Flexible Querying of Knowledge Bases

Alex Poulovassilis

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

Leeds University,
October 2015

Outline of the talk

1. Introduction and Motivation
   - Example: data integration
   - Relaxation of conjunctive queries on RDF/S knowledge bases
2. Regular Path Queries and approximate matching
3. Extending SPARQL 1.1 with approximation and relaxation
   - Flexible querying of RDF/S knowledge bases
   - Query rewriting-based implementation
4. User interaction
5. Concluding Remarks

More details can be found in the papers cited. Also in "Applications of Flexible Querying to Graph Data", draft lecture notes from EDBT 2015 Summer School, at www.dcs.bbk.ac.uk/~ap/applicationsFlexGraphQ.pdf

1. Introduction and Motivation

- Increasing volumes of *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

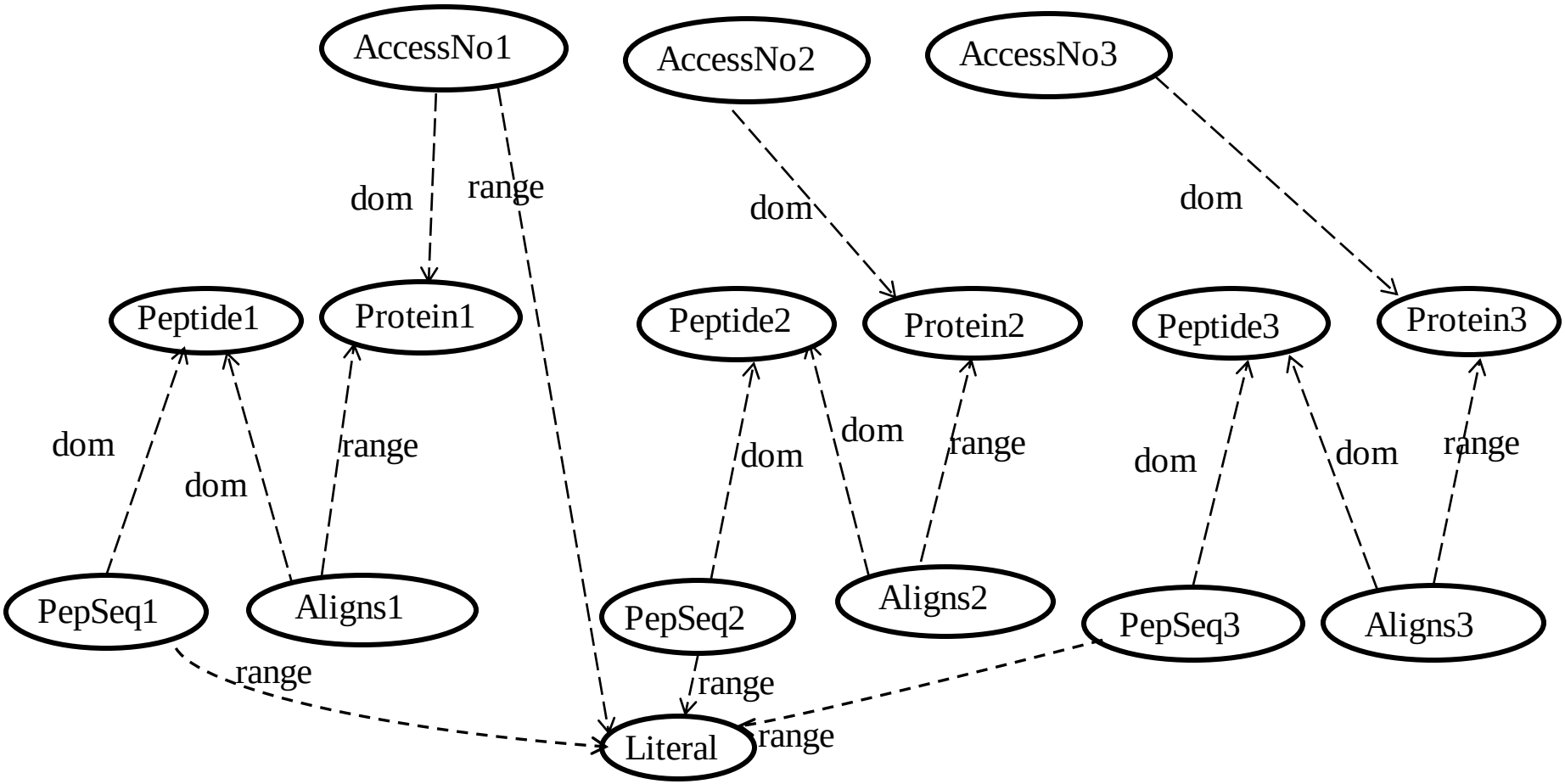
- We look at two kinds of flexible querying:
  - query *relaxation* – returns *additional* answers compared to the exact form of the query
  - query *approximation* – returns *different* answers compared to the exact form of the query
- as well as their combination

Example: data integration

- Much work has been done in developing architectures and methodologies for biological data integration
- Beneficial for scientists by providing them with easy access to more data, leading to new analyses and new scientific insights
- Traditional data integration methodologies need semantic mappings between the different data sources to be determined "up front", requiring early commitment of resources
- Recent research has focussed on incremental, "pay-as-you-go" data integration approaches supporting *incremental refinement* of the global schema or ontology as time and resources allow

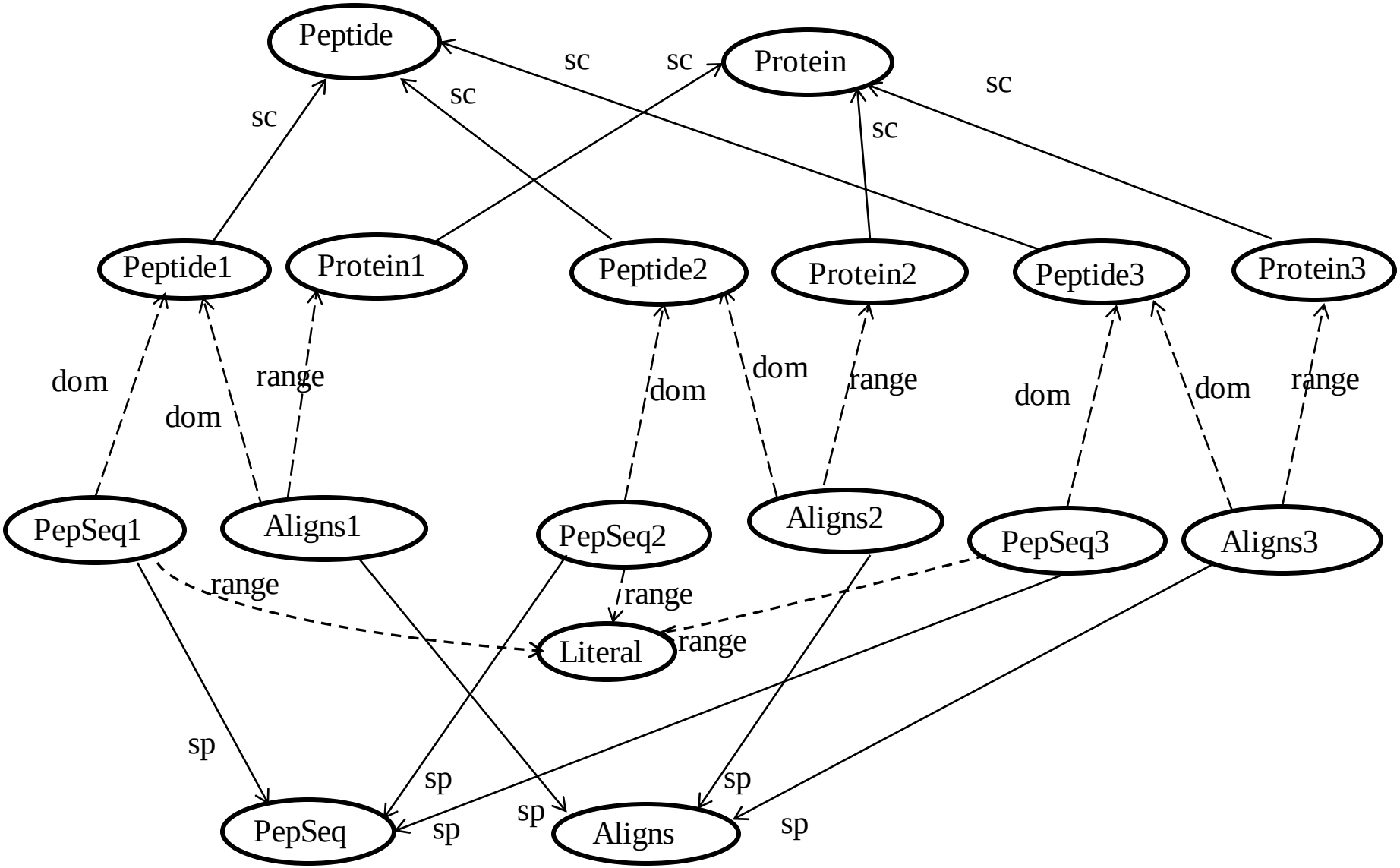
Example: data integration

- Suppose an ontology includes
  - classes Peptide1, Protein1, Peptide2, Protein2, Peptide3, Protein3 arising from the integration of three different source databases, DB1, DB2, DB3
  - properties
    - PepSeq*i*, $1 \leq i \leq 3$, each with domain Peptide*i* and range Literal
    - Aligns*i*, $1 \leq i \leq 3$, each with domain Peptide*i* and range Protein*i*
    - AccessNo*i*, $1 \leq i \leq 3$, each with domain Protein*i* and range Literal


AccessNo1
AccessNo2
AccessNo3
dom
range
dom
dom
Peptide1
Protein1
Peptide2
Protein2
Peptide3
Protein3
dom
dom
range
dom
dom
range
dom
dom
range
PepSeq1
Aligns1
PepSeq2
Aligns2
PepSeq3
Aligns3
range
range
range
Literal

Example: data integration

- A Data Integrator observes semantic alignments between these classes/properties and adds additional super-classes/properties to integrate the underlying data extents from the 3 databases:
  - superclass Peptide of the classes Peptide*i*
  - superclass Protein of the classes Protein*i*
  - superproperty PepSeq of the properties PepSeq*i*, with domain Peptide and range Literal
  - superproperty Aligns of the properties Aligns*i*, with domain Peptide and range Protein
  - superproperty AccessNo of the properties AccessNo*i*, with domain Protein and range Literal
- A fragment of this global ontology is shown in the next figure (omitting AccessNo*i* and AccessNo properties, and domain and range information of PepSeq and Aligns)


Peptide
Protein
sc
sc
sc
sc
sc
sc
Peptide1
Protein1
Peptide2
Protein2
Peptide3
Protein3
dom
dom
range
dom
dom
range
dom
dom
range
PepSeq1
Aligns1
PepSeq2
Aligns2
PepSeq3
Aligns3
range
range
range
Literal
sp
sp
sp
sp
sp
sp
PepSeq
Aligns

Example: data integration

- Suppose a user only familiar with DB1 poses the query

```
?Y, ?Z <- RELAX (?X,PepSeq1,"ATLITFLCDR"),
          RELAX (?X,Aligns1,?Y),
          RELAX (?Y,AccessNo1,?Z)
```

- The syntax is that of a *conjunctive query* comprising
  - one or more *triple patterns* on its RHS
  - zero or more variables on its LHS - must also appear in the RHS

- Also allows a relaxation operator **RELAX** to be applied to triple patterns that the user would like to be matched flexibly

Example: data integration

- In its non-relaxed form, the query returns identifiers and accession numbers of proteins identified in DB1 through experiments yielding the peptide sequence "ATLITFLCDR"
- A first level of relaxation of all three triple patterns results in the following query:

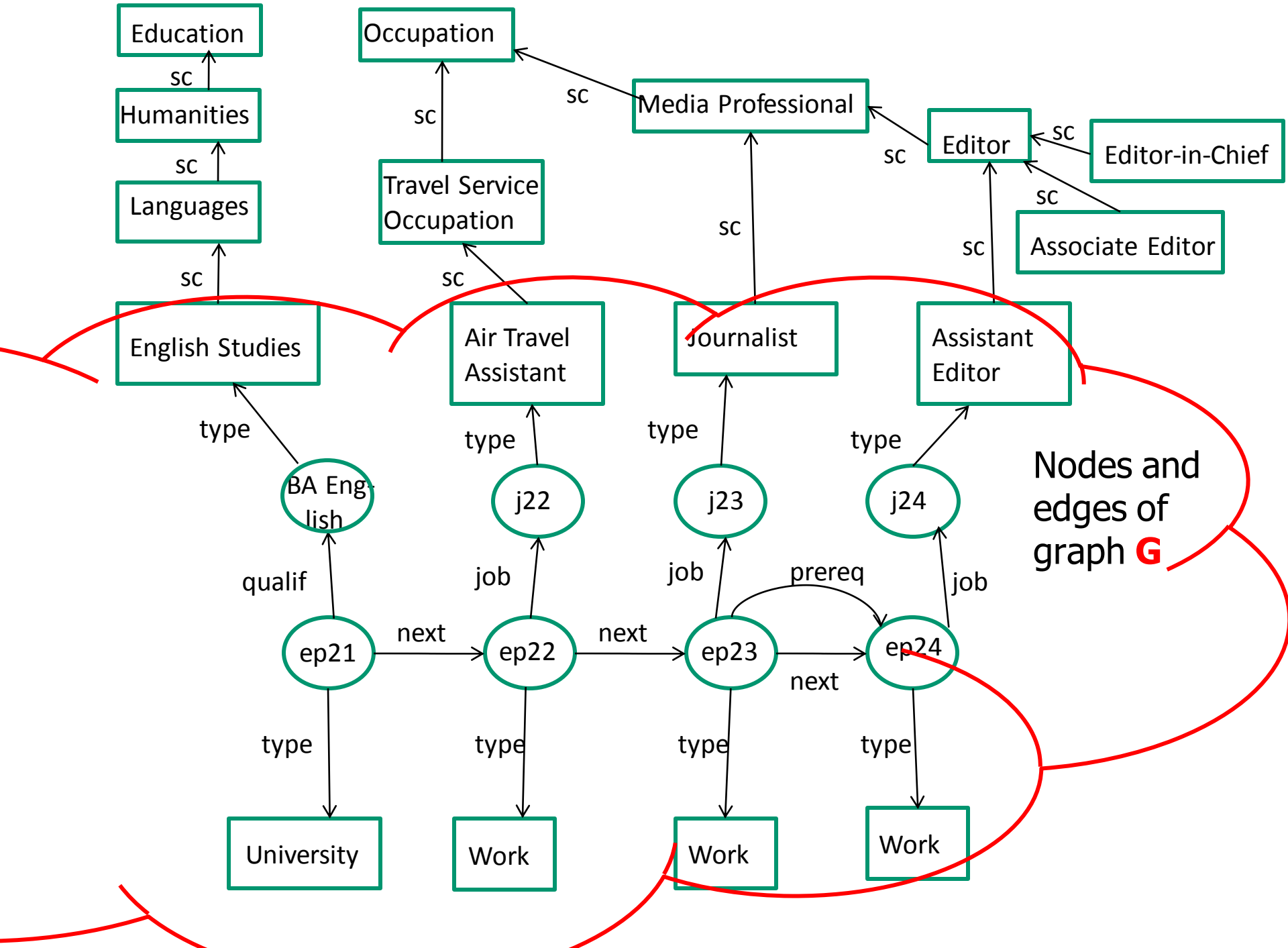
```
?Y, ?Z <- RELAX (?X,PepSeq,"ATLITFLCDR"),
          RELAX (?X,Aligns,?Y),
          RELAX (?Y,AccessNo,?Z)
```

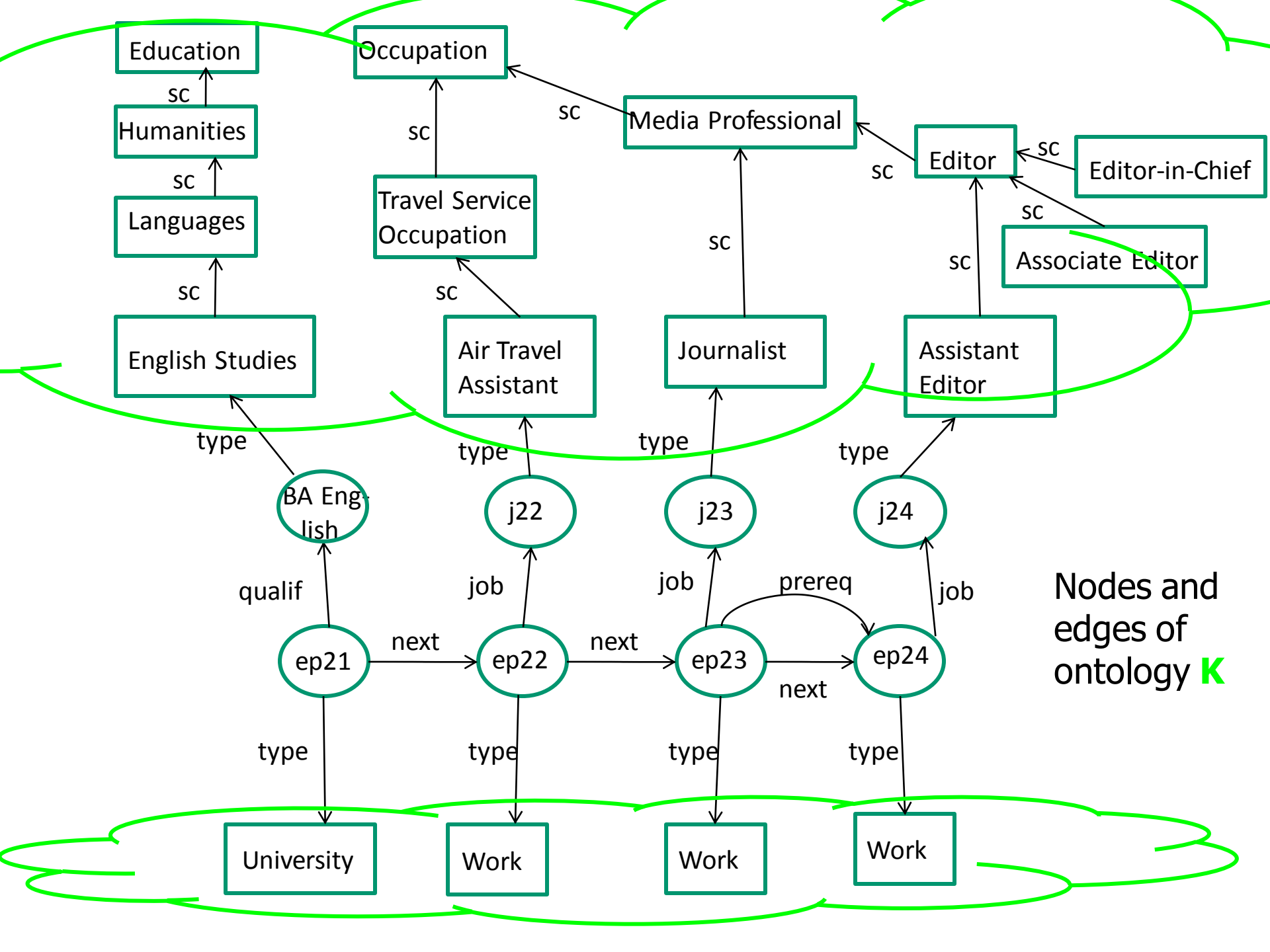
- Evaluation of this automatically expands the result set to include similar results also from DB2 and DB3, *without the user needing to have detailed knowledge of their schemas*

Relaxation of Conjunctive Queries on RDF/S Knowledge Bases

From C.A.Hurtado, A. Poulovassilis and 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 $\Sigma \cup \{type\}$
  - 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


Education
sc
Humanities
sc
Languages
sc
English Studies
Occupation
sc
Travel Service Occupation
sc
Air Travel Assistant
Media Professional
sc
Journalist
sc
Editor
sc
Editor-in-Chief
sc
Associate Editor
sc
Assistant Editor
type
BA Eng-lish
type
j22
type
j23
type
j24
qualif
job
job
prereq
job
ep21
next
ep22
next
ep23
ep24
next
type
type
type
type
University
Work
Work
Work
Nodes and edges of graph G


Education
Occupation
Humanities
Media Professional
Languages
Travel Service Occupation
Editor
Editor-in-Chief
Associate Editor
English Studies
Air Travel Assistant
Journalist
Assistant Editor
BA Eng-lish
j22
j23
j24
ep21
ep22
ep23
ep24
University
Work
Work
Work
sc
type
qualif
job
next
prereq
Nodes and edges of ontology K

Relaxation of Conjunctive 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) $\dfrac{(a,sp,b)\ (b,sp,c)}{(a,sp,c)}$ (2) $\dfrac{(a,sp,b)\ (x,a,y)}{(x,b,y)}$

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

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

Relaxation of Conjunctive Queries

- The *closure* of an RDF/S graph I under these rules is denoted by *cl* (I)
- Query evaluation takes place on the restriction of *cl* (I) to edges with labels in ∑ U {type} U propertyNodes(N_K)
- Subgraphs of K induced by edges labelled sc or sp need to be acyclic:
  - allows an unambiguous cost to be assigned to a relaxed query
- Also, K must be *equal to its extended reduction*, *extRed* (K)
  - allows *direct relaxations* corresponding to the "smallest" possible relaxation steps to be unambiguously applied to queries
  - this in turn allows query answers to be returned to users incrementally, in order of increasing cost

Relaxation of Conjunctive Queries

To compute *extRed* (K):

(a) compute *cl* (K);

(b) apply rules (e1)-(e4) below in reverse till no more rules can be applied;

(c) apply RDFS inference rules (1) and (3) in reverse till no more rules can be applied.

(e1) $\dfrac{(b,dom,c)\ (a,sp,b)}{(a,dom,c)}$ (e2) $\dfrac{(b,range,c)\ (a,sp,b)}{(a,range,c)}$

(e3) $\dfrac{(a,dom,b)\ (b,sc,c)}{(a,dom,c)}$ (e4) $\dfrac{(a,range,b)\ (b,sc,c)}{(a,range,c)}$

Relaxation of Conjunctive 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

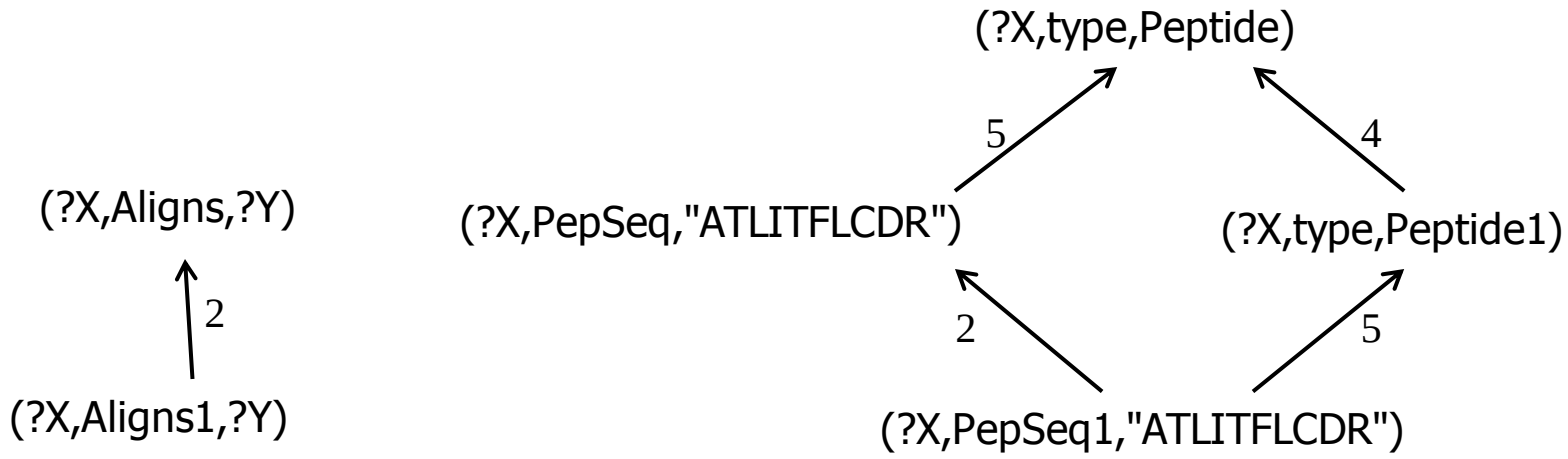
- Triple pattern (x, p, y) *relaxes* to a triple pattern (x′, p′, y′), written (x, p, y) ≤ (x′, p′, y′), if there is a sequence of direct relaxations deriving (x′, p′, y′) from (x, p, y)

- The *relaxation distance* of (x′, p′, y′) from (x, p, y) is the minimum cost of such a sequence of direct relaxations

Relaxation graphs of triple patterns

(?X,Aligns1,?Y) and (?X,PepSeq1,"ATLITFLCDR")

assuming that K is the proteomics ontology earlier:

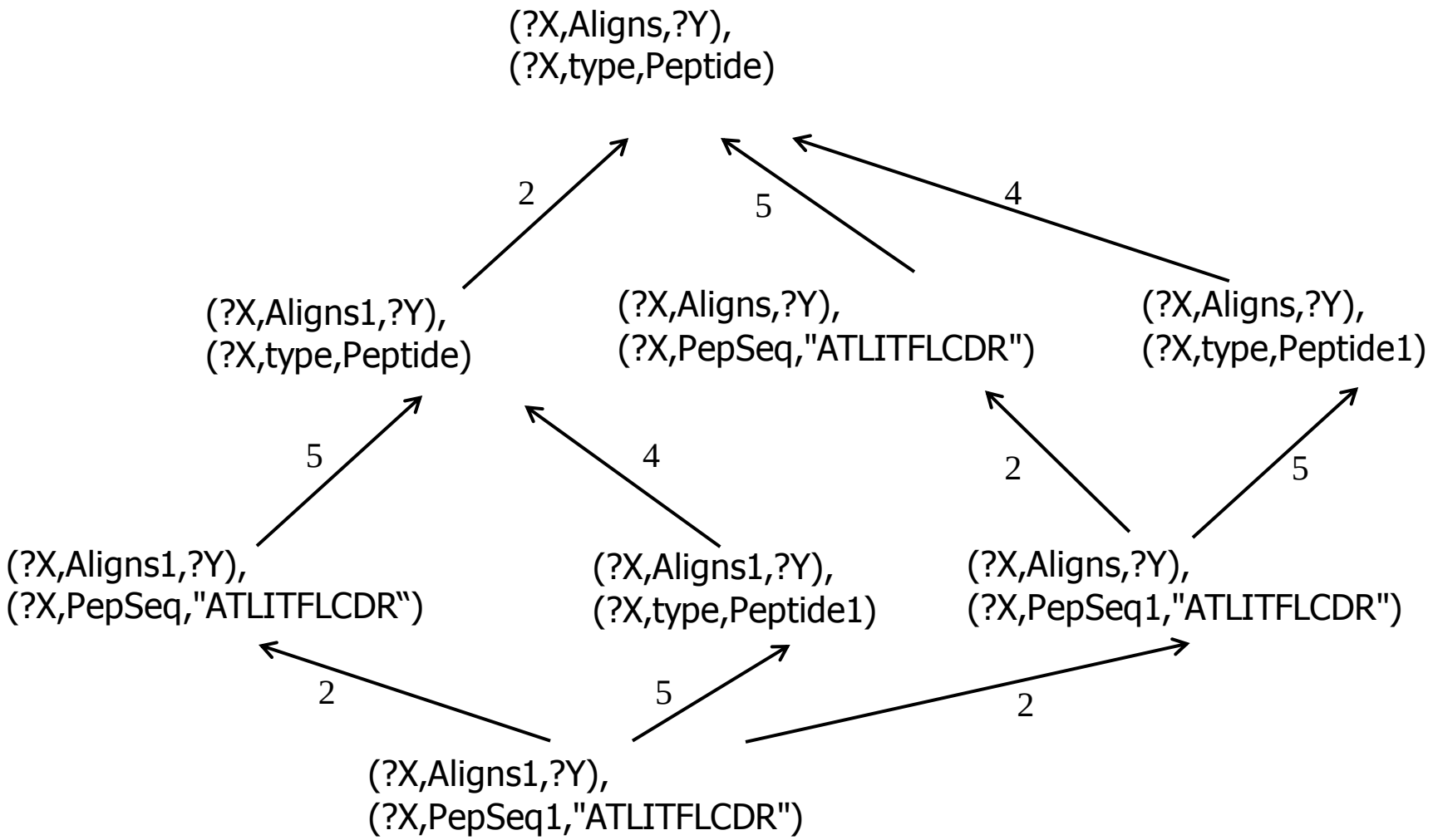


Relaxation of Conjunctive Queries

- 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 page shows the relaxation graph of graph pattern

(?X,Aligns1,?Y), (?X,PepSeq1,"ATLITFLCDR")

- 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 distance* of P'_n from P_n is the minimum cost of such a sequence of direct graph pattern relaxations


(?X,Aligns,?Y),
(?X,type,Peptide)
2
5
4
(?X,Aligns1,?Y),
(?X,type,Peptide)
(?X,Aligns,?Y),
(?X,PepSeq,"ATLITFLCDR")
(?X,Aligns,?Y),
(?X,type,Peptide1)
5
4
2
5
(?X,Aligns1,?Y),
(?X,PepSeq,"ATLITFLCDR")
(?X,Aligns1,?Y),
(?X,type,Peptide1)
(?X,Aligns,?Y),
(?X,PepSeq1,"ATLITFLCDR")
2
5
2
(?X,Aligns1,?Y),
(?X,PepSeq1,"ATLITFLCDR")

2. Regular Path Queries and Approximate Matching

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

- *Regular path queries* 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 label l from a finite alphabet Σ 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

  vars ← (X, R, Y)

  vars is the subset of {X,Y} that are variables, R is a regular expression over Σ U {type}

- A *regular expression* R *over* Σ 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 Σ U {type}, _ denotes the disjunction of all symbols in Σ U {type}, the operators have their usual meaning

Exact matching of RPQs

- A *semipath* p in a graph G from node v to node w is a sequence

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

  such that v_1=v, v_{n+1}=w, 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 string $l_1 \ldots l_n$ is in $\mathcal{L}$(R), the language recognised by R

Exact matching of RPQs

- Given an RPQ Q:

  vars ← (X, R, Y)

- let θ be a matching from {X, Y} to the nodes of graph G, that maps each constant to itself and such that 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(vars) for all such matchings θ

Approximate matching of RPQs

- We allow the following edit operations on semipaths:
  - *insertions*, *deletions* and *substitutions* of labels
  - *inversion* of a label (i.e. reverse edge traversal) and *transposition* of a pair of labels are handled as special cases of substitution
- The application of each edit operation has a 'cost' associated with it
- We envisage the user specifying which edit operations the query system should apply when answering a particular query
- The user can specify the cost associated with each edit operation; or these can be system-defined

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

Approximate matching of RPQs

- The *edit distance* of a semipath p to a regular expression R

  edist (p,R)

  is the minimum edit distance from p to any semipath conforming to R

- For graph G, query Q, matching θ:
  - tuple θ(vars) has *edit distance* *edist* (p,R) to Q if 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) in G
  - note, if p conforms to R, then θ(vars) has edit distance 0 to Q

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

3. SPARQL^{AR} : extending SPARQL with Approximation and Relaxation

From A. Cali, R. Frosini, A. Poulovassilis and P. T. Wood. Flexible querying for SPARQL. ODBASE 2014

- We are investigating 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 does not support notions of query approximation or relaxation (except for OPTIONAL)
- Our ODBASE 2014 paper introduced **APPROX** and **RELAX** operators for property path queries: we term our language **SPARQL^{AR}**
- Showed in ODBASE 2014 that this does not increase the complexity classes of the SPARQL 1.1 fragments studied

3.1 Flexible Querying of YAGO

Example 1

Suppose the user wants to find the geographic coordinates of the "Battle of Waterloo" event by posing this query on YAGO:

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

Returns no answers since YAGO does not store the geographic coordinates of Waterloo

Example: Flexible Querying of YAGO

- The user may choose to approximate the query triple pattern:

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

- The system can now apply an edit operation that deletes happenedIn from the property path. YAGO does store coordinates of the "Battle of Waterloo" event. Resulting query returns the desired answers:

```
"4.4"^^<degrees>
"50.683333333333"^^<degrees>
```

Example: Flexible Querying of YAGO

Example 2

Consider the following portion of the YAGO ontology:

Nodes: hasFamilyName, hasGivenName, label, actedIn, Actor

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

Suppose the user is looking for the family names of actors who played in the film "Tea with Mussolini" :

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

Example: Flexible Querying of YAGO

- Query returns only 4 answers: some actors have only a first name (e.g. Cher); others have their full name recorded using label

- The user may choose to relax the second triple pattern, in an attempt to retrieve more answers:

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

- System can now replace hasFamilyName by label. Resulting query now returns also names recorded through hasGivenName and label properties (255 answers)

Example: Flexible Querying of YAGO

Example 3

Suppose user wishes to find events taking place in Berkshire in 1643 and poses the following query on YAGO (Event is used for simplicity here but the actual URI is <wordnet_event_100029378>):

```
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
    ?x rdf:type Event .
    ?x yago:on "1643-##-##" .
    ?x yago:in "Berkshire" }
```

Returns no results since no property edges labelled on or in in YAGO

Example: Flexible Querying of YAGO

- User may choose to approximate second and third triple patterns:

```
SELECT * WHERE {
    ?x rdf:type Event .
    APPROX ( ?x yago:on "1643-##-##" ) .
    APPROX ( ?x yago:in "Berkshire" ) }
```

- System can now substitute on by happenedOnDate (which does appear in YAGO) and in by happenedIn. However, the query still returns no answers, since happenedIn does not connect event instances directly to literals such as "Berkshire"

Example: Flexible Querying of YAGO

- User may choose to stop any further query approximation, and to relax the third triple pattern of the current query:

```
SELECT * WHERE {
    ?x rdf:type Event .
    ?x yago:happenedOnDate "1643-##-##“ .
    RELAX ( ?x yago:happenedIn "Berkshire" )}
```

- System can now replace the third triple pattern by ?x rdf:type Event using knowledge in YAGO that the domain of happenedIn is Event

- Resulting query returns all events occurring in 1643, including “Siege of Reading” that happened in 1643 in Berkshire; but also several events that did not happen in Berkshire

Example: Flexible Querying of YAGO

- User may now choose to approximate further the earlier third triple pattern, rather than relaxing it:

```
SELECT * WHERE {
    ?x rdf:type Event .
    ?x yago:happenedOnDate "1643-##-##“ .
    APPROX ( ?x yago:happenedIn "Berkshire" )}
```

- System can now insert, after happenedIn the property label that connects URIs to their labels

- Resulting query now returns the only event recorded as occurring in 1643 in Berkshire i.e. the “Siege of Reading”

3.2 Semantics of SPARQL^{AR}

- For specifying the formal semantics of SPARQL^{AR} we extend SPARQL query evaluation, which returns a set of *mappings*. A mapping is a partial function

  $\mu : ULV \rightarrow UL$

  s.t. $\mu(x)=x$ for all x in UL; where U, L, V are pairwise disjoint sets of URIs, literals and variables

- SPARQL^{AR} query evaluation returns a set of *mapping/cost pairs*

  (μ,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 (see ODBASE 2014 paper for details)

3.3 Query Rewriting-based Implementation Approach

- We adopt a *query rewriting approach* whereby a SPARQL^{AR} 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 approximated/relaxed triple pattern (x_i, R_i, y_i) in Q and each URI p appearing in R_i, a set of new queries is constructed from Q_0 by applying all possible one-step edit/relaxation operations to p: these are the "1st-generation" queries
  - To each 1st-gen 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/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 Approach

- Process continues for a bounded number of generations, accumulating the cost of the sequence of edit/relaxation operations applied to obtain each query in the i^{th} generation
- The rewriting (represented by function **QRA** in the next algorithm) terminates once the cost of all the queries generated in a generation has exceeded a maximum value m

- Ordinary SPARQL query evaluation (represented by function **Eval** in next algorithm) is applied to each query generated by **QRA**, returning a mapping to which is assigned the cost of that query
- The resulting set of mapping/cost pairs M is maintained in order of non-decreasing cost, as a priority queue.
- If a mapping is generated more than once, only the one with the lowest cost is retained in M

Query Rewriting-based Implementation Approach

Algorithm **Flexible Query Evaluation**

input : query Q; approx/relax maximum cost m; graph G; ontology K

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

```
M := {}
for each (Q',cost) in QRA(Q,m,K) do
   foreach (μ,0) in Eval(Q',G) do
      M := M U {(μ,c)}
return M
```

Extended version of the ODBASE 2014 paper (to appear in SWJ) gives formal proofs of the soundness and completeness of SPARQL^{AR} query evaluation algorithms w.r.t. the language semantics

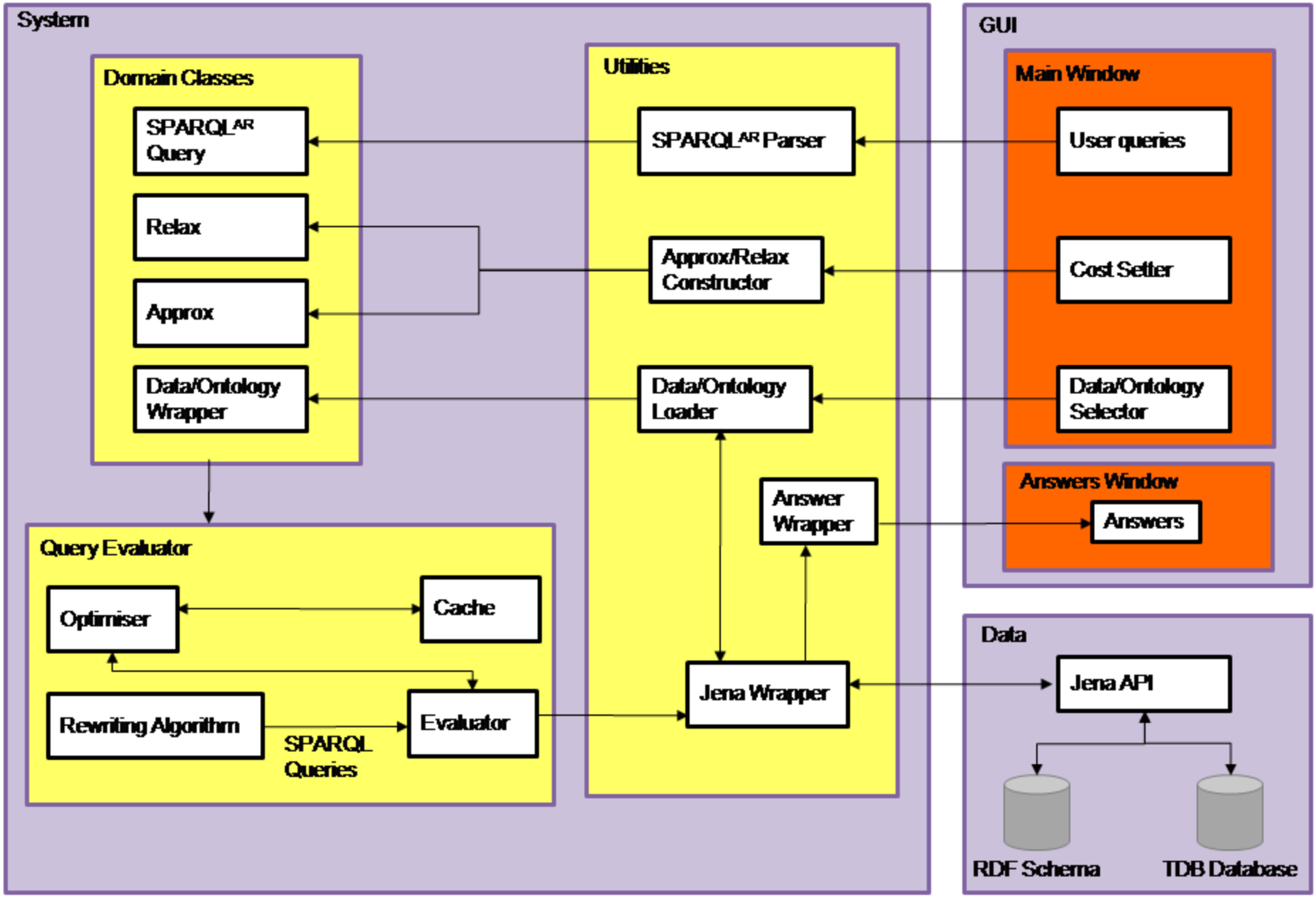
3.4 System Architecture

- Prototype implementation of SPARQL^{AR} is in Java and uses Jena for SPARQL query evaluation. Comprises three layers:

  - **GUI** supports user interaction with system, allowing queries to be submitted, costs of edit and relaxation operators to be set, data sets and ontologies to be selected, query answers to be displayed

  - **System** is responsible for processing of SPARQL^{AR} queries. Provides classes relating to construction, rewriting, optimisation and evaluation of queries

  - **Data layer** connects to the selected RDF dataset and ontology using the JENA API. Jena library methods are used to execute SPARQL queries over the RDF dataset and to load the ontology into memory. RDF datasets are stored as a TDB database


System
Domain Classes
SPARQL^AR Query
Relax
Approx
Data/Ontology Wrapper
Utilities
SPARQL^AR Parser
Approx/Relax Constructor
Data/Ontology Loader
Answer Wrapper
Query Evaluator
Optimiser
Cache
Rewriting Algorithm
SPARQL Queries
Evaluator
Jena Wrapper
GUI
Main Window
User queries
Cost Setter
Data/Ontology Selector
Answers Window
Answers
Data
Jena API
RDF Schema
TDB Database

Performance evaluation

- Our ODBASE 2014 paper reports on performance study using data generated from the Lehigh University Benchmark
- Larger-scale performance study using YAGO is described in the forthcoming SWJ paper
- Results show that SPARQL^{AR} query evaluation using a query rewriting approach is promising
- Difference between execution time of exact form and APPROX/RELAX forms of queries is acceptable for queries with fewer than 5 conjuncts
- For most other queries trialled, a simple caching technique brings down the run times of APPROX/RELAX forms to reasonable levels
- For more complex queries - e.g. involving combinations of Kleene closure * and the wildcard symbol _ within a property path - more sophisticated optimisation techniques are needed

3.5 User Interaction

- An area needing more work is how such a querying system might provide *explanations* to the user of how the 'cost' of each result has been derived from a sequence of edits/relaxations

- Also, allowing the user to *interactively control* and *visualise* how queries are incrementally generated, what cost is associated with a query, and what results are returned, would help user to decide if answers being returned are useful and to try out different edits/relaxations

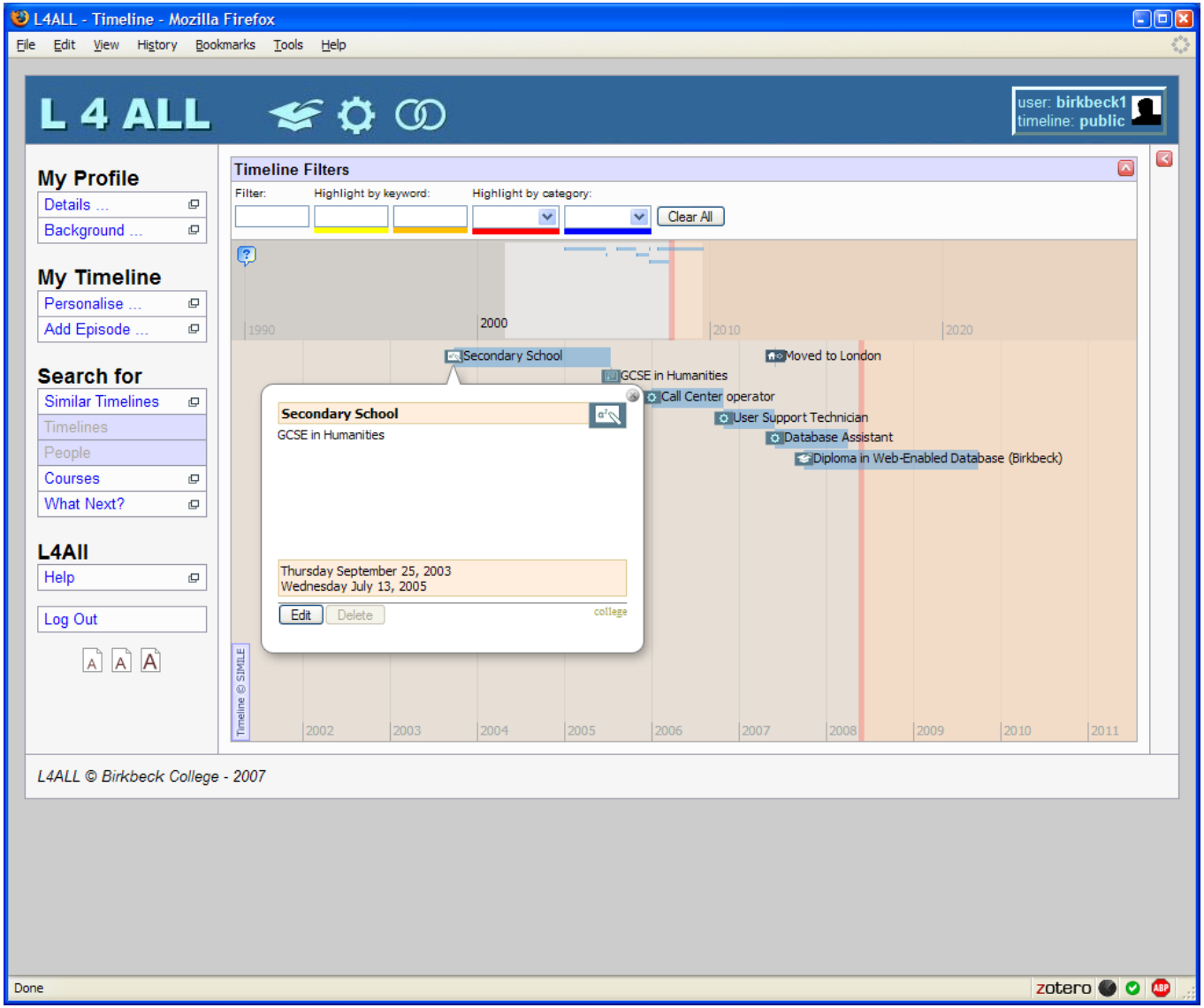
- We have done some initial work in this direction, in the area of flexible querying of lifelong learners' metadata

- Design, implementation, evaluation of interactive flexible querying facilities and visualisations for end-users in a more general setting, are an area requiring further work

Example: Lifelong Learning Networks

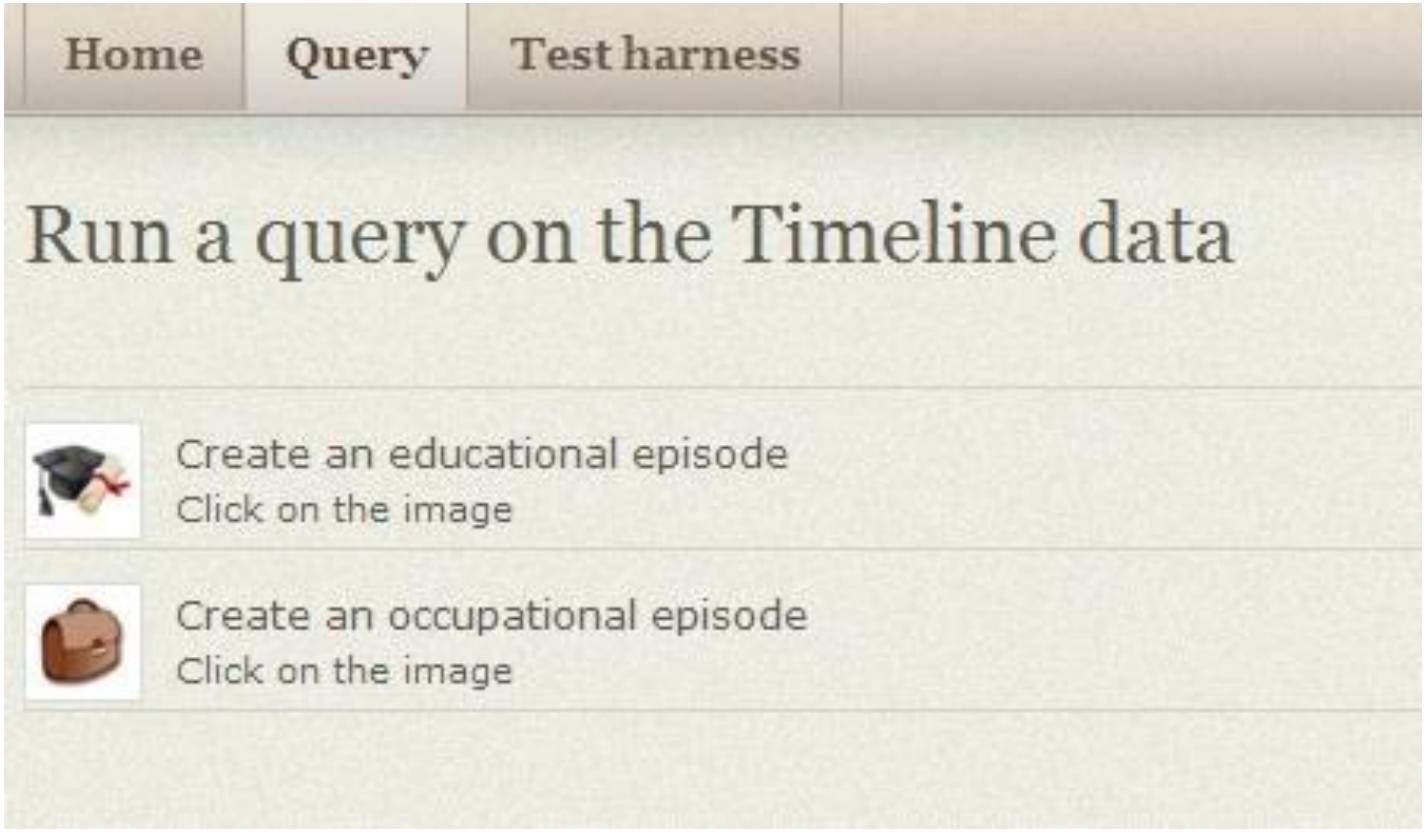
- The L4All system developed in JISC-funded research at the London Knowledge Lab allows users to maintain a chronological record of their episodes of learning and work: their personal 'timelines'
  - c.f. graph **G** and ontology **K** illustrated earlier
- The aim was for users to be able to search the timeline data of others, and identify possible choices for their own future learning and professional development by seeing what others with a similar background have gone on to do

Main L4All timeline construction & search screen



ApproxRelax Prototype GUI

From A. Poulovassilis, P.Selmer and P. T. Wood. Flexible Querying of Lifelong Learner Metadata. IEEE Trans. on Learning Technologies, 5(2), pp 117-129, 2012



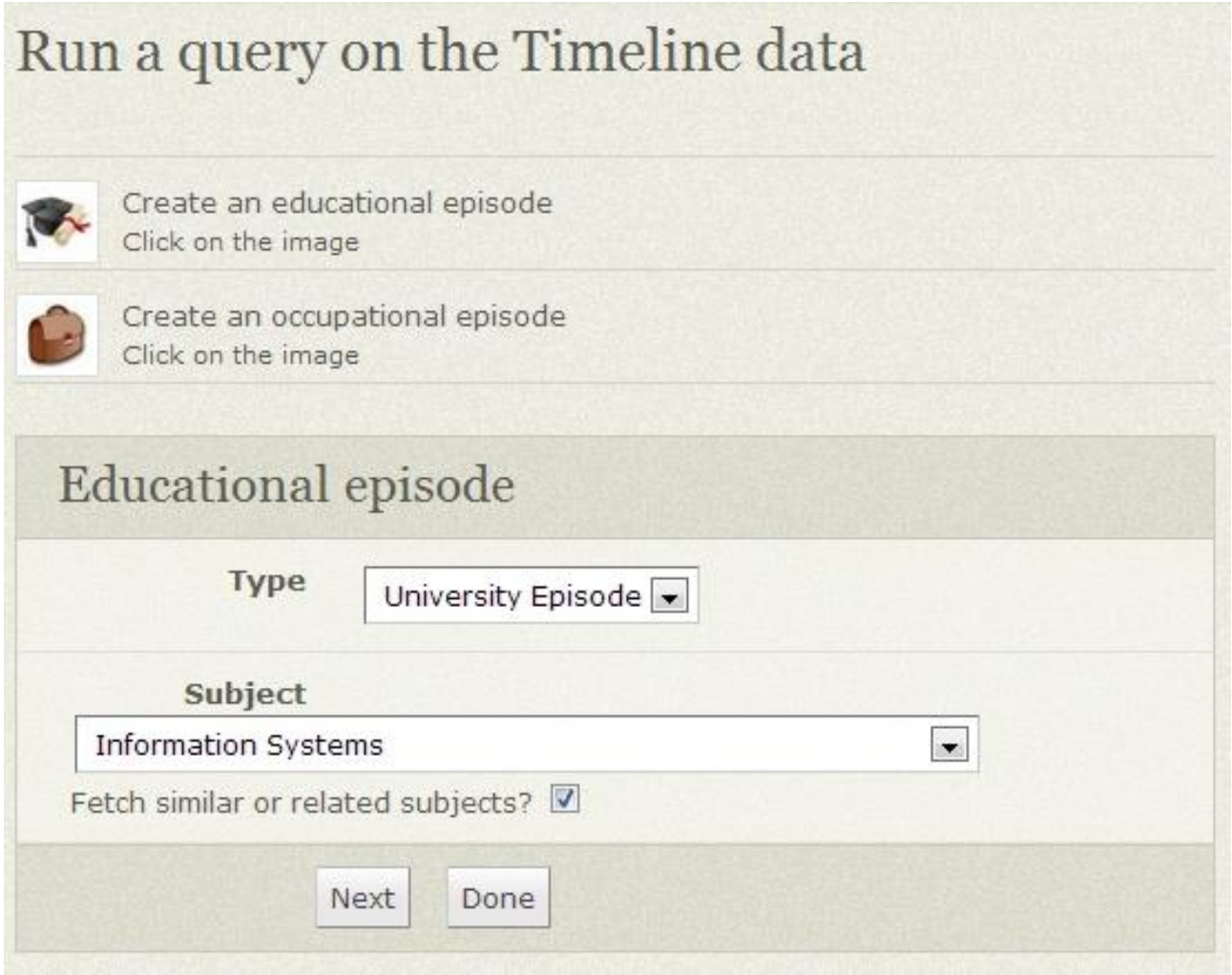
Example Usage Scenario

- Suppose Gaby is studying on a Foundation Degree in Information Technology and wishes to find out what possible future career choices there may be for her by seeing what other people with qualifications in Information Systems have gone on to do in their careers

- She is interested in jobs categorized under Software Professionals and similar

- She also decides to broaden her search by allowing matching of qualifications that are similar to Information Systems

Creating a query

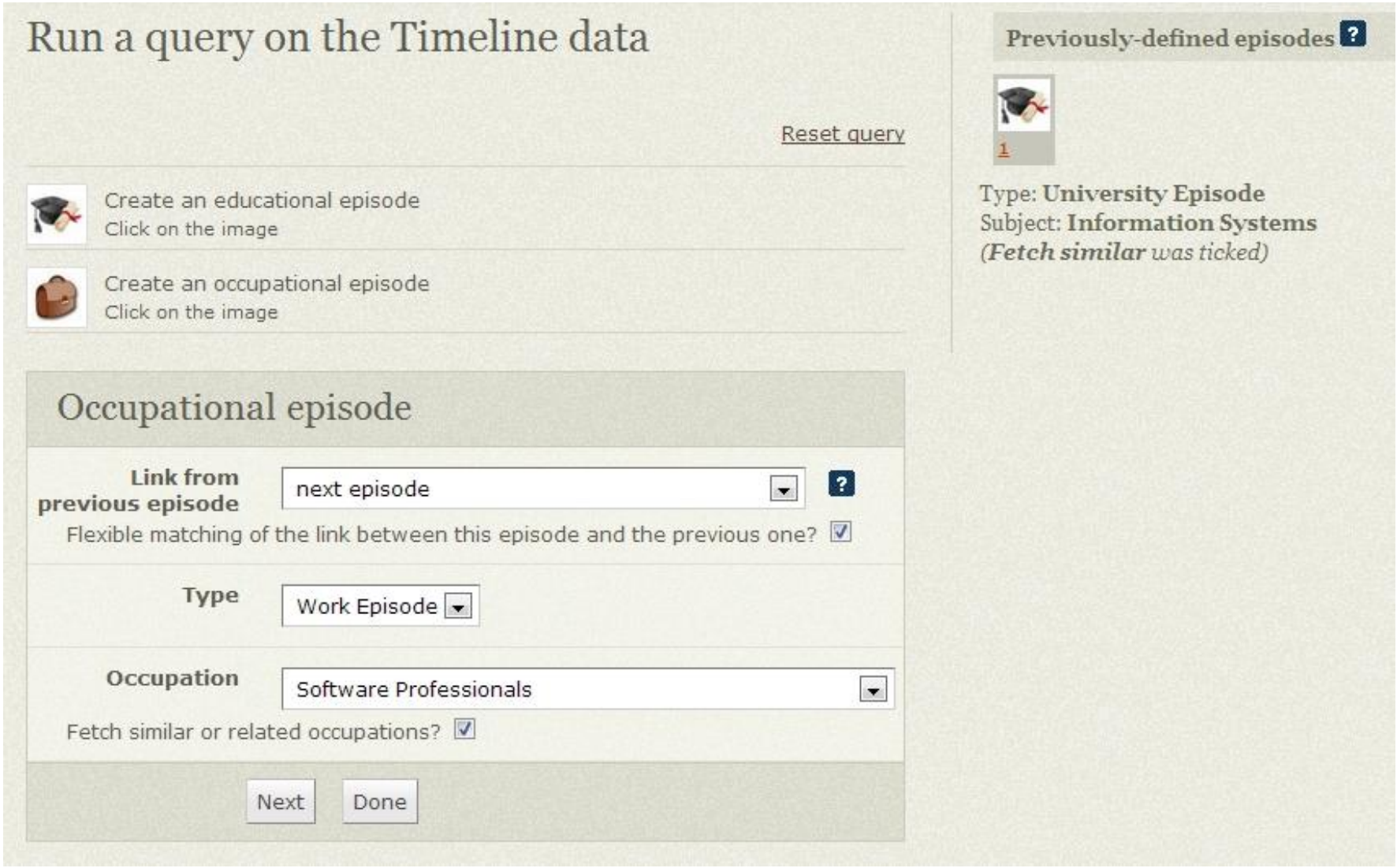


Creating a query

- At this point, the system generates internally this initial query (a conjunction of two RPQs):

  ?A ← (?A, type, UniversityEpisode),
  RELAX (?A, qualif.type, InformationSystems)

Creating a query



Creating a query

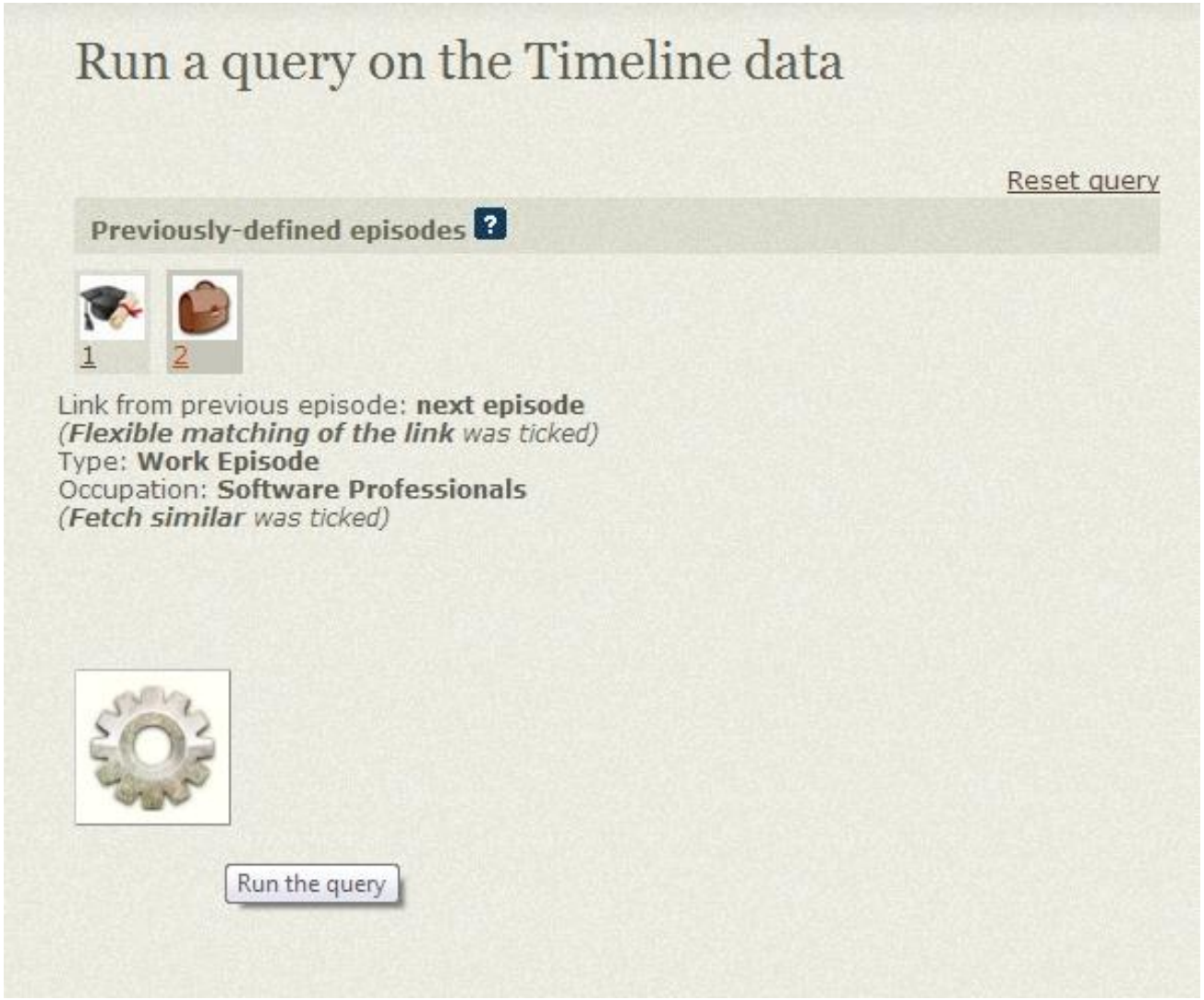
- At this point, the system extends the earlier partial query with the following additional conjuncts:

APPROX (?A, next, ?B) ,
(?B, type, WorkEpisode) ,
RELAX (?B, job.type, SoftwareProfessionals)

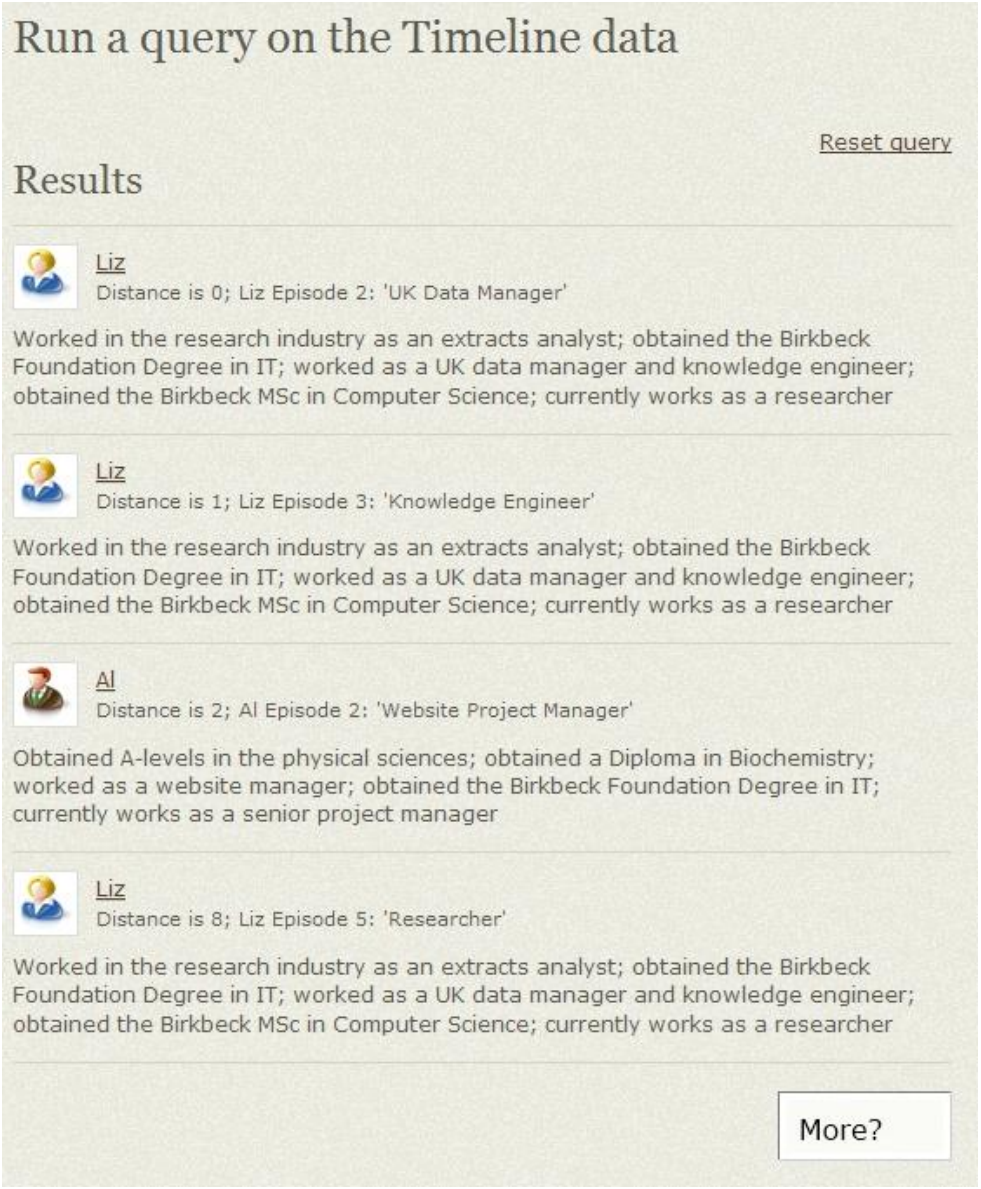
- Resulting in the final overall query:

?A, ?B ← (?A, type, UniversityEpisode) ,
RELAX (?A, qualif.type, InformationSystems) ,
APPROX (?A, next, ?B) ,
(?B, type, WorkEpisode) ,
RELAX (?B, job.type, SoftwareProfessionals)

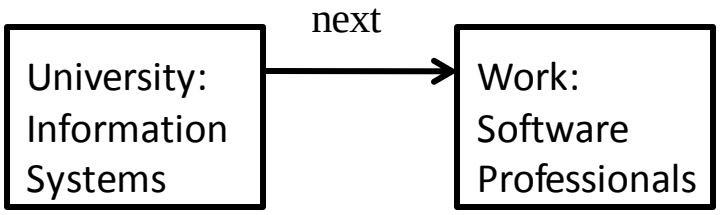
Viewing your query



Viewing query results



Viewing query and selected timeline together



5. Concluding Remarks

- Have given an overview of motivation, theoretical foundations, and implementation approach for extending SPARQL 1.1 with query relaxation and approximation
- Directions of ongoing work:
  - logical and physical optimisations
  - more extensive performance studies
- Future work includes:
  - identifying new Use Cases for query relaxation and approximation over graph-structured knowledge bases
  - investigating complexity implications of extending more expressive query languages with relaxation/approximation (e.g. languages incorporating path variables and aggregation functions)
  - designing user interfaces that allow users to control and visualise how flexible queries are incrementally generated and executed

SPARQL^{AR} complexity results

From ODBASE 2014 paper. Plus additional results from R. Frosini, A. Cali, A. Poulovassilis and P. T. Wood. Flexible query processing for SPARQL. To appear in the Semantic Web Journal, 2015.

Table 1
Complexity of various SPARQL^{AR} fragments.

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