Supporting Users' Flexible Querying of Knowledge Graphs

Alex Poulovassilis

Joint work with Riccardo Frosini, Petra Selmer, Andrea Cali, Peter Wood

FQAS 2017, 21-22 June 2017

Knowledge Lab

Birkbeck
UNIVERSITY OF LONDON

Outline of the talk

Introduction and Motivation

Ontology-based relaxation of queries on RDF/S knowledge bases

Approximate matching of Regular Path Queries

Extending SPARQL 1.1 with approximation and relaxation
- RELAX
- APPROX
- FLEX

Ongoing and future work

Introduction and Motivation

Increasing volumes of graph-structured data arising from many application areas, e.g. RDF linked data on the web

Volumes, complexity and heterogeneity of data necessitates support for users' querying through flexible query processing techniques:

- users' queries do not have to match exactly the data structures being queried
- query system can automatically make changes to a query so as to help the user find relevant information
- answers to queries are returned in ranked order, in increasing "distance" from the original query

Introduction and Motivation

We look at unifying three kinds of flexible querying:

- query **relaxation** – returns *additional* answers compared to the exact form of the query
- query **approximation** – returns *different* answers compared to the exact form of the query
- **similarity matching** of literals – returns *additional* answers using text similarity measures over RDFS properties that contain text e.g.

  rdfs:comment, rdfs:label, and their subproperties, such as hasAbstract in DBPedia, hasGloss[ary] in YAGO

Recent work

A. Cali, R. Frosini, A. Poulovassilis and P. T. Wood: Flexible querying for SPARQL, Proc. ODBASE 2014.
R. Frosini, A. Cali, A. Poulovassilis and P. T. Wood: Flexible query processing for SPARQL, Semantic Web Journal, 8(4), pp 533-563, 2017

We have designed a SPARQL 1.1-based language called **SPARQL^{AR}** in which users can apply **APPROX** or **RELAX** operators to triple patterns within queries

Users (or the query system) can assign a **cost** to the application of any edit/relaxation operation to the property path appearing within an APPROXed/RELAXed triple pattern

Answers to queries are returned in **ranked order** of non-decreasing cost

In interacting with the query system, the user sets a **maximum overall cost**, to place an upper bound on the amount of approximation/relaxation that the system should apply to the query

Alternatively, user can ask for the **top-k answers** for some k

Current work

One question is, *how does a user decide which triple patterns should have APPROX applied to them and which RELAX?*

Also, it would be advantageous *for the user to interactively control*

- which specific edit/relaxation operations should be applied to which triple patterns
- which properties or classes should be affected
- the cost of each edit/relaxation operation applied

Could help the user understand the provenance of the query results returned, decide if answers are useful, try out different query edits/relaxations etc.

Current work

We present here a third operator – FLEX – that users can apply to triple patterns in their SPARQL 1.1 queries.

FLEX allows edit and relaxation operations to be applied *concurrently* to a triple pattern

Aim is to support greater ease of querying for users – they do not have to decide between using APPROX or RELAX

- Answers to queries are again returned in ranked order
- The user again selects a maximum cost for query answers
- Or a maximum number of answers, k

Ontology-based relaxation of queries on RDF/S Knowledge Bases

C.A.Hurtado, A. Poulovassilis, P. T. Wood: Query relaxation in RDF, Journal of Data Sematics X:31-61, 2008

Our data model comprises a directed graph $G = (N,E)$ and an ontology $K = (N_K,E_K)$

N contains nodes representing entity instances or entity classes, each labelled with a distinct constant

Each edge in E is labelled with a symbol drawn from a finite alphabet $\sum \cup \{type\}$

- **type** is used to connect an entity instance to its class

N_K contains nodes representing entity classes or properties, each labelled with a distinct constant. Each edge in E_K is labelled with a symbol from {sc,sp,dom,range}

The model encompasses RDF data, except for blank nodes. Plus a fragment of the RDFS vocabulary: rdf:type, rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain, rdfs:range (ρDF)

Ontology-based relaxation of queries

An RDF/S graph I_1 ***entails*** an RDF/S graph I_2 if I_2 can be derived from I_1 by applying the following rules iteratively to I_1:

(1) $\frac{(a,sp,b)\ (b,sp,c)}{(a,sp,c)}$ (2) $\frac{(a,sp,b)\ (x,a,y)}{(x,b,y)}$

(3) $\frac{(a,sc,b)\ (b,sc,c)}{(a,sc,c)}$ (4) $\frac{(a,sc,b)\ (x,type,a)}{(x,type,b)}$

(5) $\frac{(a,dom,c)\ (x,a,y)}{(x,type,c)}$ (6) $\frac{(a,range,d)\ (x,a,y)}{(y,type,d)}$

The ***closure*** of an RDF/S graph I under these rules is denoted by *cl* (I)

Ontology-based relaxation of queries

Query evaluation is on the graph obtained by restricting *cl* (G U K) to edges with labels in ∑ U {type} U propertyNodes(N_K)

- We call this the ***closure of the data graph G with respect to the ontology*** K, *closure* $_K$ (G)

Subgraphs of K induced by edges labelled **sc** or **sp** need to be *acyclic,* so that an unambiguous cost to be assigned to a relaxed query

Also, K must be *equal to its* ***extended reduction, extRed*** **(K)**, so that ***direct relaxations*** corresponding to the "smallest" possible relaxation steps can be unambiguously applied to queries; allows answers to be returned to users incrementally in order of non-decreasing cost

Ontology-based relaxation of queries

To compute *extRed* (K):

(a) compute *cl* (K);

(b) let D be the set of triples in *cl* (K) that can be derived using RDFS inference rules (1) or (3), or of rules (e1)-(e4) below;

(c) return *cl* (K) – D

$$\text{(e1)}\ \frac{(b,\text{dom},c)\ (a,\text{sp},b)}{(a,\text{dom},c)} \qquad \text{(e2)}\ \frac{(b,\text{range},c)\ (a,\text{sp},b)}{(a,\text{range},c)}$$

$$\text{(e3)}\ \frac{(a,\text{dom},b)\ (b,\text{sc},c)}{(a,\text{dom},c)} \qquad \text{(e4)}\ \frac{(a,\text{range},b)\ (b,\text{sc},c)}{(a,\text{range},c)}$$

Ontology-based relaxation of queries

Triple pattern (x, p, y) *directly relaxes* to triple pattern (x', p', y') w.r.t. ontology K = *extRed* (K) if *vars* (x,p,y) = *vars* (x',p',y') and (x',p',y') is derived from (x,p,y) by applying one of RDFS inference rules (1)-(6)

- each such application of a rule has a cost associated with it

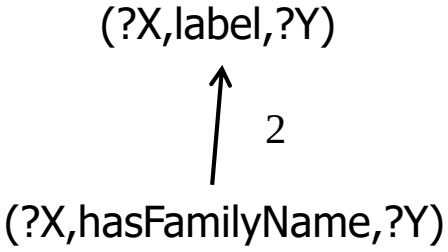
The *relaxation graph* of a triple pattern is the directed acyclic graph induced by the direct triple pattern relaxation relation

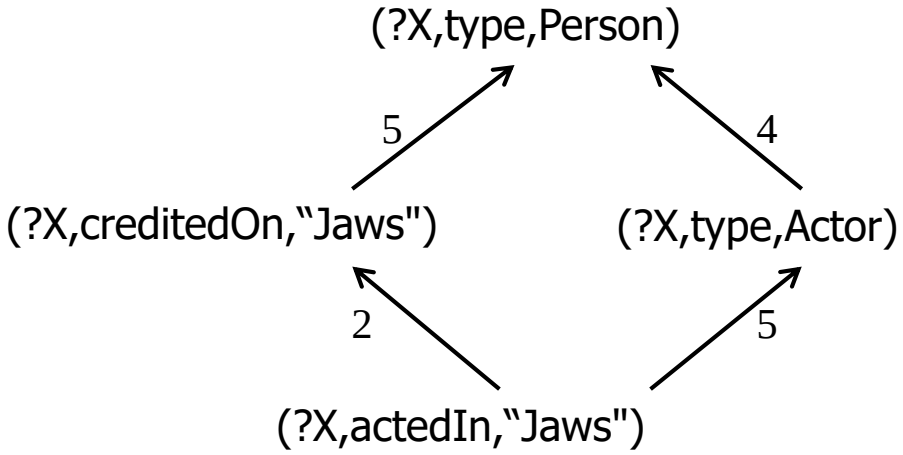
Triple pattern (x, p, y) *relaxes* to a triple pattern (x', p', y'), written (x, p, y) ≤ (x', p', y'), if there is a sequence of direct relaxations deriving (x', p', y') from (x, p, y)

- the *relaxation cost* is the minimum cost of such a sequence of direct relaxations

Example relaxation graphs for two triple patterns:

(?X, hasFamilyName, ?Y) and (?X,actedIn,“Jaws”)





Ontology-based relaxation of queries

Given a graph pattern P_n consisting of n triple patterns, the *graph pattern relaxation relation* $\leq_n$ is the direct product, n times, of $\leq$

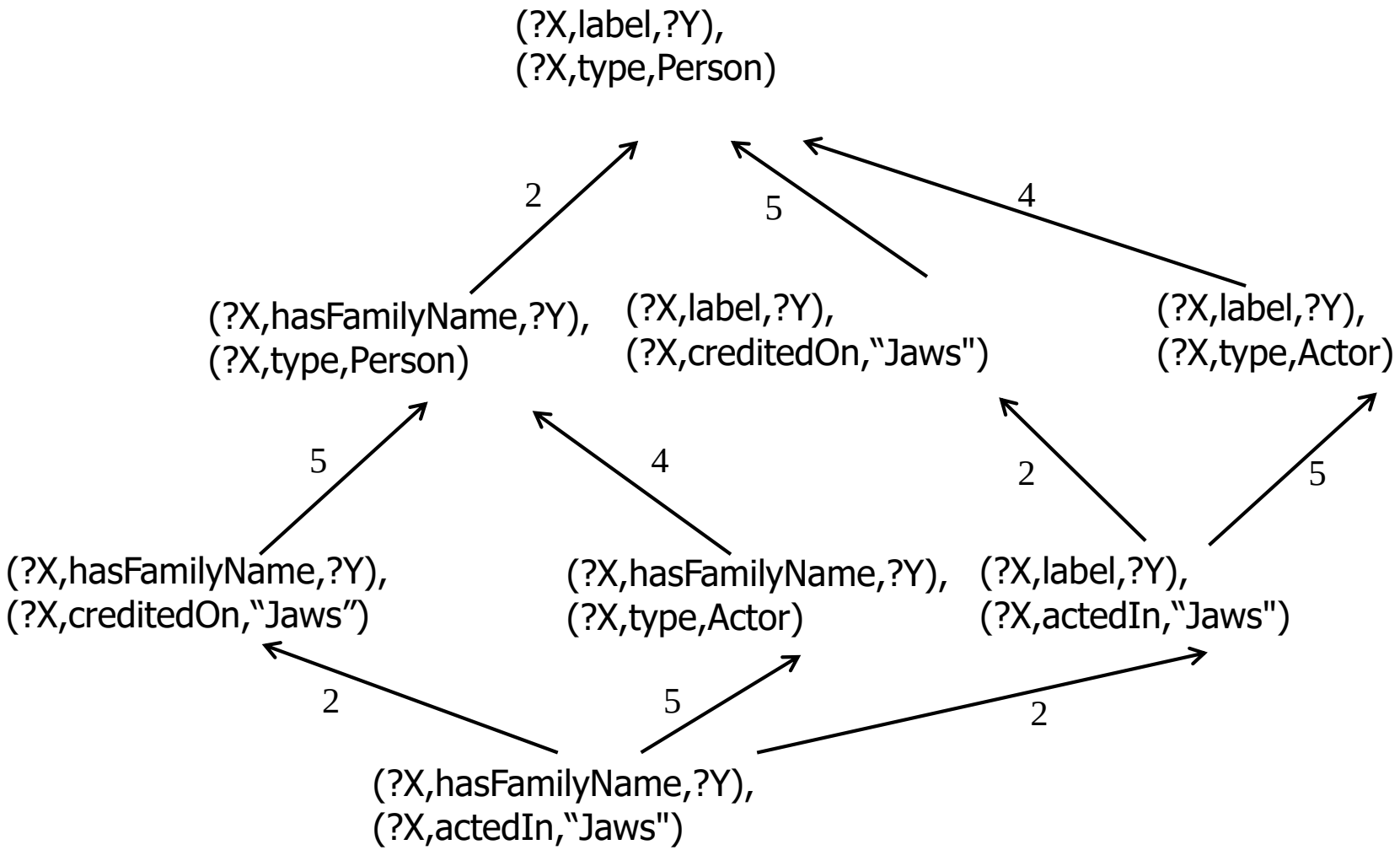
The *direct graph pattern relaxation relation* is the reflexive, transitive reduction of $\leq_n$

The *relaxation graph* of P_n is the directed acyclic graph induced by the direct graph pattern relaxation relation

E.g. the next slide shows the relaxation graph of graph pattern

(?X,hasFamilyName,?Y), (?X,actedIn,“Jaws”)

A graph pattern P_n *relaxes* to a graph pattern P'_n if there is a sequence of direct graph pattern relaxations that derives P'_n from P_n; the *relaxation cost* is the minimum cost of such a sequence of direct graph pattern relaxations


(?X,label,?Y),
(?X,type,Person)
2
5
4
(?X,hasFamilyName,?Y),
(?X,type,Person)
(?X,label,?Y),
(?X,creditedOn,“Jaws")
(?X,label,?Y),
(?X,type,Actor)
5
4
2
5
(?X,hasFamilyName,?Y),
(?X,creditedOn,“Jaws”)
(?X,hasFamilyName,?Y),
(?X,type,Actor)
(?X,label,?Y),
(?X,actedIn,“Jaws")
2
5
2
(?X,hasFamilyName,?Y),
(?X,actedIn,“Jaws")

Approximate Matching of Regular Path Queries

C.A.Hurtado, A. Poulovassilis, P. T. Wood. Ranking approximate answers to semantic web queries. ESWC 2009

Regular Path Queries (RPQs) assist users in querying complex or irregular graph-structured data by finding *paths* in the data graph that match a regular expression over edge labels

Same data model as before, comprising a directed graph G = (N,E)

- each node is labelled with a constant
- each edge e is labelled with a symbol l from a finite alphabet $\sum$ U {type}
- edges can be traversed in either direction
- for edge label l, l^- specifies reverse traversal of an edge
- for an already inverted label l^- in a query, $(l^-)^-$ is just l

Regular Path Queries

A *regular path query* (RPQ) is of form

$$(x, R, y)$$

where x, y are constants (node identifiers) or variables, and R is a regular expression over $\sum$ U {type}

A *regular expression* R *over* $\sum$ U {type} is defined as

$$R := \varepsilon \mid a \mid a^- \mid \_ \mid (R1.R2) \mid (R1|R2) \mid R^* \mid R^+$$

ε is the empty string, a is any symbol in $\sum$ U {type}, _ denotes the disjunction of all symbols in $\sum$ U {type}, the operators have their usual meaning

Exact matching of RPQs

A *semipath* p in a graph G from node n to node m is a sequence

$$v_1 , l_1 , v_2 , l_2 , \ldots, v_n , l_n , v_{n+1}$$

such that v_1=n, v_{n+1}=m, and for each v_i , l_i , v_{i+1} there is in G an edge $v_i \rightarrow v_{i+1}$ labelled l_i or an edge $v_{i+1} \rightarrow v_i$ labelled l_i^-

A semipath p *conforms* to a regular expression R if the sequence of labels $l_1 \ldots l_n$ is in $\mathcal{L}$(R), the language recognised by R

Exact matching of RPQs

Given an RPQ Q = (x, R, y), let θ be a matching from {x,y} to the nodes of graph G such that

- a constant is mapped to itself and
- there is a semipath from θ(x) to θ(y) which conforms to R

The *exact answer* of Q on G is the set of tuples θ(x,y) for all such matchings θ

Approximate matching of RPQs

Let q be a sequence of labels and l a label in $\sum \cup \sum^{-} \cup \{\text{type}, \text{type}^{-}\}$

We allow the following *edit operations*:

insertion of l into q

deletion of l from q

substitution of some label other than l by l in q

The application of each edit operation has a 'cost' associated with it, which may be user-specified or system-defined

inversion and *transposition* operations are subsumed by substitution

Approximate matching of RPQs

Consider a semipath p :

$$v_1 , l_1 , v_2 , l_2 , \ldots, v_n , l_n , v_{n+1}$$

and a semipath q :

$$w_1 , l'_1 , w_2 , l'_2 , \ldots, w_m , l'_m , w_{m+1}$$

The *edit distance from semipath* p *to semipath* q is the minimum cost of any sequence of edit operations which transforms the sequence of labels $l'_1 l'_2 \ldots l'_m$ to the sequence of labels $l_1 l_2 \ldots l_n$

Approximate matching of RPQs

The *edit distance from semipath* p *to regular expression* R, *edist* (p,R), is the minimum edit distance from p to any semipath conforming to R

Given graph G, query Q=(x,R,y), and matching θ, we define tuple θ(x,y) as having the *edit distance edist* (p,R), where p is a semipath from θ(X) to θ(Y) that has the minimum edit distance to R of *any* semipath from θ(X) to θ(Y)

The *approximate top-k answer* of Q on G is the list of *k* tuples θ(x,y) with minimum edit distance to Q, ranked in order of non-decreasing edit distance

Extending SPARQL with Approximation and Relaxation

See ODBASE 2014 and SWJ 2017 papers

We have investigated query relaxation and approximate matching in the pragmatic setting of **SPARQL 1.1**

SPARQL 1.1 supports regular path queries over the RDF graph – known as *property path queries*. But it does not support notions of query approximation or relaxation (except for OPTIONAL)

Our ODBASE 2014 and SWJ 2017 papers introduced **APPROX** and **RELAX** operators for property path queries: we termed the resulting language **SPARQL^{AR}**

We showed in those papers that this does not increase the complexity classes of the SPARQL 1.1 query fragments studied

Semantics of flexible SPARQL queries

For specifying the query semantics we extend SPARQL query evaluation, which returns a set of *mappings*. A mapping is a partial function

$$\mu : U \cup L \cup V \rightarrow U \cup L$$

such that $\mu(x)=x$ for all x in $U \cup L$, where U, L, V are pairwise disjoint sets of URIs, literals and variables

In our case, query evaluation returns a set of *mapping/cost pairs*

$$(\mu,c)$$

c is a non-negative number indicating the cost of answers arising from mapping μ, i.e. the sum of the costs of the edit and relaxation operations applied to the original query to generate this mapping

Complexity of flexible SPARQL query evaluation

Our complexity proofs hinge on:

- the construction of an *approximate automaton* for query triple patterns that have APPROX applied to them
- the construction of a *relaxed automaton* for query triple patterns that have RELAX to them
- the construction of a *weighted product automaton* H of the graph G with the approximate or relaxed automaton
- a shortest path traversal in H using Dijkstra's algorithm

Automata for APPROX

If **APPROX** has been applied to (x,P,y):

1. Construct a weighted NFA M_P to recognise $\mathcal{L}(P)$:

M_P has states S and transitions T; weights on all transitions are 0

2. Construct the *approximate automaton* A_P corresponding to M_P:

same set of states S as M_P ;

additional transitions corresponding to edge label insertions/ deletions/substitutions whose weights are the costs of these operations: A_P has $|S|$ states and $O(|S|^2)$ transitions

3. Form the weighted *product automaton* H of A_P and G, viewing each node in G as both an initial and a final state:

$O(|S||N|)$ states and $O(|S|^2|E|)$ transitions

Automata for RELAX

If **RELAX** has been applied to (x,P,y):

1. Construct a weighted NFA M_P to recognise $\mathcal{L}(P)$:

M_P has states S and T; weights on all transitions are 0

2. Construct the *relaxed automaton* R_P corresponding to M_P:

add to M_P all transitions and states that can be inferred by iteratively applying the RDFS inference rules, incrementing the transition weights with each new rule application;

R_P has $O(|S||N_K|)$ states and $O(|S|^2|N_K||E_K|)$ transitions

3. Form the weighted *product automaton* H of R_P and G, viewing each node in G as both an initial and a final state:

$O(|S||N_K||N|)$ states and $O(|S|^2|N_K||E_K||E|)$ transitions

SPARQL^{AR} complexity results

See ODBASE 2014 and SWJ 2017 papers for details

Operators	Data Complexity	Query Complexity	Combined Complexity
RELAX, APPROX	O(|E|)	P-Time	P-Time
AND,FILTER,RegEx	O(|E|)	O(|Q|^2)	O(|E| |Q|^2)
AND,FILTER,RegEx,RELAX, APPROX	O(|E|)	P-Time	P-Time
AND,RegEx,FILTER, SELECT	P-Time	NP-Complete	NP-Complete
AND,RegEx,FILTER, SELECT, RELAX, APPROX	P-Time	NP-Complete	NP-Complete
AND,RegEx,FILTER, SELECT, UNION, RELAX, APPROX	P-Time	NP-Complete	NP-Complete

SPARfL

We now go a step further and propose a FLEX operator, which users can apply to selected triple patterns. We call the resulting language SPARfL

FLEX allows approximation and relaxation operations to be applied *concurrently* to a triple pattern

Aims to allow greater ease of querying for users, so that they are not compelled to choose one of APPROX or RELAX

Answers are returned in ranked order, in order of non-decreasing cost

flex operations

Given an ontology K = *extRed* (K) and a graph G = *closure* $_K$ (G), a *flex operation* is

- either an *edit operation* on a symbol in $\sum \cup \sum^-$
- or a *direct relaxation operation* using ontology K

Given a triple pattern Q=(x,P,y), matching θ, semipath q from θ(x) to θ(y) conforming to P, and semipath p:

- the *distance from* p *to* q is the minimum cost of any sequence of flex operations that transforms q to p;
- the *distance from* p *to* θ(Q) is the minimum distance from p to any semipath q from θ(x) to θ(y) conforming to P
- the *distance of tuple* θ(x,y) is the minimum distance to θ(Q) from *any* semipath in G
- the *top-k answer* of Q on G is the list of *k* tuples θ(x,y) with minimum distance to Q, ranked in order of non-decreasing distance

Ontology-based relaxation of RPQs

The next slide shows the relaxation graph of q where Q is the RPQ

(?Y, hasFamilyName^-/actedIn, “Jaws”)

and q is the sequence of labels $\text{hasFamilyName}^- \circ \text{actedIn}$

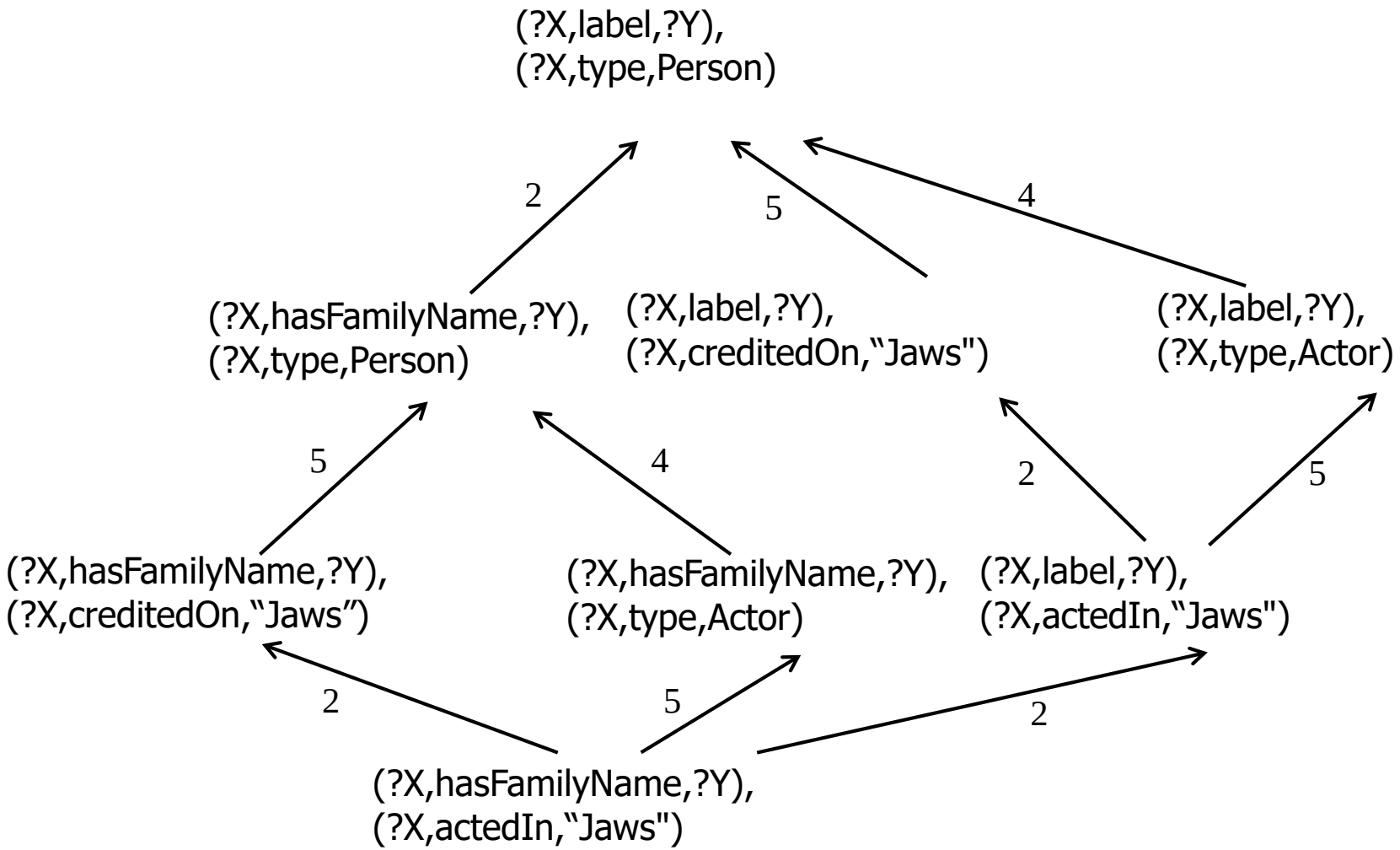
This has the “triple form”

($Y?$, hasFamilyName^-, ?X), (?X, actedIn,“Jaws”)

for a fresh variable ?X

This triple form is normalised to

(X?, hasFamilyName, ?Y), (?X, actedIn,“Jaws”)


(?X,label,?Y),
(?X,type,Person)
2
5
4
(?X,hasFamilyName,?Y),
(?X,type,Person)
(?X,label,?Y),
(?X,creditedOn,“Jaws")
(?X,label,?Y),
(?X,type,Actor)
5
4
2
5
(?X,hasFamilyName,?Y),
(?X,creditedOn,“Jaws”)
(?X,hasFamilyName,?Y),
(?X,type,Actor)
(?X,label,?Y),
(?X,actedIn,“Jaws")
2
5
2
(?X,hasFamilyName,?Y),
(?X,actedIn,“Jaws")

Approximation of RPQs

Consider RPQ

(?Y, hasFamilyName$^-$/actedIn, “Jaws”)

and the sequence of labels hasFamilyName$^-$ ◦ actedIn

This can be converted to the triple form

(Y?, hasFamilyName$^-$, ?X), (?X, actedIn,“Jaws”)

where ?X is a fresh variable

Approximation of RPQs

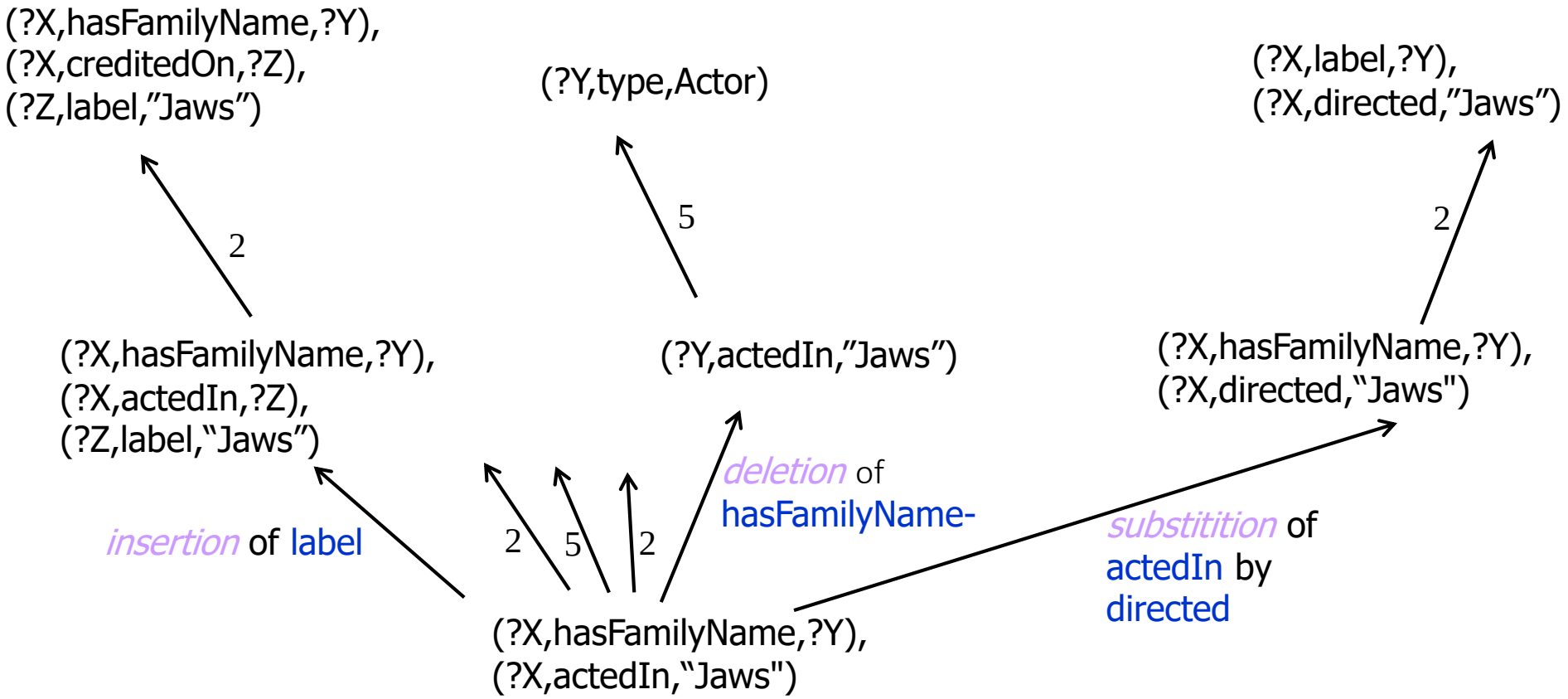
Starting with (Y?, hasFamilyName$^-$, ?X), (?X, actedIn,“Jaws”):

insertion of label may result in
(Y?, hasFamilyName$^-$,?X), (?X, actedIn, ?Z), (?Z, label “Jaws”)
(for fresh ?Z)

deletion of hasFamilyName$^-$ may result in
(?Y, actedIn,“Jaws”)

substitution of actedIn by directed may result in
(Y?, hasFamilyName$^-$, ?X), (?X, directed,“Jaws”)

FLEXing of RPQs – combining relaxation and approximation possibilities (partial)



Flexible Querying of YAGO – Example 1

Suppose the user wants to find "What are the geographical coordinates of the 'Battle of Waterloo' event?" Knows that the URL of the resource representing the Battle of Waterloo in YAGO is

http://yago-knowledge.org/resource/Battle_of_Waterloo

but is not certain about how to find its geographical coordinates.

Poses this SPARfL query, setting *max-cost* to 0 initially (in the hope that the exact form of the query will return the required information):

```
SELECT * WHERE {
    FLEX  (
      <http://yago-knowledge.org/resource/Battle_of_Waterloo>
      yago:happenedIn/(yago:hasLongitude|yago:hasLatitude)
      ?x ) }
```

The system finds no exact answers. The user therefore increases *max-cost* to 1

Suppose the user has assigned

- a cost of 1 to all edit operations
- a cost of 1 to relaxation rules 2 & 4
- a cost of 2 to relaxation rules 5 & 6

The user selects the following options when prompted by the system regarding which relaxation and edit operations are applicable:

- Relaxation rule 2? yes
- Relaxation rule 4? yes
- Relaxation rule 5? yes
- Relaxation rule 6? yes

Insert? yes

The system then asks the user which of the properties p appearing in the FLEXed triple pattern should be subject to insertion of a property (either to the left or to the right of p). User specifies

happenedIn – yes ; hasLongitude – no ; hasLatitude – no

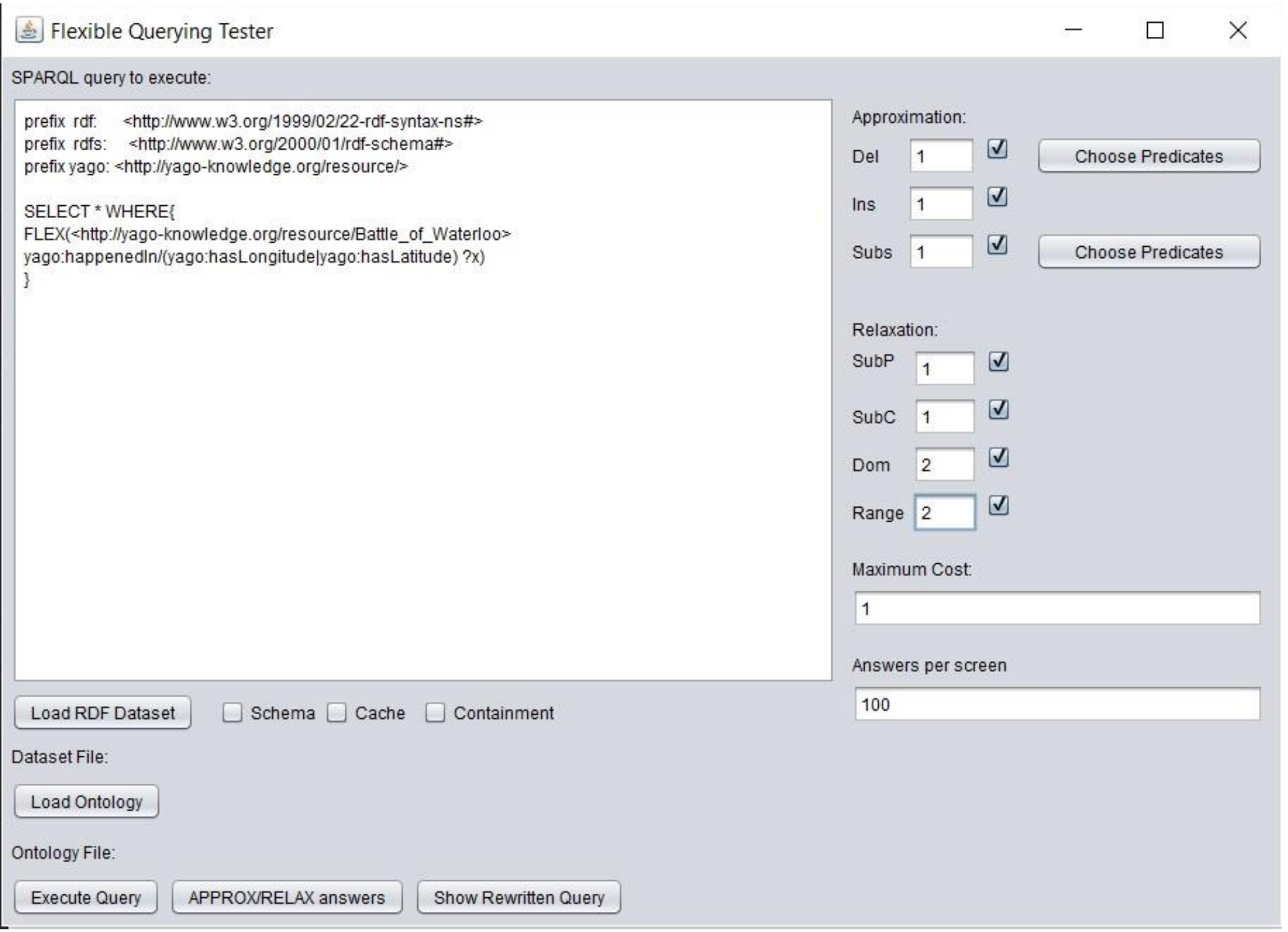
Delete? yes

The system then asks the user to specify which properties should be subject to deletion (the user may wish to "keep" some of them). User specifies

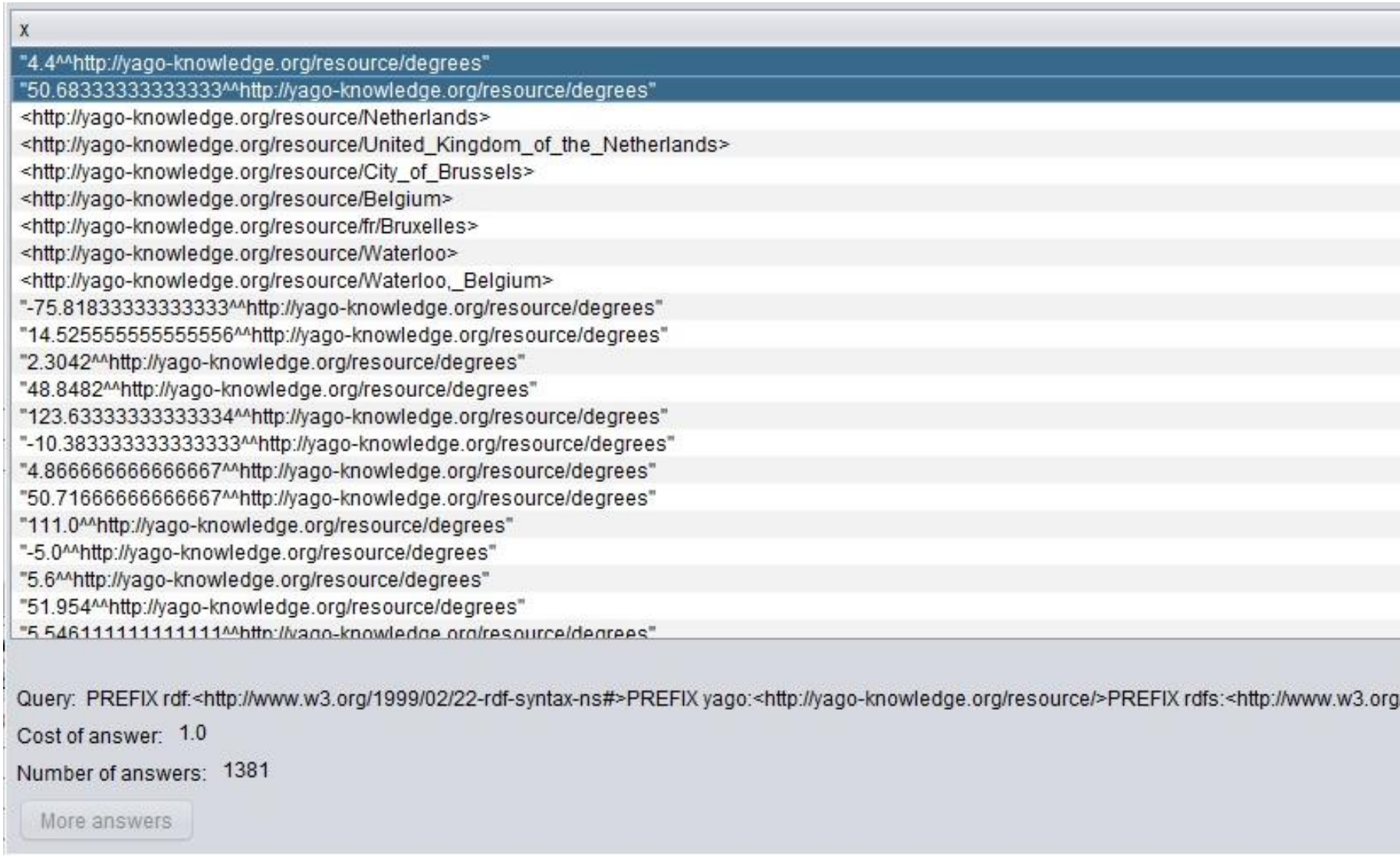
happenedIn – yes ; hasLongitude – no ; hasLatitude – no

Substitute? yes The system then asks the user to specify which properties can be substituted. User specifies

happenedIn – yes ; hasLongitude – yes ; hasLatitude–yes


Flexible Querying Tester
SPARQL query to execute:
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix yago: <http://yago-knowledge.org/resource/>
SELECT * WHERE{
FLEX(<http://yago-knowledge.org/resource/Battle_of_Waterloo>
yago:happenedIn/(yago:hasLongitude|yago:hasLatitude) ?x)
}
Approximation:
Del 1
Choose Predicates
Ins 1
Subs 1
Choose Predicates
Relaxation:
SubP 1
SubC 1
Dom 2
Range 2
Maximum Cost:
1
Answers per screen
100
Load RDF Dataset
Schema
Cache
Containment
Dataset File:
Load Ontology
Ontology File:
Execute Query
APPROX/RELAX answers
Show Rewritten Query

First screen of answers:



One of the resulting cost-1 queries returns the answers the user is seeking: application of a Delete to property happenedIn gives query

```
SELECT * WHERE {
    <http://yago-knowledge.org/resource/Battle_of_Waterloo>
    (yago:hasLongitude|yago:hasLatitude)
    ?x ) }
```

YAGO does store directly the coordinates of the “Battle of Waterloo" event, and the resulting query returns the desired answers:

"4.4"^^<degrees>

"50.68333333333333"^^<degrees>

Example 2

Suppose the user is familiar with this part of the YAGO ontology:

Nodes: hasFamilyName, hasGivenName, label, actedIn, Actor

Edges: (hasFamilyName, sp, label), (hasGivenName, sp, label),
(actedIn, domain, actor)

Wants to find "What are the family names of actors who played in the film 'Tea with Mussolini'?" Knows the URL of the resource representing that film in YAGO. So poses the query

```
SELECT * WHERE {
  ?x yago:actedIn <http://yago-knowledge.org/resource/
                   Tea_with_Mussolini> .
  ?x yago:hasFamilyName ?z }
```

The query returns only a few answers (many actors have their full name recorded using the property label, not FamilyName). User may choose to flexibly match the second triple pattern, in an attempt to retrieve more answers, setting *max-cost* to 1:

```
SELECT * WHERE {
  ?x yago:actedIn <http://yago-knowledge.org/resource/
                   Tea_with_Mussolini> .
  FLEX ( ?x yago:hasFamilyName ?z ) }
```

The user selects the following options when prompted by the system:

Relaxation rules 2, 4, 5, 6? yes

Insert? yes (there is only one property, hasFamilyName, that can be subject to insertion)

Delete? no (the user doesn't want it to be deleted)

Substitute? yes (there is only one property to be substituted)

One of the resulting cost-1 queries will return the answers the user is seeking: application of Rule 2 (property relaxation) to hasFamilyName replaces it by label giving the query

```
SELECT * WHERE {
  ?x yago:actedIn <http://yago-knowledge.org/resource/
                   Tea_with_Mussolini> .
  ?x yago:label ?z }
```

This query returns names recorded through the label property, and also through its hasGivenName and hasFamilyName subproperties; this includes the original results, plus many more

First screen of cost-1 answers:

x	z
<http://yago-knowledge.org/resource/Cher>	"Allman"
<http://yago-knowledge.org/resource/Cher>	"Bono"
<http://yago-knowledge.org/resource/Joan_Plowright>	"Olivier"
<http://yago-knowledge.org/resource/Judi_Dench>	"Williams"
<http://yago-knowledge.org/resource/Maggie_Smith>	"Stephens"
<http://yago-knowledge.org/resource/Maggie_Smith>	"Cross"
<http://yago-knowledge.org/resource/Maggie_Smith>	"Larkin"
<http://yago-knowledge.org/resource/Lily_Tomlin>	"Wagner"
<http://yago-knowledge.org/resource/Lily_Tomlin>	"Montano"
<http://yago-knowledge.org/resource/Cher>	"Cher"
<http://yago-knowledge.org/resource/Cher>	"Cher@eng"
<http://yago-knowledge.org/resource/Cher>	"Cher (cantant)@cat"
<http://yago-knowledge.org/resource/Cher>	"Cher@deu"
<http://yago-knowledge.org/resource/Cher>	"ara@شير"
<http://yago-knowledge.org/resource/Cher>	"Cher@lit"
<http://yago-knowledge.org/resource/Cher>	"Cher@ita"
<http://yago-knowledge.org/resource/Cher>	"Sera@lav"
<http://yago-knowledge.org/resource/Cher>	"Cherilyn Sarkisian"
<http://yago-knowledge.org/resource/Cher>	"Cher (spevacka)@slk"
<http://yago-knowledge.org/resource/Cher>	"Шэр@bel"
<http://yago-knowledge.org/resource/Cher>	"Šēra@lav"
<http://yago-knowledge.org/resource/Cher>	"Artisten Cher@nno"

Query: PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>PREFIX yago:<http://yago-knowledge.org/resource/>PREFIX rdfs:<http://www.w3.org/2000/01/rdf-s

Cost of answer: 1.0

Number of answers: 850

More answers

Example 3

Suppose the user is a History of Science researcher and wishes to find people born in London who have some connection to science. Poses this query:

```
SELECT * WHERE {
        ?p yago:wasBornIn/yago:isLocatedIn yago:London .
        ?p rdf:type ?c . ?c yago:hasGloss “scientist” }
```

This query returns no results, as there is no precise match of the property hasGloss with the literal “scientist”

Using similarity matching, results can be returned whose glossary description contains the word “scientist” or similar

```
SELECT * WHERE {
   ?p yago:wasBornIn/yago:isLocatedIn yago:London .
   ?p rdf:type ?c .
   SIM (?c yago:hasGloss “scientist”) }
```

Similarity is in the range [0,1]

The cost of the SIM triple is c x (1 – similarity)

weighting c can be set by the user or the system

Suppose the user sets c to 1 and *max-cost* to 1

Top-10 results are:

<http://yago-knowledge.org/resource/Joseph_Jackson_Lister>	**<http://yago-knowledge.org/resource/wordnet_scientist_110560637>**
<http://yago-knowledge.org/resource/Edward_Upward>	<http://yago-knowledge.org/resource/wordnet_person_100007846>
<http://yago-knowledge.org/resource/James_Marcus>	<http://yago-knowledge.org/resource/wordnet_person_100007846>
<http://yago-knowledge.org/resource/Joseph_Tremain>	<http://yago-knowledge.org/resource/wordnet_person_100007846>
<http://yago-knowledge.org/resource/David_Eldridge_(dramatist)>	<http://yago-knowledge.org/resource/wordnet_person_100007846>
<http://yago-knowledge.org/resource/Michael_Nicholson>	<http://yago-knowledge.org/resource/wordnet_person_100007846>
<http://yago-knowledge.org/resource/Chris_Jarvis_(presenter)	<http://yago-knowledge.org/resource/wordnet_person_100007846>
<http://yago-knowledge.org/resource/Glen_Berry>	<http://yago-knowledge.org/resource/wordnet_person_100007846>
<http://yago-knowledge.org/resource/Millicent_Martin>	<http://yago-knowledge.org/resource/wordnet_person_100007846>
<http://yago-knowledge.org/resource/Catherine_McGoohan>	<http://yago-knowledge.org/resource/wordnet_person_100007846>

Only one scientist is found (the similarity is 0.8 and hence the cost of the answer 0.2) – Joseph Jackson Lister

The user therefore decides to expand the search, to include scientists who are more loosely connected with London, and poses this query:

```
SELECT * WHERE {
    FLEX (?p yago:wasBornIn/yago:isLocatedIn yago:London) .
    ?p rdf:type ?c . ?c yago:hasGloss ?string .
    SIM (?c yago:hasGloss “scientist”) }
```

User sets *max-cost* to 1.5

selects Yes for all relaxation rules

selects No for Insert and Delete

selects Yes for Substitute,

for both properties wasBornIn and isLocatedIn

Several resulting queries will return relevant answers, e.g.

```
SELECT DISTINCT ?p  WHERE {
   ?p _ / yago:isLocatedIn yago:London .
   ?p rdf:type ?c . ?c yago:hasGloss ?string .
   SIM (?c yago:hasGloss “scientist”) }
```

obtained through a substitution of wasBornIn by _

First 22 non-exact results (all at cost 1.2) are:

p
<http://yago-knowledge.org/resource/Joseph_Jackson_Lister>
<http://yago-knowledge.org/resource/Naomi_Oreskes>
<http://yago-knowledge.org/resource/Andrew_Fabian>
<http://yago-knowledge.org/resource/William_Robert_Grove>
<http://yago-knowledge.org/resource/S._Francis_Boys>
<http://yago-knowledge.org/resource/John_Beddington>
<http://yago-knowledge.org/resource/Roger_Woolger>
<http://yago-knowledge.org/resource/John_Napier_(primatologist)>
<http://yago-knowledge.org/resource/Robert_Hooke>
<http://yago-knowledge.org/resource/Morris_Ginsberg>
<http://yago-knowledge.org/resource/Edward_Routh>
<http://yago-knowledge.org/resource/Tony_Brooker>
<http://yago-knowledge.org/resource/Nancy_Rothwell>
<http://yago-knowledge.org/resource/Frederick_Abel>
<http://yago-knowledge.org/resource/Nicholas_Kemmer>
<http://yago-knowledge.org/resource/Walter_Thomas_James_Morgan>
<http://yago-knowledge.org/resource/Frederick_Gowland_Hopkins>
<http://yago-knowledge.org/resource/Ronald_Sydney_Nyholm>
<http://yago-knowledge.org/resource/Martin_Fleischmann>
<http://yago-knowledge.org/resource/Salimuzzaman_Siddiqui>
<http://yago-knowledge.org/resource/John_Grenville>
<http://yago-knowledge.org/resource/Frederick_Twort>

Query: PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>PREFIX yago:<http://yago-knowledge.org/resource/>PREFIX rdfs:<http://www.w3.org/2000/

Suppose the user is interested in retrieving more answers of a similar kind and increases *max-cost* to 2.5

Several resulting queries will return relevant answers, e.g.

```
SELECT DISTINCT ?p  WHERE {
   ?p _ / _  yago:London .
   ?p rdf:type ?c . ?c yago:hasGloss ?string .
   SIM (?c yago:hasGloss “scientist”) }
```

obtained through a second substitution, of isLocatedIn by _

First 22 results at cost > 1.5 (all at cost 2.2) are:

p
<http://yago-knowledge.org/resource/John_David_Kennedy>
<http://yago-knowledge.org/resource/Adam_Kendon>
<http://yago-knowledge.org/resource/John_Canton>
<http://yago-knowledge.org/resource/William_Roy>
<http://yago-knowledge.org/resource/Angus_Maddison>
<http://yago-knowledge.org/resource/Crispin_Nash-Williams>
<http://yago-knowledge.org/resource/Geoffrey_Douglas_Hale_Carpenter>
<http://yago-knowledge.org/resource/Geoffrey_Dummer>
<http://yago-knowledge.org/resource/Henry_Hallett_Dale>
<http://yago-knowledge.org/resource/John_Cranke>
<http://yago-knowledge.org/resource/Thomas_Edmondson>
<http://yago-knowledge.org/resource/John_Hopkinson>
<http://yago-knowledge.org/resource/Siegfried_Frederick_Nadel>
<http://yago-knowledge.org/resource/C._V._Boys>
<http://yago-knowledge.org/resource/John_Ralfs>
<http://yago-knowledge.org/resource/John_Venn>
<http://yago-knowledge.org/resource/Grahame_Clark>
<http://yago-knowledge.org/resource/Sigmund_Freud>
<http://yago-knowledge.org/resource/Horace_Lamb>
<http://yago-knowledge.org/resource/Saunders_Mac_Lane>
<http://yago-knowledge.org/resource/W._B._R._Lickorish>
<http://yago-knowledge.org/resource/Joshua_King>

Query: PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>PREFIX yago:<http://yago-knowledge.org/resource/>PREFIX rdfs:<http://www.w3

To explore scientists with even looser connections with London, the user can now select Yes for Insert and Delete, for both properties, leaving *max-cost* the same as before (2.5)

Several resulting queries will return relevant answers, e.g.

```
SELECT * WHERE {
   ?p yago:isPlacedIn yago:London .
   ?p rdf:type ?c . ?c yago:hasGloss ?string .
   SIM (?c yago:hasGloss “scientist”) }
```

obtained by deletion of wasBornIn, and relaxation of isLocatedIn to isPlacedIn

Additional results are:

p
<http://yago-knowledge.org/resource/Henry_Hallett_Dale>
<http://yago-knowledge.org/resource/William_Ross_Ashby>
<http://yago-knowledge.org/resource/Jon_Speelman>
<http://yago-knowledge.org/resource/Sandy_Douglas>
<http://yago-knowledge.org/resource/Michael_Atiyah>
<http://yago-knowledge.org/resource/Walter_Thomas_James_Morgan>
<http://yago-knowledge.org/resource/Francis_Haskell>
<http://yago-knowledge.org/resource/William_Jackson_Pope>
<http://yago-knowledge.org/resource/Arnold_Toynbee>
<http://yago-knowledge.org/resource/Joseph_Banks>
<http://yago-knowledge.org/resource/Jocelyn_Field_Thorpe>
<http://yago-knowledge.org/resource/John_Hadley>
<http://yago-knowledge.org/resource/Ronald_Rivlin>
<http://yago-knowledge.org/resource/John_Pond>
<http://yago-knowledge.org/resource/Sidney_Gilchrist_Thomas>
<http://yago-knowledge.org/resource/Henry_Gray>
<http://yago-knowledge.org/resource/Arthur_T._Ippen>
<http://yago-knowledge.org/resource/Henry_Enfield_Roscoe>
<http://yago-knowledge.org/resource/Robert_H._Crabtree>
<http://yago-knowledge.org/resource/William_Conybeare_(geologist)>
<http://yago-knowledge.org/resource/Paul_Bernays>
<http://yago-knowledge.org/resource/William_Henry_Perkin>

Query: PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>PREFIX yago:<http://yago-knowledge.org/resource/>PREFIX rdfs:<http://www

Complexity of querying with FLEX

Complexity proof hinges on:

1. construction first of an approximate automaton A_P for each query triple pattern (x,P,y) that has FLEX applied to it:

$O(|S|)$ states and $O(|S|^2)$ transitions

2. construction of the relaxed automaton of A_P i.e. the *relaxed-approximate automaton* of P, RA_P:

$O(|S||N_K|)$ states and $O(|S|^2|N_K||E_K|)$ transitions

3. construction of the weighted product automaton H of the graph G with RA_P :

$O(|S||N_K||N|)$ states and $O(|S|^2|N_K||E_K||E|)$ transitions

4. Performing a shortest path traversal of H using Dijkstra's algorithm

Complexity of querying with FLEX

No increase in complexity classes compared to SPARQL^{AR}, subject to the proviso that edit operations are not allowed on type and type$^-$

This proviso means that the relaxed-approximate automaton RA_P can be constructed in two steps, as above, and is bounded in size to $O(|S||N_K||N|)$ states and $O(|S|^2|N_K||E_K||E|)$ transitions

Proof of this can be found in A.Poulovassilis, P.Selmer, P.T.Wood: Approximation and relaxation of semantic web path queries, J. Web Semantics, 40, pp 1-21, 2016

Complexity of SIM

SIM traverses properties in WordNet in order to make semantic connections between the search term and an actual text snippet in the data

It takes account of the path length between words in WordNet (up to a maximum length N) and the weights associated with the properties connecting them

If there are D different properties in WordNet and a maximum of n words in a text snippet, the complexity is $O(nD^N)$

For more details, see See A. Calì, S. Capuzzi, M. M. Dimartino, Riccardo Frosini, Recommendation of Text Tags in Social Applications Using Linked Data. ICWE Workshops 2013, pp 187-191

Query Rewriting-based Implementation

Similarly to SPARQL^{AR} (see ODASE 2014 and SWJ 2017 papers) we adopt a *query rewriting approach* whereby a SPARfL query Q is rewritten to a set of SPARQL 1.1 queries for evaluation:

Query Rewriting Algorithm starts by generating the query Q_0 that returns the exact answer of Q

For each FLEXed triple pattern (x_i, P_i, y_i) in Q and each URI p appearing in P_i, a set of new queries is constructed from Q_0 by applying all possible one-step edit and relaxation operations to p: these are the `1st-generation' queries

To each 1st-generation query Q_1 is assigned the cost of applying the edit or relaxation operation that derived it

A new set of queries is constructed by applying a second step of approximation and relaxation to each 1st-gen query Q_1 – these are the `2nd-generation' queries; we accumulate summatively the cost of the 2 edit or relaxation operations applied to obtain each 2nd-gen query

Query Rewriting-based Implementation

This process continues for a bounded number of generations, accumulating the cost of the sequence of edit and relaxation operations applied to obtain each query in the i^{th} generation

The rewriting (represented by function QRA in the next slide) terminates once the cost of all the queries generated in a generation has exceeded the maximum cost set by the user/application

- We make a conservative assumption about the cost of any instances of SIM in the query, that they will contribute 0 cost

SPARQL query evaluation extended with SIM (represented by Eval-SIM in next slide) is applied to each query Q' generated by QRA, returning a set of mappings, to each assigned the cost of any instances of SIM

The resulting set of mapping/cost pairs M is maintained as a priority queue, in order of non-decreasing combined query cost and SIM cost

If a mapping is generated more than once, only the one with the lowest cost is retained in M

Query Rewriting-based Implementation

Algorithm Flexible Query Evaluation

input : query Q; maximum cost max-cost; graph G; ontology K

output: list M of mapping/cost pairs, sorted by non-decreasing cost

M := {}

for each (Q',queryCost) in **QRA**(Q,max-cost,K) do

  foreach (μ,simCost) in **Eval-SIM**(Q',G) do

    M := M U {(μ,queryCost+simCost)}

return M

Ongoing work

Formally show **soundness and completeness of the SPARfL** evaluation algorithm w.r.t. SPARfL language semantics

Performance evaluation of SPARfL:

- complexity analysis shows no increase in complexity classes compared with SPARQL^{AR} but empirical confirmation needed

Optimisation for SPARQL^{AR} query evaluation (subquery pre-computation, graph summarisation, query containment) and, subsequently, for SPARfL

SPARQL^{AR} performance study

10 queries over YAGO – see SWJ 2017 paper for details.
All edit and relaxation operation costs are set to 1. Max-cost is set to 2

Query	Triple patterns	No. of RELAX triple patterns	No. of APPROX triple patterns	Comments
1	1	1		
2	2	1		Relaxed query returns every Event
3	3	1		
4	4	1		
5	5	1	2	Did not finish: Kleene closure, presence of _ in rewritten queries
6	6	1	2	Many query rewritings, presence of _
7	7	2	1	Many empty/already evaluated queries
8	8	1	1	Presence of _ in rewritten queries
9	9	2		Kleene closure
10	10	1	2	Many query rewritings

SPARQL^{AR} performance study

In Table 4, "A/R" is unoptimised execution time; "optimised A/R" utilises pre-computation of single triple patterns and of pairs of triple patterns

Table 2

Numbers of answers (Exact and A/R) and numbers of rewritten queries (A/R).

	Q_1	Q_2	Q_3	Q_4	Q_5
Exact	6491	116	106	8546	585150
A/R	6494	60614	6867	8586	N/A
# of queries	2	5	5	2	95

Table 4

Query execution time (in seconds).

	Q_1	Q_2	Q_3	Q_4	Q_5
Exact	0.321	0.008	0.009	1.512	7670
A/R	0.340	66.32	0.81	1.571	N/A
optimised A/R	0.440	60.4	2.31	1.01	N/A

SPARQL^{AR} performance study

In Table 5, "A/R" is unoptimised execution time; "optimised A/R" utilises pre-computation of single triple patterns and of pairs of triple patterns

Table 3

Numbers of answers (Exact and A/R) and numbers of rewritten queries (A/R).

	Q_6	Q_7	Q_8	Q_9	Q_{10}
Exact	28	5	1540	0	0
A/R	14431	N/A	22540	0	0
# of queries	154	36	17	29	47

Table 5

Query execution time (in seconds).

	Q_6	Q_7	Q_8	Q_9	Q_{10}
Exact	0.123	5	0.173	1.23	323.100
A/R	N/A	N/A	272.875	N/A	N/A
optimised A/R	60.23	N/A	12.475	0.08	100.4

SPARQL^{AR} performance study

The number of rewritten queries depends mostly on the presence of APPROX and on the complexity of the property path it is applied to

Our pre-computation technique is able to reduce query evaluation time for most queries that have a large number of rewritings

Q5 problematic due to presence of both Kleene closure and _

Q7 problematic due to presence of many empty/already evaluated queries

We are therefore exploring three further optimisation approaches:

- **graph summarisation**, to replace _ by a disjunction of specific URIs appearing in the graph, and remove queries that will return no answers
- **query containment**, to reduce the number of queries that need to be evaluated by discarding queries whose answer set is contained in the answer set of another query (at the same cost)
- use of **path indexes** to speed up the evaluation of Kleene closure

Future work

Finer-grained costing of substitution operations and ontology-based relaxations e.g. via syntactic/semantic similarity measures

Design, implementation, evaluation **of visual query interaction and explanation facilities** for end-users

Extending **more expressive languages** with flexible querying, e.g.

- *extended conjunctive regular path queries,* which may contain comparisons between path variables in the query body and path variables in the query head[1]
- queries with *aggregation functions* such as count, sum, max, min in query head[2]
- *larger fragments of SPARQL* e.g. negation, aggregation, federation

1. P. Barcelo, L. Libkin, A. W. Lin, and P. T. Wood. Expressive languages for path queries over graph-structured data. ACM Trans. Database Systems, 37(4), 2012.
2. P. T. Wood. Query languages for graph databases. ACM SIGMOD Record, 41(1), pp 50-60, 2012.