Supporting Users’ Flexible Querying of Knowledge Graphs

Alex Poulouvassilis

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

FQAS 2017, 21-22 June 2017
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

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

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 $\Sigma \cup \{\text{type}\}$

- **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 $\{\text{sc,sp,dom,range}\}$

The model encompasses RDF data, except for blank nodes. Plus a fragment of the RDFS vocabulary: $\text{rdf:type, rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain, rdfs:range (pDF)}$
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 $\text{cl}(I)$.
Ontology-based relaxation of queries

Query evaluation is on the graph obtained by restricting $\text{cl} (G \cup K)$ to edges with labels in $\Sigma \cup \{\text{type}\} \cup \text{propertyNodes}(N_K)$

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

Subgraphs of $K$ induced by edges labelled $\text{sc}$ or $\text{sp}$ 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, $\text{extRed}(K)$, 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 $\text{extRed}(K)$:

(a) compute $\mathit{cl}(K)$;

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

(c) return $\mathit{cl}(K) - D$

\[
\begin{align*}
\text{(e1)} & \quad (b, \text{dom}, c) (a, \text{sp}, b) \quad (a, \text{dom}, c) \\
\text{(e2)} & \quad (b, \text{range}, c) (a, \text{sp}, b) \quad (a, \text{range}, c) \\
\text{(e3)} & \quad (a, \text{dom}, b) (b, \text{sc}, c) \quad (a, \text{dom}, c) \\
\text{(e4)} & \quad (a, \text{range}, b) (b, \text{sc}, c) \quad (a, \text{range}, c)
\end{align*}
\]
Ontology-based relaxation of queries

Triple pattern \((x, p, y)\) \textit{directly relaxes} to triple pattern \((x', p', y')\) w.r.t. ontology \(K = extRed (K)\) if \(\text{vars} (x,p,y) = \text{vars} (x',p',y')\) 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 \textit{relaxation graph} of a triple pattern is the directed acyclic graph induced by the direct triple pattern relaxation relation

Triple pattern \((x, p, y)\) \textit{relaxes} to a triple pattern \((x', p', y')\), written \((x, p, y) \leq (x', p', y')\), if there is a sequence of direct relaxations deriving \((x', p', y')\) from \((x, p, y)\)

- the \textit{relaxation cost} is the minimum cost of such a sequence of direct relaxations
Example relaxation graphs for two triple patterns:

\((?X, \text{hasFamilyName}, ?Y)\) and \((?X, \text{actedIn}, \text{“Jaws”})\)
Ontology-based relaxation of queries

Given a graph pattern \( P_n \) consisting of \( n \) triple patterns, the \textit{graph pattern relaxation relation} \( \leq_n \) is the direct product, \( n \) times, of \( \leq \).

The \textit{direct graph pattern relaxation relation} is the reflexive, transitive reduction of \( \leq_n \).

The \textit{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

\[(?X, \text{hasFamilyName}, ?Y), (?X, \text{actedIn}, \text{“Jaws”})\]

A graph pattern \( P_n \) \textit{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 \textit{relaxation cost} is the minimum cost of such a sequence of direct graph pattern relaxations.
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 $\Sigma \cup \{\text{type}\}$
- 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 $\Sigma U \{\text{type}\}$

A regular expression $R$ over $\Sigma U \{\text{type}\}$ is defined as

$$R := \varepsilon | a | a^- | _ | (R_1R_2) | (R_1|R_2) | R^* | R^+$$

$\varepsilon$ is the empty string, $a$ is any symbol in $\Sigma U \{\text{type}\}$, $|$ denotes the disjunction of all symbols in $\Sigma U \{\text{type}\}$, 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

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

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 $l_1 ... l_n$ is in $\mathcal{L}(R)$, the language recognised by $R$. 
Exact matching of RPQs

Given an RPQ $Q = (x, R, y)$, let $\theta$ 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 $\theta(x)$ to $\theta(y)$ which conforms to $R$

The *exact answer* of $Q$ on $G$ is the set of tuples $\theta(x,y)$ for all such matchings $\theta$
Approximate matching of RPQs

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

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* \( R \), \( \text{edist}(p,R) \), is the minimum edit distance from \( p \) to any semipath conforming to \( R \).

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

The *approximate top-\( k \) answer* of \( Q \) on \( G \) is the list of \( k \) tuples \( \theta(x,y) \) 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\textsuperscript{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 \( \mu \), 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 $\text{APPROX}$ 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 \textit{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 \textit{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 \(O(|S| \cdot |N_K|)\) states and \(O(|S|^2 \cdot |N_K| \cdot |E_K|)\) 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:
   \(O(|S| \cdot |N_K| \cdot |N|)\) states and \(O(|S|^2 \cdot |N_K| \cdot |E_K| \cdot |E|)\) transitions
**SPARQLAR** complexity results

See ODBASE 2014 and SWJ 2017 papers for details

<table>
<thead>
<tr>
<th>Operators</th>
<th>Data Complexity</th>
<th>Query Complexity</th>
<th>Combined Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELAX, APPROX</td>
<td>$O(</td>
<td>E</td>
<td>)$</td>
</tr>
<tr>
<td>AND, FILTER, RegEx</td>
<td>$O(</td>
<td>E</td>
<td>)$</td>
</tr>
<tr>
<td>AND, FILTER, RegEx, RELAX, APPROX</td>
<td>$O(</td>
<td>E</td>
<td>)$</td>
</tr>
<tr>
<td>AND, RegEx, FILTER, SELECT</td>
<td>P-Time</td>
<td>NP-Complete</td>
<td>NP-Complete</td>
</tr>
<tr>
<td>AND, RegEx, FILTER, SELECT, RELAX, APPROX</td>
<td>P-Time</td>
<td>NP-Complete</td>
<td>NP-Complete</td>
</tr>
<tr>
<td>AND, RegEx, FILTER, SELECT, UNION, RELAX, APPROX</td>
<td>P-Time</td>
<td>NP-Complete</td>
<td>NP-Complete</td>
</tr>
</tbody>
</table>
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 \( K = \text{extRed} (K) \) and a graph \( G = \text{closure}_K (G) \), a flex operation is

- either an edit operation on a symbol in \( \Sigma \cup \Sigma^- \)
- or a direct relaxation operation using ontology \( K \)

Given a triple pattern \( Q=(x,P,y) \), matching \( \theta \), semipath \( q \) from \( \theta(x) \) to \( \theta(y) \) 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 \( \theta(Q) \) is the minimum distance from \( p \) to any semipath \( q \) from \( \theta(x) \) to \( \theta(y) \) conforming to \( P \)
- the distance of tuple \( \theta(x,y) \) is the minimum distance to \( \theta(Q) \) from any semipath in \( G \)
- the top-\( k \) answer of \( Q \) on \( G \) is the list of \( k \) tuples \( \theta(x,y) \) 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

\[(?Y, \text{hasFamilyName}^\circ/\text{actedIn}, \text{“Jaws”})\]

and q is the sequence of labels \(\text{hasFamilyName}^\circ \circ \text{actedIn}\)

This has the “triple form”

\[(?Y, \text{hasFamilyName}^\circ, ?X), (?X, \text{actedIn}, \text{“Jaws”})\]

for a fresh variable ?X

This triple form is normalised to

\[(X?, \text{hasFamilyName}, ?Y), (?X, \text{actedIn}, \text{“Jaws”})\]
Approximation of RPQs

Consider RPQ

\[(?Y, \text{hasFamilyName}^{-}/\text{actedIn}, \text{“Jaws”})\]

and the sequence of labels \(\text{hasFamilyName}^{-} \circ \text{actedIn}\)

This can be converted to the triple form

\[(?Y?, \text{hasFamilyName}^{-}, ?X), (?X, \text{actedIn}, \text{“Jaws”})\]

where \(?X\) is a fresh variable
Approximation of RPQs

Starting with \((Y?, \text{hasFamilyName}-, ?X), (?X, \text{actedIn}, "Jaws")\):

*insertion* of *label* may result in
\((Y?, \text{hasFamilyName}-, ?X), (?X, \text{actedIn}, ?Z), (?Z, \text{label} \ "Jaws")\)
(for fresh ?Z)

*deletion* of *hasFamilyName-* may result in
\((?Y, \text{actedIn}, "Jaws")\)

*substitution* of *actedIn* by *directed* may result in
\((Y?, \text{hasFamilyName}-, ?X), (?X, \text{directed}, "Jaws")\)
FLEXing of RPQs – combining relaxation and approximation possibilities (partial)

(\(?X, \text{hasFamilyName}, ?Y\),
(\(?X, \text{actedIn}, ?Z\),
(\(?Z, \text{label,} \text{"Jaws"}\)
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
<table>
<thead>
<tr>
<th>URL</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://yago-knowledge.org/resource/degrees">http://yago-knowledge.org/resource/degrees</a></td>
<td>1381</td>
</tr>
</tbody>
</table>

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

```sql
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 {
  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: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:

<table>
<thead>
<tr>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;Cher&quot;</td>
</tr>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;Cher@eng&quot;</td>
</tr>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;Cher (cantant)@cat&quot;</td>
</tr>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;Cher@deu&quot;</td>
</tr>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;ara@&quot;</td>
</tr>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;Cher@lit&quot;</td>
</tr>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;Cher@ita&quot;</td>
</tr>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;Sera@lav&quot;</td>
</tr>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;Cherilyn Sarkisian&quot;</td>
</tr>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;Cher (spevacka)@slk&quot;</td>
</tr>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;Шор@bel&quot;</td>
</tr>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;Šěra@lav&quot;</td>
</tr>
<tr>
<td><img src="http://yago-knowledge.org/resource/Cher" alt="Image" /></td>
<td>&quot;Artisten_Cher@anno&quot;</td>
</tr>
</tbody>
</table>

Query: 
```
```

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 .
  SIM (?c yago:hasGloss "scientist") }
```

Similarity is in the range [0,1]

The cost of the SIM triple is \( c \times (1 - \text{similarity}) \)

weighting \( c \) can be set by the user or the system

Suppose the user sets \( c \) to 1 and \( \text{max-cost} \) to 1
Top-10 results are:


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) .
    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 .
  SIM (?c yago:hasGloss “scientist”) }
```

obtained through a substitution of `wasBornIn` by `_`
First 22 non-exact results (all at cost 1.2) are:
Suppose the user is interested in retrieving more answers of a similar kind and increases \textit{max-cost} to 2.5

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

\begin{verbatim}
SELECT DISTINCT ?p WHERE {
  SIM (?c yago:hasGloss "scientist") }
\end{verbatim}

obtained through a second substitution, of \texttt{isLocatedIn} by _
First 22 results at cost > 1.5 (all at cost 2.2) are:

| <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_Edmundson> |
| <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/N_B_R_Lickorish> |
| <http://yago-knowledge.org/resource/Joshua_Kingo> |

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 .
  ?c yago:hasGloss ?string .
  SIM (?c yago:hasGloss "scientist")
}
```

obtained by deletion of *wasBornIn*, and relaxation of *isLocatedIn* to *isPlacedIn*
Additional results are:
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\textsuperscript{AR}, subject to the proviso that edit operations are not allowed on type and type

This proviso means that the relaxed-approximate automaton RA\textsubscript{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 $O(nD^N)$.

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\(^A\)R but empirical confirmation needed

**Optimisation** for SPARQL\(^A\)R query evaluation (subquery pre-computation, graph summarisation, query containment) and, subsequently, for SPARfL
**SPARQL\textsuperscript{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

<table>
<thead>
<tr>
<th>Query</th>
<th>Triple patterns</th>
<th>No. of RELAX triple patterns</th>
<th>No. of APPROX triple patterns</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td>Relaxed query returns every Event</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>Did not finish: Kleene closure, presence of _ in rewritten queries</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>Many query rewritings, presence of _</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>Many empty/already evaluated queries</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>Presence of _ in rewritten queries</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>2</td>
<td></td>
<td>Kleene closure</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>Many query rewritings</td>
</tr>
</tbody>
</table>
**SPARQL<sup>AR</sup> 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

<table>
<thead>
<tr>
<th></th>
<th>Q&lt;sub&gt;1&lt;/sub&gt;</th>
<th>Q&lt;sub&gt;2&lt;/sub&gt;</th>
<th>Q&lt;sub&gt;3&lt;/sub&gt;</th>
<th>Q&lt;sub&gt;4&lt;/sub&gt;</th>
<th>Q&lt;sub&gt;5&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>6491</td>
<td>116</td>
<td>106</td>
<td>8546</td>
<td>585150</td>
</tr>
<tr>
<td>A/R</td>
<td>6494</td>
<td>60614</td>
<td>6867</td>
<td>8586</td>
<td>N/A</td>
</tr>
<tr>
<td># of queries</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>95</td>
</tr>
</tbody>
</table>

Table 4

<table>
<thead>
<tr>
<th></th>
<th>Q&lt;sub&gt;1&lt;/sub&gt;</th>
<th>Q&lt;sub&gt;2&lt;/sub&gt;</th>
<th>Q&lt;sub&gt;3&lt;/sub&gt;</th>
<th>Q&lt;sub&gt;4&lt;/sub&gt;</th>
<th>Q&lt;sub&gt;5&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>0.321</td>
<td>0.008</td>
<td>0.009</td>
<td>1.512</td>
<td>7670</td>
</tr>
<tr>
<td>A/R</td>
<td>0.340</td>
<td>66.32</td>
<td>0.81</td>
<td>1.571</td>
<td>N/A</td>
</tr>
<tr>
<td>optimised A/R</td>
<td>0.440</td>
<td>60.4</td>
<td>2.31</td>
<td>1.01</td>
<td>N/A</td>
</tr>
</tbody>
</table>
**SPARQLAR 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>
<thead>
<tr>
<th>Table 3</th>
<th>Numbers of answers (Exact and A/R) and numbers of rewritten queries (A/R).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$Q_6$</td>
</tr>
<tr>
<td>Exact</td>
<td>28</td>
</tr>
<tr>
<td>A/R</td>
<td>14431</td>
</tr>
<tr>
<td># of queries</td>
<td>154</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5</th>
<th>Query execution time (in seconds).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$Q_6$</td>
</tr>
<tr>
<td>Exact</td>
<td>0.123</td>
</tr>
<tr>
<td>A/R</td>
<td>N/A</td>
</tr>
<tr>
<td>optimised A/R</td>
<td>60.23</td>
</tr>
</tbody>
</table>
SPARQL$^\text{AR}$ 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
- queries with *aggregation functions* such as count, sum, max, min in query head
- *larger fragments of SPARQL* e.g. negation, aggregation, federation

---