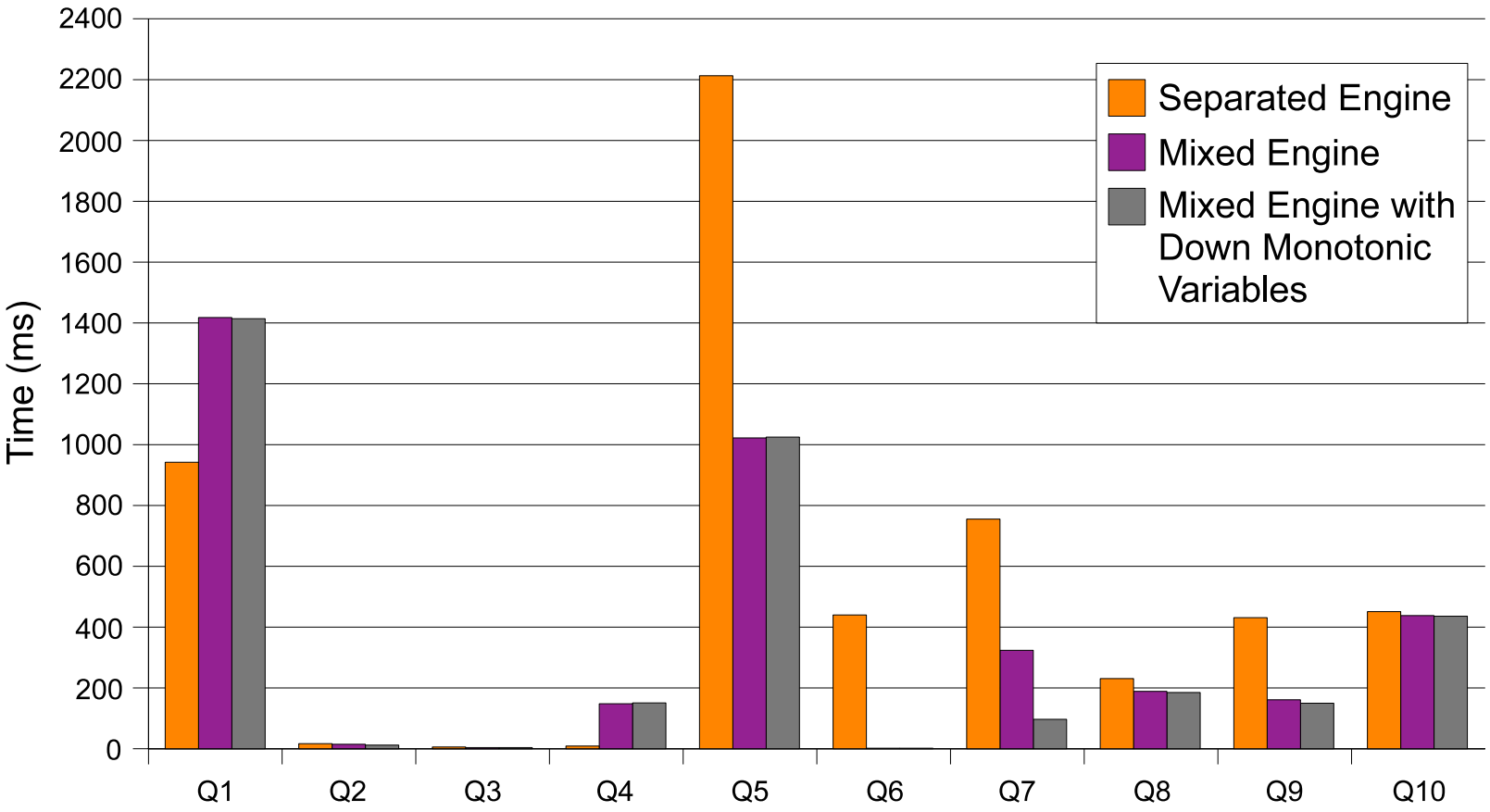
# Other Examples (2)

## Example (Q12 – conjunctive ABox query with undistinguished variable)

Find all students whose advisors teach a course they take :

$$\text{PropertyValue}(?x, \texttt{advisor}, \_ : a), \text{PropertyValue}(\_ : a, \texttt{teacherOf}, ?y),$$
$$\text{PropertyValue}(?x, \texttt{takesCourse}, ?y)$$

# 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, dynamic query reordering, cores and queries with undistinguished variables

# What to Do Next ?

- moving towards OWL 1.1

# What to Do Next ?

- moving towards OWL 1.1
- different cost functions

# What to Do Next ?

- moving towards OWL 1.1
- different cost functions
- more SPARQL stuff – optimized implementation of SPARQL algebra, like UNION, OPTIONAL, FILTER, etc . . .

# What to Do Next ?

- moving towards OWL 1.1
- different cost functions
- more SPARQL stuff – optimized implementation of SPARQL algebra, like UNION, OPTIONAL, FILTER, etc . . .
- . . . an much more

# Resources



Petr Kremen and Evren Sirin.
SPARQL-DL Implementation Experience.
In *OWLED*, 2008.


Evren Sirin and Bijan Parsia.
SPARQL-DL: SPARQL Query for OWL-DL.
In *OWLED*, 2007.