Chapter 4
Applications of Flexible Querying to Graph Data

Alexandra Poulovassilis

Abstract Graph data models provide flexibility and extensibility, which makes them well-suited to modelling data that may be irregular, complex, and evolving in structure and content. However, a consequence of this is that users may not be familiar with the full structure of the data, which itself may be changing over time, making it hard for users to formulate queries that precisely match the data graph and meet their information-seeking requirements. There is a need, therefore, for flexible querying systems over graph data that can automatically make changes to the user's query so as to find additional or different answers, and so help the user to retrieve information of relevance to them. This chapter describes recent work in this area, looking at a variety of graph query languages, applications, flexible querying techniques and implementations.

4.1 Introduction

Due to their fine modelling granularity (in its simplest form, comprising just nodes and edges, naturally representing entities and relationships), graph data models provide flexibility and extensibility, which makes them well-suited for modelling complex, dynamically evolving datasets. Moreover, graph data models are typically semi-structured: there may not be a schema associated with the data; if there is a schema, then aspects of it may be missing from parts of the data and, conversely, parts of the data may not correspond to the schema. This makes graph data models well-suited to modelling heterogeneous and irregular datasets. Graph data models place a greater focus on the relationships between entities than other approaches to data modelling, viewing relationships as important as the entities themselves.

In recent years there has been a resurgence of academic and industry interest in graph databases, due to the generation of large volumes of data from web-based,

A. Poulovassilis (✉)
Birkbeck, University of London, London, UK
e-mail: ap@dcs.bbk.ac.uk

G. Fletcher et al. (eds.), *Graph Data Management*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-319-96193-4_4

mobile and pervasive applications centred on the relationships between entities, for example: the web graph itself; RDF Linked Data[1]; social and collaboration networks[2] (Martin et al. 2011; Suthers 2015); transportation and communication networks (Deo 2004); biological networks (Lacroix et al. 2004; Leser and Trissl 2009); workflows and business processes (Vanhatalo et al. 2008); customer relationship networks (Wu et al. 2009); intelligence networks (Ayers 1997; Chen et al. 2011); and much more![3]

As the volume of graph-structured data continues to grow, users may not be aware of its full details and may need to be assisted by querying systems which do not require queries to match exactly the data structures being queried, but rather can automatically make changes to the query so as to help the user find the information being sought. The `OPTIONAL` clause of SPARQL (Harris and Seaborne 2013) has the aim of returning matchings to a query that may fail to match some of the query's triple patterns. However, it is possible to "relax" a SPARQL query in ways other than just ignoring optional triple patterns, for example, making use of the knowledge encoded in an ontology associated with the data in order to replace an occurrence of a class in the query by a superclass, or an occurrence of a property by a superproperty.

This observation motivated the introduction in Hurtado et al. (2008) of a `RELAX` clause for querying RDF data, which can be applied to those triple patterns of a query that the user would like to be matched flexibly. These triple patterns are successively made more general so that the overall query returns successively more answers, at increasing 'costs' from the exact form of the query. We review this work on ontology-based query relaxation in this section, starting with an example application in heterogeneous data integration in Sect. 4.1.1.

Section 4.2 goes beyond conjunctive queries to consider conjunctive *regular path queries* over graph data, and *approximate answering* of such queries. In contrast to query relaxation, which generally returns additional answers compared to the exact form of a database query, query approximation returns potentially *different* answers to the exact form of a query.

Section 4.3 considers combining both query relaxation and approximate answering for conjunctive regular path queries over graph data, describing also an automaton-based implementation. Section 4.4 considers extending SPARQL 1.1 with query relaxation and approximation, describing an implementation based on query rewriting. Along the way, we consider applications of query relaxation and query approximation for graph data in areas such as heterogeneous data integration, ontology querying, educational networks, transport networks and analysis of user–system interactions. Section 4.5 covers additional topics: possible user interfaces for supporting users in incrementally constructing and understanding flexible queries

[1]http://linkeddata.org, http://www.w3.org/standards/semanticweb, accessed at 18/6/2015.

[2]https://snap.stanford.edu/data, accessed at 18/6/2015.

[3]See for example http://neo4j.com/use-cases, http://www.objectivity.com/products/infinitegraph, http://allegrograph.com/allegrograph-at-work, accessed at 18/6/2015.

and the answers being returned; and possible extensions to the query languages considered so far with additional flexibility beyond relaxation and approximation, and with additional expressivity in the form of path variables. Section 4.6 gives an overview of related work on query languages for graph data and flexible querying of such data. Section 4.7 gives our concluding remarks and possible directions of future work.

Flexible Database Query Processing

Before beginning our discussion of flexible query processing for graph data, we first review the main approaches to flexible query processing for other kinds of data. Due to the considerable breadth of this area, the references cited here are representative of the approaches discussed rather than an exhaustive list. Readers are referred to the proceedings of the bi-annual conference on Flexible Query Answering Systems (FQAS) for a broad coverage of work in this area.

Query languages for structured data models, such as SQL and OQL, include WHERE clauses that allow filtering criteria to be applied to the data matched by their SELECT clauses. Therefore, a natural way to relax queries expressed in such languages is by dropping a selection criterion, or by 'widening' a selection criterion so as to match a broader range of values (Bosc and Pivert 1992; Heer et al. 2008). Another common approach to query relaxation is to allow *fuzzy* matching of selection criteria, accompanied by a scoring function that determines the degree of matching of the returned query answers (Galindo et al. 1998; Na and Park 2005; Bordogna and Psaila 2008; Bosc et al. 2009). Conversely, queries can be made more specific by adding user preferences as additional filter conditions, with possibly fuzzy matching of such conditions (Mishra and Koudas 2009; Eckhardt et al. 2011). Chu et al. (1996) use type abstraction hierarchies to both generalise and specialise queries, while Zhou et al. (2007) explore statistically based query relaxation through 'malleable' schemas containing overlapping definitions of data structures and attributes.

Turning to approximate query answering, approaches include histograms (Ioannidis and Poosala 1999), wavelets (Chakrabarti et al. 2001) and sampling (Babcock et al. 2003). Sassi et al. (2012) describe a system that enables the user to issue an SQL aggregation query, see results as they are being produced, and dynamically control query execution. Fink and Olteanu (2011) study approximation of conjunctive queries on probabilistic databases by specifying lower- and upper-bound queries that can be computed more efficiently.

In principle, techniques proposed for flexible querying of structured data can also be applied to graph-structured data. However, such techniques do not focus on the connections (i.e. edges and paths) inherent in graph-structured data, thus missing opportunities for further supporting the user through approximation or relaxation of the path structure that may be present in a graph query.

Semi-structured data models aim to support data that are self-describing and that need not rigidly conform to a schema (Abiteboul et al. 1997; Buneman et al. 2000; Fernandez et al. 2000; Bray et al. 2008). Generally, such data can be modelled as a tree, though cyclic connections between nodes may also be allowed by the model (e.g. in XML, through the ID/IDREF constructs). Much work has been done on relaxing tree-pattern queries over XML data. For example, Amer-Yahia et al. (2004) undertake query relaxation through removal of conditions from XPath expressions; Theobald et al. (2005) support relaxation by expanding queries using vocabulary information drawn from an ontology or thesaurus; Liu et al. (2010) use available XML schemas to relax queries; and Hill et al. (2010) use ontologies such as Wordnet to guide XML query relaxation. Buratti and Montesi (2008) discuss query approximation for XML based on the notion of a cost-based edit distance for transforming one path into another within an XQuery expression, while Almendros-Jimenez et al. (2014) propose a fuzzy approach to XPath query evaluation.

Similar approaches to those developed for XML can be adopted for flexible querying of graph-structured data, and indeed in subsequent sections of this chapter we discuss ontology-based relaxation of graph queries and also edit distance-based ranking of approximate answers to graph queries. However, the techniques proposed for flexibly querying XML generally assume one kind of relationship between entities (parent-child), whereas in graph-structured data there may be numerous relationships, potentially giving rise to higher complexity and diversity in the data and requiring query approximation and relaxation techniques that are able to operate on the relationships referenced within a user's query.

4.1.1 Example: Heterogeneous Data Integration

Much work has been done since the early 1990s in developing architectures and methodologies for integrating biological data (Goble and Stevens 2008). Such integrations are beneficial for scientists by providing them with easy access to more data, leading to more extensive and more reliable analyses and, ultimately, new scientific insights. Traditional data integration methodologies (Batini et al. 1986) require semantic mappings between the different data sources to be initially determined, so that a global integrated schema or ontology can be created through which the data in the sources can then be accessed. This approach means that significant resources for data integration projects must be committed upfront, and an active area of research is how to reduce this upfront effort (Halevy et al. 2006). A general approach adopted is to present initially all of the source data in an unintegrated format, and to provide tools that allow data integrators to incrementally identify semantic relationships between the different data sources and incrementally improve the global schema. Such an approach is termed 'pay-as-you-go' (Sarma and et al. 2008), since the integration effort can be committed incrementally as time and resources allow.

Heterogeneous data integration was identified in Hurtado et al. (2008) as a potential Use Case for flexible query processing over graph data. To illustrate, the In Silico Proteome Integrated Data Environment Resource (ISPIDER) project developed an integrated platform bringing together three independently developed proteomics data sources, providing an integrated global schema and support for distributed queries posed over this (Siepen et al. 2008).[4] The development of the global schema took many months. An alternative approach would have been to adopt a 'pay-as-you-go' integration approach, refining the global ontology by incrementally identifying common concepts between the data sources and integrating these using additional superclasses and superproperties.

For example, the initial ontology may include (amongst others) the following classes arising from three source databases, DB_1, DB_2, DB_3:

- $Peptide_1$, $Protein_1$, $Peptide_2$, $Protein_2$, $Peptide_3$, $Protein_3$

For simplicity here, we assume that common concepts are commonly named, and we identify the data source relating to a concept by its subscript. Likewise, it may include (amongst others) the following 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

(In proteomics, proteins consist of several peptides, each peptide comprising a sequence of amino acids; hence the properties $PepSeq_i$ above, in which the amino acid sequence is represented as a Literal. In proteomics experiments, several peptides may result from a protein identification process and each peptide aligns against a set of proteins; hence the properties $Aligns_i$ above. Each protein is characterised by an Accession Number, c.f. the properties $AccessNo_i$ above, a textual description, its predicted mass, the organism in which it is found, etc.)

A data integrator may observe some semantic alignments between the above classes and properties and may add the following superclasses and superproperties to the ontology in order to semantically integrate the underlying data extents from the three databases:

- Superclass $Peptide$ of classes $Peptide_i$, $1 \leq i \leq 3$
- Superclass $Protein$ of classes $Protein_i$, $1 \leq i \leq 3$
- Superproperty $PepSeq$ of properties $PepSeq_i$, $1 \leq i \leq 3$, with domain $Peptide$ and range Literal
- Superproperty $Aligns$ of properties $Aligns_i$, $1 \leq i \leq 3$, with domain $Peptide$ and range $Protein$
- Superproperty $AccessNo$ of properties $AccessNo_i$, $1 \leq i \leq 3$, with domain $Protein$ and range Literal.

A fragment of this global ontology is shown in Fig. 4.1 (omitting the $AccessNo_i$ and $AccessNo$ properties, and the domain and range information of $PepSeq$ and $Aligns$).

[4]The example presented here is a simplification of one given in Hurtado et al. (2008).

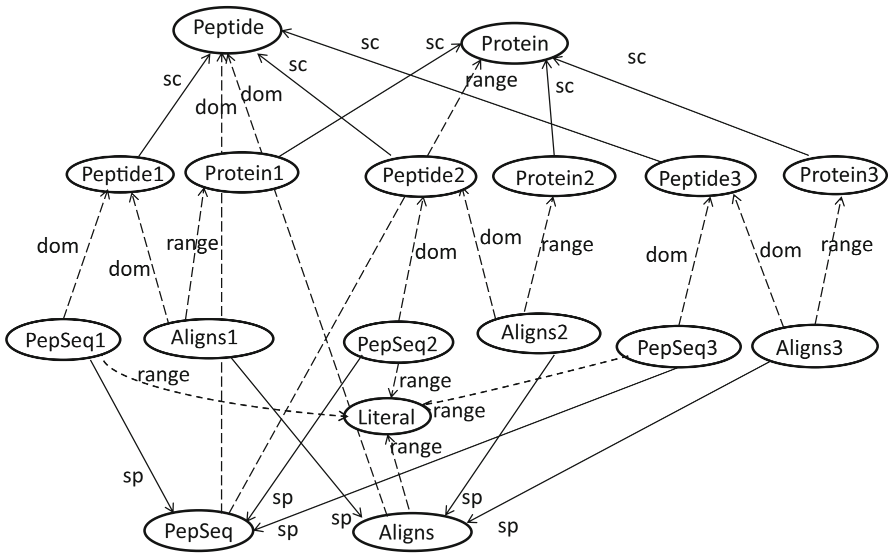


Fig. 4.1 Example ontology

Consider now the following query posed over the global ontology by a user who is only familiar with DB_1:

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

The syntax used here is that of a conjunctive query comprising one or more *triple patterns* on its right-hand side (RHS)—see Sect. 4.1.2, and zero or more variables on its left-hand side (LHS), which must also appear in the RHS. The entire RHS comprises a *graph pattern*—see Sect. 4.1.2. Variables are distinguished by an initial `?`. In its non-relaxed form, this query will return the identifiers and accession numbers of proteins identified in DB_1 through experiments yielding the peptide sequence 'ATLITFLCDR'.

A first level of relaxation of all three triple patterns in the above query results in the following query:

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

Evaluation of this query will expand the result set to include similar results also from DB_2 and DB_3, without the user needing to have detailed knowledge of their schemas.

In contrast to conventional data integration approaches, this kind of incremental integration coupled with flexible querying requires less upfront integration effort,

Fig. 4.2 RDFS inference rules

$$\text{Subproperty (1) } \frac{(a,sp,b)(b,sp,c)}{(a,sp,c)} \quad (2)\frac{(a,sp,b)(x,a,y)}{(x,b,y)}$$
$$\text{Subclass (3) } \frac{(a,sc,b)(b,sc,c)}{(a,sc,c)} \quad (4)\frac{(a,sc,b)(x,type,a)}{(x,type,b)}$$
$$\text{Typing (5) } \frac{(a,dom,c)(x,a,y)}{(x,type,c)} \quad (6)\frac{(a,range,d)(x,a,y)}{(y,type,d)}$$

allows a more exploratory approach to query answering, and does not require the user to have comprehensive knowledge of the entire global schema.

4.1.2 Theoretical Foundations of Query Relaxation

Hurtado et al. (2008) studied query relaxation in the setting of the RDF/S data model and showed that query relaxation can be naturally formalised using *RDFS entailment*. The entailment was characterised by the derivation rules given in Fig. 4.2, grounded in the semantics developed in Gutierrez et al. (2004) and Hayes (2004), and encompassing a fragment of the overall set of RDFS entailment rules known as ρDF (Munoz et al. 2007).

In the setting of Hurtado et al. (2008), an *ontology* K is a directed graph (N_K, E_K) where each node in N_K represents either a class or a property, so $N_K = classNodes\ (N_K) \cup propertyNodes(N_K)$; and each edge in E_K is labelled with a symbol from the set $\{sc, sp, dom, range\}$. These edge labels encompass a fragment of the RDFS vocabulary: rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain, rdfs:range, respectively.

In the accompanying data graph $G = (N, E)$, each node in N represents an instance or a class and each edge in E a property. The intersection of N and N_K is contained in $classNodes(N_K)$. The predicate $type$, representing the RDF vocabulary rdf:type, can be used in E to connect an instance of a class to a node representing that class.

Pairwise disjoint sets U and L of URIs and literals are assumed, respectively. Also assumed is an infinite set V of *variables*, disjoint from U and L. We abbreviate any union of the sets U, L and V by concatenating their names, e.g. $UL = U \cup L$.

Nodes in N are labelled with constants from UL (blank nodes are not considered in this work, and in any case their use is discouraged for Linked Data). Edges in E are labelled either with $type$ or a with symbol drawn from a finite alphabet Σ such that $type \notin \Sigma$ and $\Sigma \cup \{type\} \subset U$.

An *RDF triple* is a tuple $\langle s, p, o\rangle \in U \times U \times (U \cup L)$, where s is the subject, p the predicate and o the object of the triple. A *triple pattern* is a tuple $\langle x, p, y\rangle \in UV \times UV \times UVL$. A *graph pattern* is a set of triple patterns. Given a triple pattern t (graph pattern P), $vars(t)$ ($vars(P)$) is the set of variables occurring in it.

An *RDF/S graph* $I = (N_I, E_I)$ is the union of an ontology graph $K = (N_K, E_K)$ and a data graph $G = (N, E)$, i.e. $N_I = N_K \cup N$ and $E_I = E_K \cup E$.

Fig. 4.3 Additional rules for computing the extended reduction of an ontology

$$\text{(e1)}\ \frac{(b,dom,c)(a,sp,b)}{(a,dom,c)} \quad \text{(e2)}\ \frac{(b,range,c)(a,sp,b)}{(a,range,c)}$$
$$\text{(e3)}\ \frac{(a,dom,b)(b,sc,c)}{(a,dom,c)} \quad \text{(e4)}\ \frac{(a,range,b)(b,sc,c)}{(a,range,c)}$$

An RDF/S graph I_1 *entails* an RDF/S graph I_2, denoted $I_1 \models_{RDFS} I_2$, if I_2 can be derived by applying the rules in Fig. 4.2 iteratively to I_1.

The *closure* of an RDF/S graph I under these rules is denoted by $cl(I)$. Given an RDF/S graph I, query evaluation takes place on the graph given by restricting $cl(I)$ to nodes in $N \cup classNodes(N_K)$ and edges with labels in $\Sigma \cup \{type\} \cup propertyNodes(N_K)$. Each such edge is viewed as an RDF triple for the purposes of query evaluation.

In order to apply relaxation to queries, the subgraphs of the ontology K induced by edges labelled sc and sp need to be *acyclic*, so that an unambiguous cost can be assigned to a relaxed query. Moreover, K must be equal to its *extended reduction*, $extRed(K)$, which is computed as follows:

(a) Compute $cl(K)$
(b) Apply the rules of Fig. 4.3 in reverse until no more rules can be applied (applying a rule in reverse means deleting a triple deducible by the rule)
(c) Apply rules 1 and 3 of Fig. 4.2 in reverse until no more rules can be applied

Requiring that $K = extRed(K)$ allows *direct relaxations* to be applied to queries (see below), which correspond to the ‘smallest’ possible relaxation steps. This in turn allows an unambiguous cost to be associated with relaxed queries, so that query answers can be returned to users incrementally in order of increasing cost (see Hurtado et al. (2008) for a detailed discussion).

Following the terminology of Hurtado et al. (2008), a triple pattern $\langle x, p, y\rangle$ *directly relaxes* to a triple pattern $\langle x', p', y'\rangle$ with respect to an ontology $K = extRed(K)$, denoted $\langle x, p, y\rangle \prec \langle x', p', y'\rangle$, if $vars(\langle x, p, y\rangle) = vars(\langle x', p', y'\rangle)$ and $\langle x', p', y'\rangle$ is derived from $\langle x, p, y\rangle$ by applying some rule i, $1 \le i \le 6$, from Fig. 4.2.

A triple pattern $\langle x, p, y\rangle$ *relaxes to* a triple pattern $\langle x', p', y'\rangle$, denoted $\langle x, p, y\rangle \le \langle x', p', y'\rangle$, if there is a sequence of direct relaxations that derives $\langle x', p', y'\rangle$ from $\langle x, p, y\rangle$. The *relaxation cost* of deriving $\langle x', p', y'\rangle$ from $\langle x, p, y\rangle$ is the minimum cost of applying such a sequence of direct relaxations.

An essential aspect of this approach, which distinguishes it from earlier work on query relaxation, is that the answers to a query are ranked based on how ‘closely’ they satisfy the query. The notion of ranking is based on a structure called the *relaxation graph*, in which relaxed versions of the original query are ordered from less to more general.

To illustrate, Fig. 4.4 shows the relaxation graphs of two triple patterns:

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

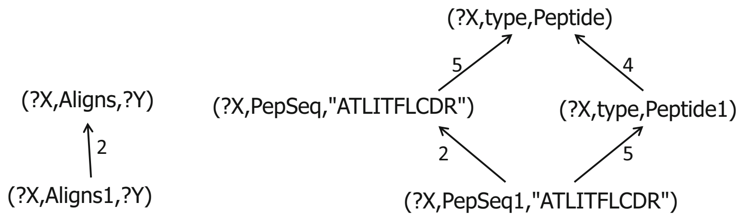


Fig. 4.4 Triple pattern relaxation graphs

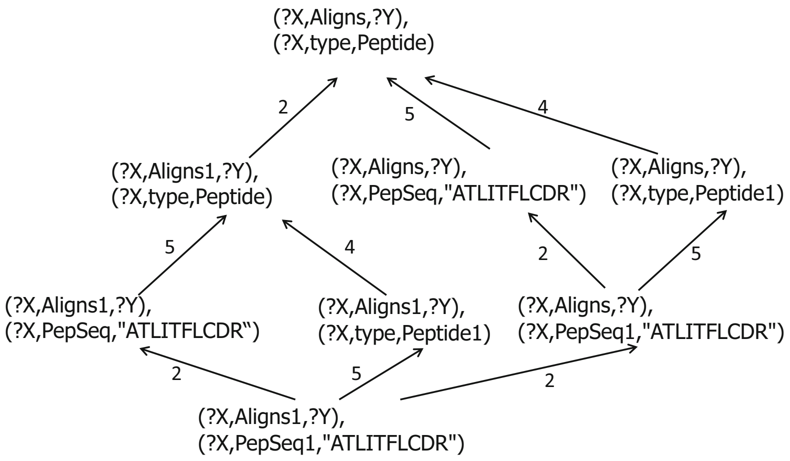

(?X,Aligns,?Y), (?X,type,Peptide)
2
5
4
(?X,Aligns1,?Y), (?X,type,Peptide)
(?X,Aligns,?Y), (?X,PepSeq,"ATLITFLCDR")
(?X,Aligns,?Y), (?X,type,Peptide1)
5
4
2
5
(?X,Aligns1,?Y), (?X,PepSeq,"ATLITFLCDR")
(?X,Aligns1,?Y), (?X,type,Peptide1)
(?X,Aligns,?Y), (?X,PepSeq1,"ATLITFLCDR")
2
5
2
(?X,Aligns1,?Y),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

Fig. 4.5 Graph pattern relaxation graph

assuming that K is the ontology of Fig. 4.1. The edges of the relaxation graph are labelled with the rule number from Fig. 4.2 which has been applied to obtain a relaxed triple pattern from one directly below it.

Triple pattern relaxation is generalised to graph pattern relaxation using the notion of the *direct product* of partial orders. The *direct product* of n partial orders $\alpha_1, \alpha_2, \dots \alpha_n$, denoted $\alpha_1 \otimes \alpha_2 \otimes \dots \otimes \alpha_n$, is another partial order α such that $(a_1, \dots a_n)\ \alpha\ (b_1, \dots b_n)$ if and only if $a_i\ \alpha_i\ b_i$ for all $1 \leq i \leq n$.

Consider graph patterns consisting of n triple patterns, $t_1, \dots, t_n$. The graph pattern relaxation relation $\leq_n$ is defined as $\leq \otimes \leq \dots \otimes \leq$ (n times). The direct graph pattern relaxation relation $\prec_n$ is the reflexive and transitive reduction of $\leq_n$. The *relaxation graph* of a graph pattern is the directed acyclic graph induced by $\prec_n$.

As an example, consider the graph pattern

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

Figure 4.4 shows the relaxation graphs of its two triple patterns and Fig. 4.5 shows their direct product.

```
Algorithm RelaxEval
Input: a query $Q$, where $body(Q) = \{t_1, \ldots, t_n\}$; an RDF/S graph $I$; and an integer $maxLevel$.
Output: the set of ranked answer tuples.

1. $k := 0$, $stillMore := true$
2. For each $t_i \in body(Q)$, compute the relaxation graph $R_i$ of $t_i$ up to level $maxLevel$.
3. While ($k \leq maxLevel$ and $stillMore$) do
    a. For each combination $t'_1 \in R_1, \ldots, t'_n \in R_n$ such that $\sum_i level(t'_i, R_i) = k$ do output
       $\pi_H(\texttt{deltaFind}(t'_1, I) \bowtie \ldots \bowtie \texttt{deltaFind}(t'_n, I))$
    b. $k := k + 1$
    c. $stillMore :=$ exist nodes $t'_1 \in R_1, \ldots, t'_n \in R_n$ such that $\sum_i level(t'_i, R_i) = k$
```

Fig. 4.6 Algorithm to compute the relaxed answer of a query

Algorithm **RelaxEval** in Fig. 4.6 (from Hurtado et al. 2008) incrementally computes the relaxed answer to a query Q and returns the answers in ranked order, where *maxLevel* is the maximum number of relaxations desired for the evaluation of Q; $body(Q)$ denotes the graph pattern in the RHS of Q; and the set $\texttt{deltaFind}(t'_i, I)$ consists of the triples $\langle s, p, o \rangle \in I$ such that t'_i matches $\langle s, p, o \rangle$ and no triple pattern directly below t'_i in the relaxation graph of t_i matches $\langle s, p, o \rangle$. This algorithm assumes that all direct relaxations of triple patterns have the same cost. We will see later two methods that are able to handle different costs.

Another class of relaxations is also discussed in Hurtado et al. (2008), consisting of relaxations that can be entailed without an ontology, such as dropping triple patterns, replacing constants with variables and breaking join dependencies. We refer the reader to that paper for details of these.

4.2 Beyond Conjunctive Queries: Regular Path Queries

Regular path queries have been proposed by several researchers as a means of assisting users in querying complex or irregular graph data by finding *paths* through the data graph that match a given regular expression over edge labels (Cruz et al. 1987; Mendelzon and Wood 1989, 1995; Fernandez and Suciu 1998).

Consider the same simple data model as introduced above, comprising a directed graph $G = (N, E)$, where each node in N is labelled with a constant and each edge in E is labelled with a symbol drawn from a finite alphabet $\Sigma \cup \{type\}$. Edges can be traversed both from their source to their target node and in reverse, from their target to their source node. The *inverse* of an edge label l, denoted by l^-, is used to specify the reverse traversal of an edge. Let $\Sigma^- = \{l^- \mid l \in \Sigma\}$. If $l \in \Sigma \cup \Sigma^- \cup \{type, type^-\}$, we use l^- to mean the *inverse* of l, that is, if l is a for some $a \in \Sigma \cup \{type\}$, then l^- is a^-, while if l is a^- for some $a \in \Sigma \cup \{type\}$, then l^- is a.

A *regular path query* (RPQ) Q has the form

$$vars \leftarrow (X, R, Y) \tag{4.1}$$

where X and Y are constants or variables, R is a regular expression over $\Sigma \cup \{type\}$, and $vars$ is the subset of $\{X, Y\}$ that are variables. A *regular expression R over $\Sigma \cup \{type\}$* is defined as follows:

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

where ϵ is the empty string, a is any symbol in $\Sigma \cup \{type\}$, '_' denotes the disjunction of all constants in $\Sigma \cup \{type\}$, and the operators have their usual meaning.

A *semipath* (Calvanese et al. 2000) p in $G = (N, E)$ from $v \in N$ to $w \in N$ is a sequence of the form $(v_1, l_1, v_2, l_2, v_3, \ldots, v_n, l_n, v_{n+1})$, where $n \geq 0$, $v_1 = v$, $v_{n+1} = w$ and for each v_i, l_i, v_{i+1} either $v_i \xrightarrow{l_i} v_{i+1} \in E$ or $v_{i+1} \xrightarrow{l_i^-} v_i \in E$. A semipath p *conforms* to a regular expression R if $l_1 \cdots l_n \in L(R)$, the language denoted by R.

Given an RPQ Q and graph G, let θ be a matching from $\{X, Y\}$ to nodes of G that maps each constant to itself and such that there is a semipath from $\theta(X)$ to $\theta(Y)$ whose concatenation of edge labels is in $L(R)$. The *answer* of Q on G is the set of tuples $\theta(vars)$ for all such matchings θ.

A *conjunctive regular path query* (CRPQ) Q consisting of n conjuncts has the form

$$Z_1, \ldots, Z_m \leftarrow (X_1, R_1, Y_1), \ldots, (X_n, R_n, Y_n) \tag{4.2}$$

in which each X_i and Y_i, $1 \leq i \leq n$, is a variable or constant, each Z_i, $1 \leq i \leq m$, is a variable appearing in the body of Q, and each R_i, $1 \leq i \leq n$, is a regular expression over $\Sigma \cup \{type\}$.

Given a CRPQ Q and graph G, let θ be a matching from variables and constants of Q to nodes of G such that (i) each constant is mapped to itself, and (ii) there is a semipath from $\theta(X_i)$ to $\theta(Y_i)$ that conforms to R_i, for all $1 \leq i \leq n$. The *answer* of Q on G is the set of m-tuples $\theta(Z_1, \ldots, Z_m)$ for all such matchings θ.

The answer to a CRPQ Q on a graph G can be computed as follows. First find, for each $1 \leq i \leq n$, a binary relation r_i over the scheme (X_i, Y_i) such that tuple $(v, w) \in r_i$ if and only if there is a semipath from node v to node w in G that conforms to R_i, $v = X_i$ if X_i is a constant, and $w = Y_i$ if Y_i is a constant. Then form the natural join of the relations $r_1, \ldots, r_n$ and project over Z_1 to Z_m.

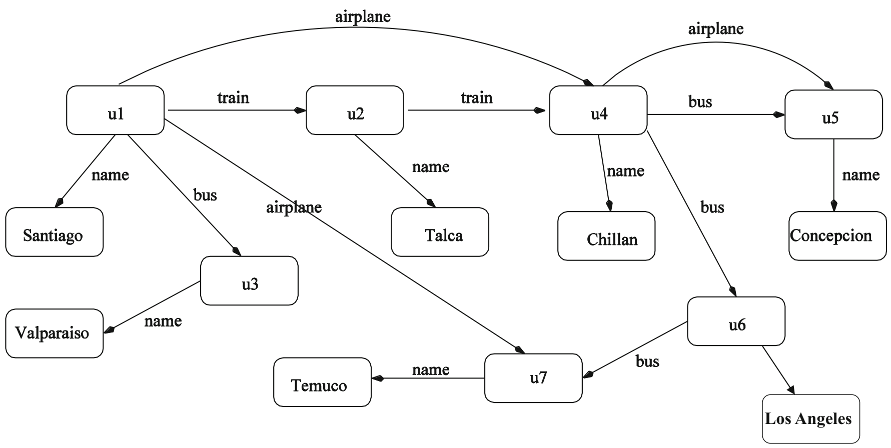


Fig. 4.7 Part of a transport network

4.2.1 Example: Transport Networks

Consider the graph in Fig. 4.7 showing information about a transport network. The nodes of the graph are city identifiers and city names. The edges show direct transport links from one city to another.[5]

Suppose we want to find the cities from which we can travel to city `u5` using only airplanes as well as to city `u6` using only trains or buses. This can be expressed by the following CRPQ query Q:

```
?X <- (?X, airplane+, u5),
      (?X, (train|bus)+, u6)
```

When Q is evaluated on G, the bindings for `?X` generated by the first conjunct are `u1`, `u4`, while those for the second conjunct are `u1`, `u2`, `u4`. Hence the answer is `u1`, `u4`.

Suppose now that a user who has little knowledge of the structure of the data wishes to find all cities reachable from Santiago by direct flights and poses the following query which makes use of the query approximation operator APPROX that we will discuss in more detail in Sect. 4.2.2:

```
?X <- APPROX (Santiago,airplane,?X)
```

The exact form of this query returns no answers because it does not match the structure of the graph. Inserting `name` after `airplane`, to obtain the regular expression `airplane. name` (at a cost of c_1, say) still returns no answers. Inserting `name` before `airplane`, to obtain the regular expression `name.airplane.name` (at an additional cost of c_1) again returns no answers. Finally, inverting the first `name`

[5]This example is adapted from one in Hurtado et al. (2009b).

label, to obtain `name-.airplane.name` (at an additional cost of c_2, say) returns as answers `Temuco` and `Chillan`, at an overall cost of $2c_1 + c_2$.

Suppose now a user formulates the following query to find cities reachable from Santiago by train, directly or indirectly. The user is also potentially interested in routes combining train and bus, and elects to allow replacement of `train` by `bus` in their query, as well as insertion of `train` and `bus`:

```
?X <- APPROX (Santiago,name-.train+.name,?X)
```

The exact answers to this query are `Talca` and `Chillan`. Replacing one occurrence of `train` by `bus` (at a cost of c_3, say), to obtain the regular expression `name-.bus.train*.name`, returns `Valparaiso` at cost c_3. Inserting `train` after `name-` (at a cost of c_4, say), to obtain `name-.train.bus.train*.name`, returns no more answers. Inserting again `train` after `name-` (at a cost of c_4), to obtain `name-.train.train.bus.` train*.name, returns answers `Concep- cion` and `Los Angeles`, at a cost of $c_3 + 2c_4$. Inserting `bus` before `train*` (at a cost of c_5, say), to obtain `name-.train.train.bus.bus.train*.name` returns answer `Temuco`, at a cost of $c_3 + 2c_4 + c_5$.

4.2.2 Approximate Matching of CRPQs

We have seen above examples of circumstances where *approximate matching* of regular path queries and ranking of query results in terms of how closely they match the original query can help the user find relevant information from unfamiliar, irregular graph data. The work in Hurtado et al. (2009b) discusses how such approximate answers can be computed for CRPQ queries, based on edit operations such as insertions, deletions, inversions, substitutions and transpositions of edge labels being applied to a semipath. A user can specify which of these edit operations should be applied by the system when answering a particular query, and the cost to be assigned when applying each operation, more formally presented as follows.

The *edit distance* from a semipath p to a semipath q is the minimum cost of any sequence of edit operations which transforms the sequence of edge labels of p to the sequence of edge labels of q. 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 that conforms to R.

Given a graph G, an RPQ Q of the form (4.1) and a matching θ from variables and constants of Q to nodes in G such that any constant is mapped to itself, the tuple $\theta(vars)$ has *edit distance edist*(p, R) to Q if p is a semipath from $\theta(X)$ to $\theta(Y)$ in G having the minimum edit distance to R of any semipath from $\theta(X)$ to $\theta(Y)$. (Note that if p conforms to R, then $\theta(vars)$ has edit distance 0 to Q.)

The *approximate top-k answer* of Q on G is the list of k tuples $\theta(vars)$ with minimum edit distance to Q, ranked in order of increasing edit distance to Q.

Generalising to CRPQs, given a graph G, a CRPQ Q of the form (4.2), and a matching θ from variables and constants of Q to nodes in G such that any constant is mapped to itself, the tuple $\theta(Z_1, \ldots, Z_m)$ has *edit distance* $edist(p_1, R_1) + \cdots + edist(p_n, R_n)$ to Q if each p_i is a semipath from $\theta(X_i)$ to $\theta(Y_i)$ in G having the minimum edit distance to R_i of any semipath from $\theta(X_i)$ to $\theta(Y_i)$. The *approximate top-k answer* of Q on G is the list of k distinct tuples $\theta(Z_1, \ldots, Z_m)$ with minimum edit distance to Q, ranked in order of increasing edit distance to Q.

Since the answers for single conjuncts are ordered by non-decreasing edit distance, pipelined execution of any rank-join operator (see Finger and Polyzotis 2009) can be used to output the answers to a CRPQ Q in order of non-decreasing edit distance.

There are a fixed number of variables in the head of a CRPQ query, so if its conjuncts are acyclic then the evaluation of the approximate top-k answer can be accomplished in polynomial time (see Gottlob et al. 2001; Grahne and Thomo 2001; Hurtado et al. 2009b).

4.3 Combining Approximation and Relaxation in CRPQs

The ideas from the previous two sections can be combined to allow *both* relaxation and approximation of CRPQs, providing their combined flexibility within one query processing framework. This possibility was first explored in Poulovassilis and Wood (2010).

4.3.1 Example: Educational Networks

The L4All system (de Freitas et al. 2008) was developed to support learners in a network of Further and Higher Education institutions in the London region. The system allows users to create and maintain a chronological record of their learning, work and personal episodes—their 'timelines'—with the aim of supporting learners in exploring learning and career opportunities and in planning and reflecting on their learning. Figures 4.8 and 4.9 illustrate a fragment of the data and metadata relating to two users' timelines. The episodes within a timeline have a start and an end date associated with them (for simplicity these are not shown in the figure). Episodes are ordered by their start date—as indicated by edges labelled `next`. There are several types of episode, e.g. `University` and `Work` episodes. Associated with each type of episode are several properties—the figures show just two of these, `qualif[ication]` and `job`.[6]

Suppose that Mary is studying for a BA in English and wishes to find out what possible future career choices there are for her. Timelines may have edges labelled `prereq` between episodes, indicating that the timeline's owner believes

[6]This example is adapted from one in Poulovassilis and Wood (2010).

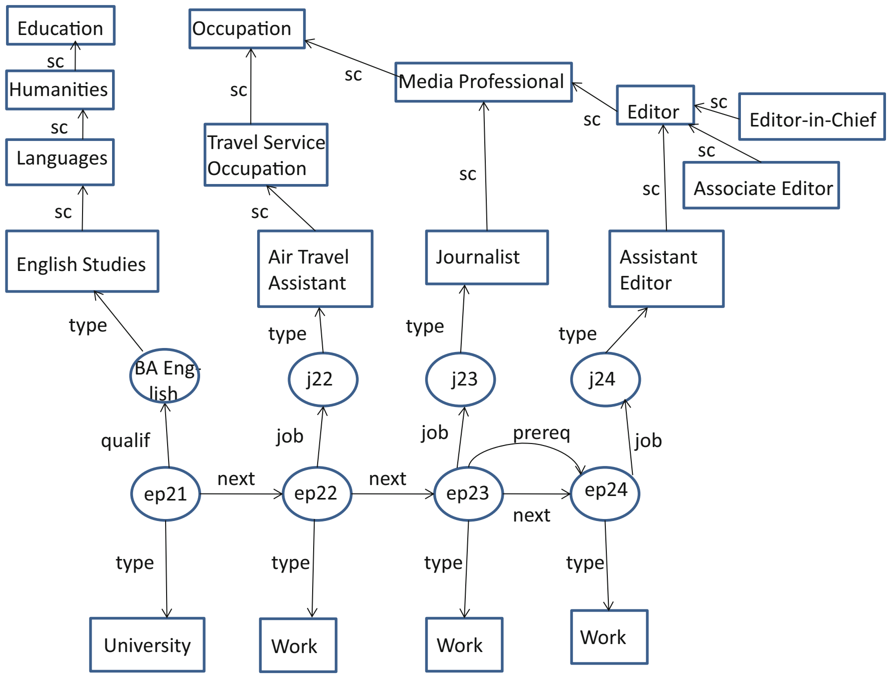


Fig. 4.8 Fragment of data and metadata from Anne's timeline

that undertaking an earlier episode was necessary in order for them to be able to proceed to or achieve a later episode. So Mary might pose this CRPQ query, Q_1:

```
(?E2,?P)<-(?E1,type,University),
          (?E1,qualif.type,EnglishStudies),
          (?E1,prereq+,?E2),
          (?E2,type,Work),
          (?E2,job.type,?P)
```

However, this will return no results even though Anne's timeline in Fig. 4.8 contains information that would be relevant to Mary. This is because, in practice, users may or may not create `prereq` metadata relating to their timelines.

If Mary chooses to allow replacement of the edge label `prereq` in her query by the label `next` (at an edit cost of 1, say), she can submit a variant of Q_1:

```
(?E2,?P)<-(?E1,type,University),
          (?E1,qualif.type,EnglishStudies),
         APPROX (?E1,prereq+,?E2),
          (?E2,type,Work),
          (?E2,job.type,?P)
```

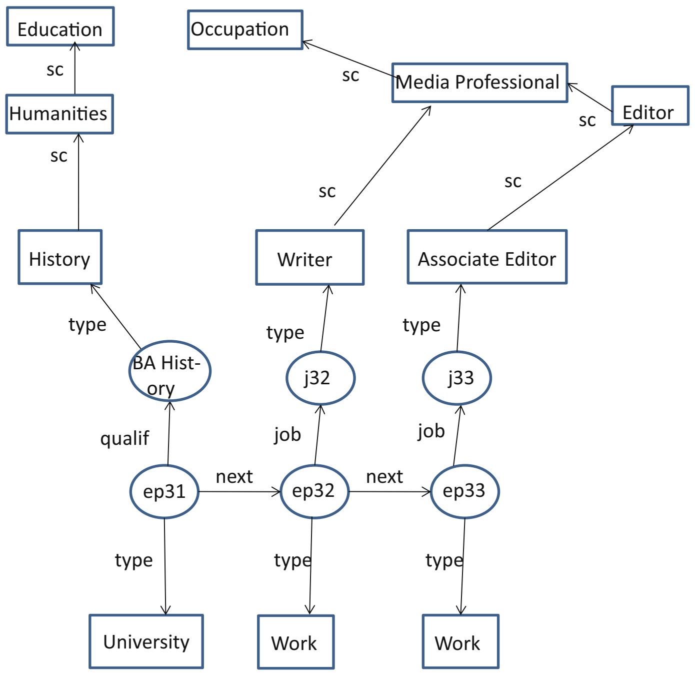


Fig. 4.9 Fragment of data and metadata from Bob's timeline

The regular expression `prereq+` can be approximated by the regular expression `next.prereq*` at edit distance 1 from `prereq+`, returning the answer

```
(ep22,AirTravelAssistant)
```

Mary may judge this not to be relevant and may seek further results, at a further level of approximation. The regular expression `next.prereq*` can be approximated by `next.next.prereq*`, at edit distance 2 from `prereq+`, returning the answers

```
(ep23,Journalist),(ep24,AssistantEditor)
```

Mary may judge these as being relevant, and she can then request the system to return the whole of Anne's timeline for her to explore further.

The previous example took as input a starting timeline episode and explored possible future work choices. The next example additionally specifies an end goal and explores how someone might reach this from a given starting point.

Suppose now Mary knows she wants to become an Assistant Editor and would like to find out how she might achieve this, given that she's done an English degree. Mary might pose this query, Q_2:

```
(?E2,?P)<-(?E1,type,University),
          (?E1,qualif.type,EnglishStudies),
          APPROX(?E1,prereq+,?E2),(?E2,job.type,?P)
          APPROX(?E2,prereq+,?Goal),(?Goal,type,Work),
          (?Goal,job.type,AssistantEditor)
```

At edit distance 0 and 1 there are no results from Anne's timeline. At edit distance 2, the answers

```
(ep22,AirTravelAssistant),(ep23,Journalist)
```

are returned, the second of which gives Mary potentially useful information.

Suppose Mary wants to know what other jobs, similar to an Assistant Editor, might be open to her. There are many categories of jobs classified under `Media Professional` but none of these will be matched by her query Q_2 above. She can pose instead query Q_3:

```
(?E2,?P)<-(?E1,type,University),
          (?E1,qualif.type,EnglishStudies),
          APPROX(?E1,prereq+,?E2),(?E2,job.type,?P)
          APPROX(?E2,prereq+,?Goal),(?Goal,type,Work),
          RELAX(?Goal,job.type,AssistantEditor) ,
```

which allows the system to relax `Assistant Editor` to its parent class `Editor`, matching jobs such as `Assistant Editor`, `Associate Editor` etc., as well as in parallel approximating the two instances of `prereq+`. Query results will be returned in increasing overall cost.

As a further extension, suppose another user, Joe, wants to know what jobs similar to being an Assistant Editor might be open to someone who has studied English or a similar subject at university. Joe may pose query Q_4 which is the same as Q_3 above but with `RELAX` in front of the second conjunct:

```
(?E2,?P)<-(?E1,type,University),
          RELAX(?E1,qualif.type,EnglishStudies),
          APPROX(?E1,prereq+,?E2),(?E2,job.type,?P)
          APPROX(?E2,prereq+,?Goal),(?Goal,type,Work),
          RELAX(?Goal,job.type,AssistantEditor)
```

Suppose Joe sets the cost of relaxing a class to its parent class to 2 and replacing the label `prereq` by the label `next` to 1. Then, the answers produced for query Q_4 from the graphs in Figs. 4.8 and 4.9 are shown in the table below. The first seven columns refer to the answers produced for each of the query conjuncts. For brevity, we do not show the full answer tuples, only the variable instantiations for each conjunct. We also show the relaxation distance (cost), rd, for the second and seventh conjucts, and the edit distance, ed, for the third and fifth conjuncts. In the table, 'Air T.A.' stands for Air Travel Assistant, 'Assist. Ed.' for Assistant Editor and 'Assoc. Ed.' for Associate Editor. The final column shows the overall query answers and their overall distance (d) (which is the sum of the rd and ed values from the second, third, fifth and seventh conjuncts). For greater clarity, the tuples contributing to the first two answers are *italicised* and those contributing to the third answer are shown in **bold**.

?E1	?E1,*rd*	?E1,?E2,*ed*	?E2,?P	?E2,?Goal,*ed*	?Goal	?Goal,*rd*	?E2,?P,*d*
ep21	*ep21,0*	ep23,ep24,0	*ep22,Air T.A.*	*ep23,ep24,0*	ep22	*e24,0*	*ep23,Journalist,2*
ep31	**ep31,4**	*ep21,ep22,1*	*ep23,Journalist*	ep21,ep22,1	*ep23*	**e33,2**	*ep22,Air T.A.,6*
		ep22,ep23,1	ep24,Assist.Ed.	*ep22,ep23,1*	*ep24*	*e23,4*	**ep32,Writer,8**
		ep31,ep32,1	**ep32,Writer**	ep31,ep32,1	ep32	e32,4	
		ep32,ep33,1	ep33,Assoc.Ed.	**ep32,ep33,1**	**ep33**	e22,6	
		ep21,ep23,2		ep21,ep23,2			
		ep21,ep24,2		ep21,ep24,2			
		ep31,ep33,2		ep31,ep33,2			

A prototype implementation extending the original L4All system with this flexible querying functionality is described in Poulovassilis et al. (2012). A GUI is provided that allows the user to incrementally build up their query through a forms-based interface, including specifying their preferences for approximation or relaxation to be applied to each subquery. Drop-down menus are used for selecting classes, properties and regular expressions. The CRPQ query is automatically, and incrementally, generated by the system from the user's interactions and preferences. Visualisations are available that allow the user to view at a glance the subqueries they have constructed so far. Query results are displayed one screenful at a time, in increasing distance from the non-approximated, non-relaxed version of the user's query. For each result, an avatar representing the timeline's owner is displayed, as well as their name, the last episode in their timeline matching the user's query, the 'distance' at which this result has been retrieved, and a summary of the timeline's owner and the contents of their timeline. The aim of this summary information is to allow the user to decide if this timeline is relevant for their needs and if they wish to explore it in more detail. These functionalities were evaluated by two Lifelong Learning expert practitioners who gave positive feedback regarding the flexibility of the querying supported and the fact that there is a clear causality between a user's information requirements, as reflected in the query they have constructed, and the results returned by the system.

4.3.2 Automaton-Based Implementation Approach

We now discuss an automaton-based approach to evaluating regular path queries supporting both query approximation and query relaxation. The description is based on that from Poulovassilis and Wood (2010), with some modifications. We refer the reader to Poulovassilis and Wood (2010) and Poulovassilis et al. (2016) for full details.

4.3.2.1 Computing Approximate Answers

Approximate matching of an RPQ query Q with respect to a graph G is achieved by applying edit operations to sequences in $L(R)$. Let q be a sequence in $L(R)$ and l be a label in $\Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$. We assume support for the following edit operations, each at some non-negative cost: *insertion* of l into q, *deletion* of l from q, *substitution* of some label other than l by l in q. The cost of substitution is assumed to be less than the combined cost of insertion and deletion (otherwise the substitution operation would be redundant). The *inversion* operation is achieved through substitution, since this allows some label a in q to be substituted by a^-. The *transposition* operation can be achieved by applying a substitution operation to each of the two labels to be transposed.

Given an RPQ Q with body (X, R, Y) and a graph $G = (N, E)$, the *approximate answer* of Q on G can be computed as follows (the italicised terms are explained in more detail below):

1. A *weighted NFA*, M_R, recognising $L(R)$ is constructed from R.
2. A *query automaton*, M_Q, is constructed from Q.
3. An *approximate automaton*, A_Q, is constructed from M_Q.
4. The *product automaton*, H, of A_Q and G is constructed.
5. One or more shortest path traversals are performed on H in order to find the approximate answer of Q on G.

Definition 4.1 A *weighted non-deterministic finite state automaton* (weighted NFA) M_R recognising $L(R)$ is the same as a normal NFA except that each transition and each final state has a weight associated with it (all of which are initially zero). It can be constructed using Thompson's construction (Aho et al. 1974), which makes use of ϵ-transitions.

Formally, $M_R = (S, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \delta, s_0, S_f, \xi)$, where: S is the set of states; $\Sigma \cup \{type\}$ is the alphabet of edge labels in G; $\delta \subseteq S \times \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\} \times \mathbb{N} \times S$ is the transition relation; $s_0 \in S$ is the start state; S_f is the set of final states, initially only consisting of $s_f \in S$; and ξ is the final weight function mapping each state in S_f to a non-negative number (initially, this will be zero for s_f).

The *query automaton* M_Q for Q is M_R with additional annotations on the initial and final states: if X (resp. Y) is a constant c, then s_0 (s_f) is annotated with c; otherwise, s_0 (s_f) is annotated with the symbol $*$ which matches any constant.

Definition 4.2 The *approximate automaton* A_Q for Q is constructed by first constructing an automaton A_R from M_R. Formally, $A_R = (S, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \delta, s_0, S_f, \xi)$, with S, δ, s_0 and S_f initially defined as in Definition 4.1. A_R is then transformed as follows:

- For each transition $(s, a, 0, t) \in \delta$ ($s \neq t$ and $a \in \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$), the transition (s, ϵ, c_d, t) is added to δ, where c_d is the cost of *deletion*.

- All ϵ-transitions are removed from δ using the method of Droste et al. (2009). The method first computes the ϵ-*closure*, which is the set of pairs of states connected by a sequence of ϵ-transitions along with the minimum summed weight for each such pair. Then for each pair (s, t) with weight w in the ϵ-closure and each transition $(t, b, 0, u) \in \delta$ $(b \neq \epsilon)$, a new transition (s, b, w, u) is added to δ. If $t \in S_f$, then s is added to S_f with $\xi[s] = w$ if $\xi[s]$ was previously undefined, or with $\xi[s]$ set to the minimum of $\xi[s]$ and w otherwise.
- For each transition $(s, a, w, t) \in \delta$ and label $b \in \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$ $(b \neq a)$, the transition $(s, b, w + c_s, t)$ is added to δ, where c_s is the cost of *substitution*.
- For each state $s \in S$ and label $a \in \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$, the transition (s, a, c_i, s) is added to δ, where c_i is the cost of *insertion*.

The *approximate automaton* A_Q for Q is formed from A_R by annotating the initial and final states in A_R with the annotations from the initial and final states, respectively, in M_Q.

Definition 4.3 Let $A_Q = (S, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \delta, s_0, S_f, \xi)$ be an approximate automaton and $G = (N, E)$ be a graph. G can be viewed as an automaton in which each node is both an initial and a final state. The *product automaton* (Mendelzon and Wood 1989), H, of A_Q and G is the weighted automaton $(T, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \sigma, I, F, \xi)$, where $I \subseteq T$ is a set of initial states and $F \subseteq T$ is a set of final states. The set of states T is given by $\{(s, n) \mid s \in S \wedge n \in N\}$. The set of transitions σ consists of transitions of the form

- $((s, n), a, c, (s', n'))$ if $(s, a, c, s') \in \delta$ and $(n, a, n') \in E$
- $((s, n), a^-, c, (s', n'))$ if $(s, a^-, c, s') \in \delta$ and $(n', a, n) \in E$

The set of initial states I is given by $\{(s_0, n) \mid n \in N\}$. The set of final states F is given by $\{(s_f, n) \mid (s_f, n) \in T \wedge s_f \in S_f\}$. ξ is the final weight function mapping each state $s \in F$ to a non-negative number. The annotations on initial and final states in H are carried over from the corresponding initial and final states in A_Q.

Having formed the product automaton H, we can now compute the approximate answer of Q on G:

(i) Suppose first that X is a constant v. If $v \notin N$, then the answer is empty. If $v \in N$, we perform a shortest path traversal of H starting from the initial state (s_0, v). Whenever we reach a final state (s_f, n) in H we output n, provided n *matches* the annotation on (s_f, n) (recall that if Y is a constant the annotation on s_f will be that constant, and if Y is a variable the annotation will be the symbol $*$). Node n matches the annotation if the annotation is n or $*$.

(ii) Now suppose X is a variable. In this case, we again perform a shortest path traversal of H, outputting nodes as above, but this time starting from state (s_0, v) for every node $v \in N$.

Two optimisations to this naive traversal to avoid starting at every node of G are described in Selmer et al. (2015). Firstly, if Y is a constant, then (X, R, Y) is transformed to (Y, R^-, X), where R^- is the reversal of R, thus reverting to case (i) above. Otherwise (i.e. both X and Y are variables), we examine the labels on the transitions outgoing from the initial state of A_Q, s_0, we retrieve from G the set of edges (v, l, w) matching these labels, and we perform the shortest path traversal starting from state (s_0, v) for each such node v.

The above evaluation can be accomplished "on-demand" by incrementally constructing the edges of the product automaton H as required, rather than computing the entire graph H, as follows. Three collections are maintained (all initially empty):

- A set $\texttt{visited}_R$ containing tuples of the form (v, n, s), representing the fact that node n of G was visited in state s of A_Q having started the traversal from node v.
- A priority queue $\texttt{queue}_R$ containing tuples of the form (v, n, s, d, f), ordered by non-decreasing values of d, where d is the edit distance associated with visiting node n in state s having started from node v, and f is a flag denoting whether the tuple is 'final' or 'non-final'.
- A list $\texttt{answers}_R$ containing tuples of the form (v, n, d), where d is the smallest edit distance of this answer tuple to Q, ordered by non-decreasing values of d. This list is used to avoid returning an answer (v, n, d') if there is already an answer (v, n, d) with $d \leq d'$.

The evaluation of Q begins by adding to $\texttt{queue}_R$ the initial tuple or tuples $(v, v, s_0, 0, f)$ as detailed in (i) and (ii) above.

Procedure `getNext` is then called to return the next query answer, in order of non-decreasing edit distance from Q. `getNext` repeatedly dequeues the first tuple of $\texttt{queue}_R$, (v, n, s, d, f), adding (v, n, s) to $\texttt{visited}_R$ if the tuple is not a final one, until $\texttt{queue}_R$ is empty. If (v, n, s, d, f) is a final tuple and the answer (v, n, d') has not been generated before for some d', the triple (v, n, d) is returned after being added to $\texttt{answers}_R$. If (v, n, s, d, f) is not final tuple, we enqueue $(v, m, s', d + d', f))$ for each transition $\overset{d'}{\rightarrow} (s', m)$ returned by $\texttt{Succ}(s, n)$ such that $(v, m, s') \notin \texttt{visited}_R$. Here, the `Succ` function returns all transitions $\overset{d'}{\rightarrow} (s', m)$ such that there is an edge from (s, n) to (s', m) in H with cost d'. Within `Succ`, the function $\texttt{nextStates}(A_Q, s, a)$ returns the set of states in A_Q that can be reached from state s on reading input a, along with the cost of reaching each. If s is a final state, its annotation matches n, and the answer (v, n, d') has not been generated before for some d', then we add the final weight function for s to d, mark the tuple as final, and enqueue the tuple.

Procedure getNext(X, R, Y)

```
     Input: query conjunct $(X, R, Y)$
     Output: triple $(v, n, d)$, where $v$ and $n$ are instantiations of $X$ and $Y$
(1)  while $nonempty(queue_R)$ do
(2)      $(v, n, s, d, f) \leftarrow dequeue(\texttt{queue}_R)$
(3)      if $f \neq$ ‘$\texttt{final}$’ then
(4)          add $(v, n, s)$ to $\texttt{visited}_R$
(5)          foreach $\xrightarrow{d'} (s', m) \in Succ(s, n)$ s.t. $(v, m, s') \notin \texttt{visited}_R$ do
(6)              $enqueue(\texttt{queue}_R, (v, m, s', d + d', f))$
(7)          if $s$ is a final state and its annotation matches $n$ and
             $\nexists d'.(v, n, d') \in \texttt{answers}_R$ then
(8)              $enqueue(\texttt{queue}_R, (v, n, s, d + \xi[s], \text{‘}\texttt{final}\text{’}))$
(9)      else
(10)         if $\nexists d'.(v, n, d') \in \texttt{answers}_R$ then
(11)             append $(v, n, d)$ to $\texttt{answers}_R$
(12)             return $(v, n, d)$
(13) return $null$
```

Function Succ(s, n)

```
    Input: state $s$ of $A_Q$ and node $n$ of $G$
    Output: set of transitions which are successors of $(s, n)$ in $H$
(1) $W \leftarrow \emptyset$
(2) for $(n, a, m) \in G$ and $(p, d) \in \texttt{nextStates}(A_Q, s, a)$ do
(3)     add the transition $\xrightarrow{d} (p, m)$ to $W$
(4) return $W$
```

4.3.2.2 Computing Relaxed Answers

We now describe how the relaxed answer of an RPQ query Q with body (X, R, Y) can be computed, starting from the weighted NFA M_R that recognises $L(R)$. Below we denote by c_i the cost of applying rule i, $i \in \{2, 4, 5, 6\}$, from Fig. 4.2 (since queries and data graphs cannot contain edges labelled sc and sp, rules 1 and 3 are inapplicable to them, although of course they are used in computing the closure of the RDF/S graph).

Given a weighted automaton $M_R = (S, \Sigma \cup \{\texttt{type}\}, \delta, s_0, S_f, \xi)$ from which all ϵ-transitions have been removed, and an ontology K such that $K = \texttt{extRed}(K)$, an automaton $M_R^K = (S', \Sigma \cup \{\texttt{type}\}, \tau, S_0, S'_f, \xi')$ is constructed as described below. The set of states S' includes the states in S as well as any new states defined below. S_0 and S'_f are sets of initial and final states, respectively, with S_0 including the initial state s_0 of M_R, S'_f including the final states S_f of M_R, and both possibly including additional states as defined below. We obtain the *relaxed automaton* of Q with respect to K, M_Q^K, by annotating each state in S_0 and S'_f either with a constant or with $*$ depending on whether X and Y in Q are constants or variables. ξ' is the final weight function mapping states in S'_f to a non-negative number. The transition relation τ includes the transitions in δ as well as any transitions added to τ by the rules defined below. The rules below are repeatedly applied until no further changes to τ and S' can be inferred. The process terminates because of the assumption that the subgraphs of K induced by edges labelled sc and sp are acyclic.

- (rule 2(i)) For each transition $(s, a, d, t) \in \tau$ and triple $(a, sp, b) \in K$, add the transition $(s, b, d + c_2, t)$ to τ.
- (rule 2(ii)) For each transition $(s, a^-, d, t) \in \tau$ and triple $(a, sp, b) \in K$, add the transition $(s, b^-, d + c_2, t)$ to τ.
- (rule 4(i)) For each transition $(s, \mathtt{type}, d, t) \in \tau$ such that $t \in S'_f$, t is annotated with c, and $(c, sc, c') \in K$, add to S' a new final state t' annotated with c' (unless there is already such a final state); add a copy of all of t's outgoing transitions to t'; and add the transition $(s, \mathtt{type}, d + c_4, t')$ to τ.
- (rule 4(ii)) For each transition transition $(s, \mathtt{type}^-, d, t) \in \tau$ such that $s \in S_0$, s is annotated with c, and $(c, sc, c') \in K$, add to S' a new initial state s' annotated with c' (unless there is already such an initial state); add a copy of all of s's incoming transitions to s'; and add the transition $(s', \mathtt{type}^-, d + c_4, t)$ to τ.
- (rule 5(i)) For each transition $(s, a, d, t) \in \tau$ such that $t \in S'_f$ and $(a, dom, c) \in K$, add to S' a new final state t' annotated with c (unless there is already such a final state); add a copy of all of t's outgoing transitions to t'; and add the transition $(s, \mathtt{type}, d + c_5, t')$ to τ.
- (rule 5(ii)) For each transition $(s, a^-, d, t) \in \tau$ such that $s \in S_0$ and $(a, dom, c) \in K$, add to S' a new initial state s' annotated with c (unless there is already such an initial state); add a copy of all of s's incoming transitions to s'; and add the transition $(s', \mathtt{type}^-, d + c_5, t)$ to τ.
- (rule 6(i)) For each transition $(s, a, d, t) \in \tau$ such that $s \in S_0$ and $(a, range, c) \in K$, add to S' a new initial state s' annotated with c (unless there is already such an initial state); add a copy of all of s's incoming transitions to s'; and add the transition $(s', \mathtt{type}^-, d + c_6, t)$ to τ.
- (rule 6(ii)) For each transition $(s, a^-, d, t) \in \tau$ such that $t \in S'_f$ and $(a, range, c) \in K$, add to S' a new final state t' annotated with c (unless there is already such a final state); add a copy of all of t's outgoing transitions to t'; and add the transition $(s, \mathtt{type}, d + c_6, t')$ to τ.

Having constructed the relaxed automaton M_Q^K, its product automaton with the closure of the graph G is then constructed, and the computation proceeds similarly to cases (i) and (ii) for computing approximate answers above, except that in (i) if X is a class c then the shortest path traversal starts from all initial states (s_0, c') such that c' is a superclass of c. The evaluation can again be accomplished 'on-demand'

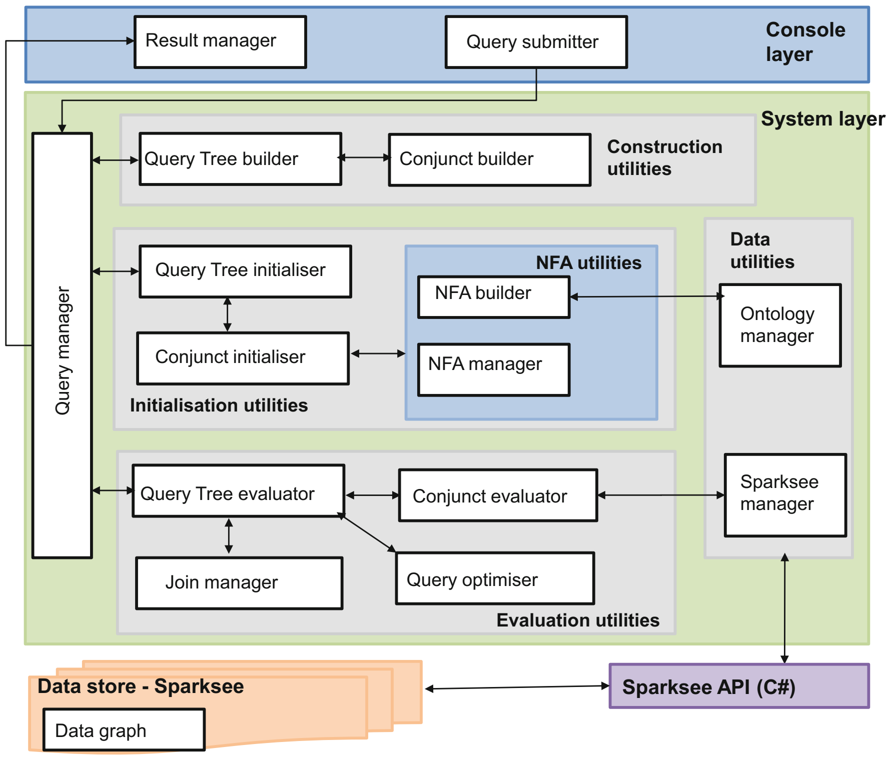


Fig. 4.10 Omega system architecture

by incrementally constructing the edges of the product automaton. The same data structures and algorithms as for computing approximate answers can be used, the only difference being that the `Succ` function now uses the automaton M_Q^K.

4.3.3 System Architecture and Performance

A prototype implementation of combined approximation and relaxation for CRPQs, called *Omega*, is described in Selmer et al. (2015) and Selmer (2016). Sparksee[7] is used as the data store. The development was undertaken using the Microsoft .NET framework. The system comprises four components (see Fig. 4.10): (1) the *console* layer, to which queries are submitted and which displays the incrementally computed query results; (2) the *system* layer in which query plans are constructed,

[7]http://www.sparsity-technologies.com, accessed at 18/6/2015.

optimised and executed; (3) the Sparksee API, which provides an interface for invoking data access methods to the data store; and (4) the data store itself.

Query evaluation commences when *Query submitter* invokes *Query manager*, passing it a CRPQ query that is to be evaluated. *Query manager* invokes *Query Tree builder* to construct the query tree, comprising inner nodes representing join operators and leaf nodes representing individual query conjuncts. *Query Tree builder* calls *Conjunct builder* to construct each leaf node of the query tree. *Query manager* next passes the query tree to *Query Tree initialiser*, which traverses the query tree in a top-down manner, beginning at the root. Whenever *Query Tree initialiser* encounters a leaf node in the query tree, it invokes *Conjunct initialiser* on that conjunct. This in turn invokes *NFA builder* to construct the automaton corresponding to the conjunct's regular expression. If the conjunct is approximated or relaxed, then *NFA manager* is invoked to produce an approximate or relaxed automaton, with the relevant edit or relaxation operators applied. For construction of a relaxed automaton, *NFA manager* interacts with *Ontology manager*, which stores the extended reduction of the ontology. *Query manager* then invokes *Query Tree evaluator*. This first invokes the *Query optimiser* to transform the query tree into its final form for execution (see below for a discussion of optimisation). *Query Tree evaluator* then traverses the optimised query tree, starting from the leftmost leaf node, and proceeding upwards. If the current query tree node is a leaf, the ranked answers for the query conjunct are computed by invoking *Conjunct evaluator*. This module constructs the weighted product automaton, H, of the conjunct's automaton with the (closure of) the data graph G. The construction of H is incremental, with *Conjunct evaluator* invoking *Sparksee manager* to retrieve only those nodes and edges of G that are required in order to compute the next batch of k results (for some predefined value of k, default 100). If the current query tree node is a join, *Query Tree evaluator* works in conjunction with *Join manager* to perform a ranked join of the answers returned thus far by its two children nodes. The join algorithm used is that described in Hurtado et al. (2009b), itself adapted from Ilyas et al. (2004). Once the root of the query tree has been reached, the processing terminates and the list of answers now holds the next k results, ranked by increasing distance. *Query manager* passes this list to *Result manager*, which displays the results in ranked order.

For constructing the automata, use is made of several data structures provided by the C5 Generic Collection library,[8] all of which have an amortised time complexity of O(1) for look-ups and updates:

- `HashSet`: a set of items (of some type T) implemented as a hash table with linear chaining
- `HashedLinkedList`: A linked list of items (of some type T) with an additional hash table to optimise item lookups within the list
- `HashDictionary`: A hash table of typed (key,value) pairs

[8]http://www.itu.dk/research/c5, accessed at 18/6/2015.

The `HashDictionary` data structure is used to implement the automata, where the key is an integer representing a 'from' state s, and the value is a `HashedLinkedList` of tuples representing the transitions outgoing from s. The priority queue $\texttt{queue}_R$ is also implemented by a `HashDictionary`. The key is an integer–boolean variable (where the integer portion represents a distance and the boolean portion represents the final or non-final tuples at that distance). The value associated with each key, implemented using a `HashedLinkedList`, comprises tuples of the form (v, n, s, d, f), ordered by increasing values of d, where d is the distance associated with visiting node n in state s having started from node v, and f is a flag denoting whether the tuple is 'final' or 'non-final'. Distinguishing between these two kinds of tuples in the priority queue allows the removal of 'final' tuples to be prioritised, so that answers may be returned earlier.

Readers are referred to Selmer et al. (2015) and Selmer (2016) for further details of the implementation and physical optimisations of the *Omega* system. Those works also report on a performance study of regular path queries with approximation and relaxation on several datasets sourced from the L4All system and from the SIMPLETAX and CORE portions of YAGO (Kasneci et al. 2009). The L4All data graphs used in the performance study were of size up to 220.8 MB for the closure of the data graph while the size of the closure of the YAGO data graph was 1.76 GB. Most of the APPROX and RELAX queries executed quickly on all datasets. However, some of the APPROX queries on YAGO either failed to terminate or did not complete within a reasonable amount of time. This was mainly due to a large number of intermediate results being generated, due to the *Succ* function returning a large number of transitions which are then converted into tuples in *GetNext* and added to $\texttt{queue}_R$. Some optimisations are explored in Selmer et al. (2015) and Selmer (2016) for such queries, enabling several—but not all—of the APPROX queries to execute faster. Future work includes making use of disk-resident data structures for $\texttt{queue}_R$ to guarantee the termination of APPROX queries with large intermediate results, and using knowledge of the graph structure (e.g. to prioritise the evaluation of rarer paths within the graph) to reduce the amount of unnecessary processing. Another promising direction is to identify labels that are rare in the graph and to split the processing of a regular expression into smaller fragments whose first or last label is a rare label, as described in Koschmieder and Leser (2012) (but not for approximated/relaxed queries).

4.4 $SPARQL^{AR}$: Extending SPARQL with Approximation and Relaxation

Relaxation of triple patterns and approximate matching of regular RPQs can be applied to the more pragmatic setting of the SPARQL 1.1 query language (Harris and Seaborne 2013). SPARQL is the predominant language for querying RDF data and, in the latest extension to SPARQL 1.1, it supports RPQs over the RDF graph (known as ‘property path queries’). However, it does not support notions of query approximation and relaxation, other than the OPTIONAL operator. Users querying complex RDF datasets may lack full knowledge of the structure of the data, its irregularities, and the URIs used within it. The schemas and URIs used can also evolve over time. This may make it difficult for users to formulate queries that precisely express their information retrieval requirements. Calì et al. (2014) and Frosini et al. (2017) investigate extensions to various fragments of SPARQL 1.1 to allow query approximation and relaxation. These works show that the introduction of the query approximation and query relaxation operators does not increase the complexity class of the language fragments studied, and complexity bounds for several fragments are derived. The extended language is called SPARQL^{AR}.

4.4.1 Example: Flexible Querying of RDF/S Knowledge Bases

Example 4.1 Suppose the user wishes to find the geographic coordinates of the ‘Battle of Waterloo’ event by posing the following query on the YAGO knowledge base,[9] which is derived from multiple sources such as Wikipedia, WordNet and GeoNames:

```
PREFIX yago:<http://yago-knowledge.org/resource/>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
  <http://yago-knowledge.org/resource/Battle_of_Waterloo>
  yago:happenedIn/(yago:hasLongitude|yago:hasLatitude)
  ?x }
```

This query uses the *property paths* extension of SPARQL 1.1, including its concatenation (/) and disjunction (|) operators. The above query does not return any answers since YAGO does not store the geographic coordinates of Waterloo.

The user may therefore choose to approximate the triple pattern in their query:

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

YAGO does store directly the coordinates of the ‘Battle of Waterloo’ event. So the system can apply an edit operation that deletes `happenedIn` from the property path, and the resulting query

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

[9]http://www.mpi-inf.mpg.de/yago-naga/yago/.

returns the desired answers, showing both high precision and high recall:

```
"4.4"^^<degrees>
"50.6833333333333"^^<degrees>
```

Example 4.2 Consider the following portion $K = (N_K, E_K)$ of the YAGO ontology, where N_K is

$$\{hasFamilyName, hasGivenName, label, actedIn, Actor\}$$

and E_K is

$$\{(hasFamilyName, sp, label), (hasGivenName, sp, label), \\ (actedIn, domain, actor)\}$$

Suppose the user is looking for the family names of all actors who played in the film 'Tea with Mussolini' and poses this query:

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

The above query returns only four answers, since some actors have only a single name (e.g. Cher), while others have their full name recorded using the `label` property.

The user may choose to relax the second triple pattern in their query 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 ) }
```

The system can now replace `hasFamilyName` by `label`, and the resulting query returns the given names of actors in that film recorded through the property `hasGivenName` (hence returning Cher), as well as actors' full names recorded through the property `label`: a total of 255 results.

Example 4.3 Suppose a user wishes to find events that took place in Berkshire in 1643 and poses the following query on YAGO (in the query, we use 'Event' for simplicity but the actual URI is `<wordnet_event_100029378>`):

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

This query returns no results because there are no property edges named `on` or `in` in YAGO.

The user may choose to approximate the second and third triple patterns of their query:

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

The system can now substitute `on` by `happenedOnDate` (which does appear in YAGO) and `in` by `happenedIn`, giving the following query:

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

This still returns no answers, since `happenedIn` does not connect event instances directly to literals such as `"Berkshire"`.

The user can choose to relax now the third triple pattern of the above query:

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

The system can replace the triple `?x yago:happenedIn "Berkshire"` by the triple `?x rdf:type Event`, using knowledge encoded in YAGO that the domain of `happenedIn` is `Event`, giving the following query, which returns all events recorded as occurring in 1643:

```
SELECT * WHERE {
    ?x rdf:type Event .
    ?x yago:happenedOnDate "1643-##-##" .
    ?x rdf:type Event }
```

Several answers are returned by this query, including the 'Siege of Reading' that happened in 1643 in Berkshire, but also several events that did not happen in Berkshire:

```
<http://yago-knowledge.org/resource/Battle_of_Olney_Bridge>
<http://yago-knowledge.org/resource/Battle_of_Heptonstall>
<http://yago-knowledge.org/resource/Siege_of_Reading>
<http://yago-knowledge.org/resource/Torstenson_War>
<http://yago-knowledge.org/resource/Battle_of_Alton>
<http://yago-knowledge.org/resource/Second_Battle_of_Middlewich>
<http://yago-knowledge.org/resource/Kieft's_War>
```

So the query exhibits better recall than the original query, but possibly low precision.

The user can instead choose to approximate the third triple pattern:

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

The system can now insert the property `label` that connects URIs to their labels, giving the following query:

```
SELECT * WHERE {
    ?x rdf:type Event .
    ?x yago:happenedOnDate "1643-##-##" .
    ?x yago:happenedIn/label "Berkshire" }
```

This query now returns the only event recorded as occurring in 1643 in Berkshire, i.e. the ‘Siege of Reading’. It exhibits both better recall than the original query and also high precision.

4.4.2 Query Rewriting-Based Implementation Approach

For specifying the semantics of SPARQL^{AR} queries, we extend the notion of SPARQL query evaluation from returning a set of (exact) mappings to returning a set of mapping/cost pairs of the form $\langle \mu, c \rangle$, where μ is a mapping and c is a non-negative number that indicates the cost of the answers arising from this mapping. Following on from the definitions of sets V, U and L, triples and triple patterns in Sect. 4.1, we have the following definitions (c.f. Pérez et al. 2006):

Definition 4.4 (Mapping) A *mapping* μ from ULV to UL is a partial function $\mu : ULV \rightarrow UL$ such that $\mu(x) = x$ for all $x \in UL$, i.e. μ maps URIs and literals to themselves. The set $var(\mu)$ is the subset of V on which μ is defined. Given a triple pattern $\langle x, z, y \rangle$ and a mapping μ such that $var(\langle x, z, y \rangle) \subseteq var(\mu)$, $\mu(\langle x, z, y \rangle)$ is the triple obtained by replacing the variables in $\langle x, z, y \rangle$ by their image according to μ.

Definition 4.5 (Compatibility and Union of Mappings) Two mappings μ_1 and μ_2 are *compatible* if $\forall x \in var(\mu_1) \cap var(\mu_2)$, $\mu_1(x) = \mu_2(x)$. The *union* of two mappings $\mu = \mu_1 \cup \mu_2$ can be computed only if μ_1 and μ_2 are compatible. The resulting μ is a mapping such that $var(\mu) = var(\mu_1) \cup var(\mu_2)$ and: for each x in $var(\mu_1) \cap var(\mu_2)$, $\mu(x) = \mu_1(x) = \mu_2(x)$; for each x in $var(\mu_1)$ but not in $var(\mu_2)$, $\mu(x) = \mu_1(x)$; and for each x in $var(\mu_2)$ but not in $var(\mu_1)$, $\mu(x) = \mu_2(x)$.

The *union* of two sets of SPARQL^{AR} query evaluation results, $M_1 \cup M_2$, comprises the following set of mapping/cost pairs:

$$\{\langle \mu, c \rangle \mid \langle \mu, c_1 \rangle \in M_1 \text{ or } \langle \mu, c_2 \rangle \in M_2, \text{ with } c = c_1 \text{ if } \nexists c_2.\langle \mu, c_2 \rangle \in M_2, c = c_2 \text{ if } \nexists c_1.\langle \mu, c_1 \rangle \in M_1, \text{ and } c = min(c_1, c_2) \text{ otherwise}\}.$$

In Calì et al. (2014) and Frosini et al. (2017) a *query rewriting* approach is adopted for SPARQL^{AR} query evaluation, in which a SPARQL^{AR} query Q is rewritten to a set of SPARQL 1.1 queries for evaluation. We summarise this approach here, refering the reader to those papers for further details.

To keep track of which triple patterns in Q need to be relaxed or approximated, such triple patterns are labelled with A for approximation and R for relaxation. The query rewriting algorithm starts by generating a query Q_0 which returns the exact answer of Q, i.e. ignoring any APPROX and RELAX operators. For each triple pattern $\langle x_i, R_i, y_i \rangle$ in Q_0 labelled with A or R, and each URI p that appears in R_i, a set of new queries is constructed by applying all possible one-step edit operations or relaxation operations to p (these are the 'first-generation' queries). To each such 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 or relaxation to each query Q_1 (the 'second-generation' queries), accumulating summatively the cost of the two edit or relaxation operations applied to obtain each query and assigning this cost to the query. The process continues for a bounded number of generations, accumulating summatively the cost of the sequence of edit or relaxation operations applied to obtain each query in the ith generation. The rewriting process terminates once the cost of all the queries generated in a generation has exceeded a maximum value m.

The overall query evaluation algorithm is defined below, where *QRA* denotes the Query Rewriting Algorithm and it is assumed that the output set, M, of mapping/cost pairs is maintained in order of increasing cost, e.g. as a priority queue. Ordinary SPARQL query evaluation—denoted *SPARQLeval* in the algorithm—is applied to each query generated by *QRA*, in ranked order of the query costs. *SPARQLeval* takes as input a SPARQL query Q' and a graph G and returns a set of (exact) mappings. The mappings are then assigned the cost of the query Q'. If a mapping is generated more than once, only the one with the lowest cost is retained in M (by the semantics of the union operator, $\cup$, applied to sets of mapping/cost pairs).

Algorithm 7: SPARQL^{AR} flexible query evaluation

```
input  : Query $Q$; maximum cost $m$; Graph $G$; Ontology $K$.
output: List of mapping/cost pairs, $M$, sorted by cost.
$M := \emptyset$;
foreach $\langle Q', cost \rangle \in QRA(Q,m,K)$ do
    foreach $\mu \in SPARQLeval(Q',G)$ do
        $M := M \cup \{\langle \mu, cost \rangle\}$
return M;
```

A formal study of the correctness and termination of the Query Rewriting Algorithm can be found in Frosini et al. (2017) where the Rewriting Algorithm itself is also specified in detail.

4.4.3 System Architecture and Performance

A prototype implementation of SPARQL^{AR} is described in Frosini et al. (2017). The implementation is in Java and Jena is used for the SPARQL query execution.

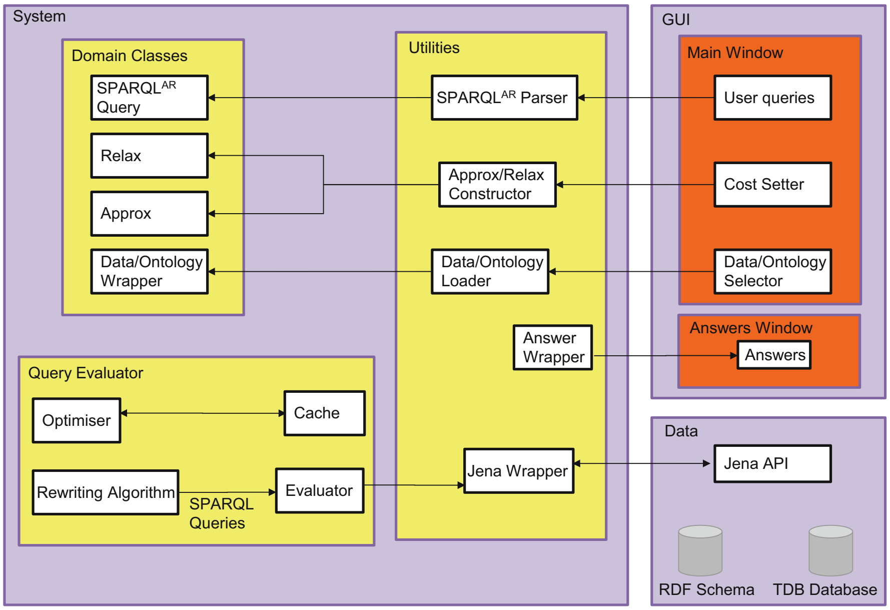


Fig. 4.11 SPARQL^{AR} system architecture

Figure 4.11 illustrates the system architecture, consisting of three layers: the GUI layer, the System layer, and the Data layer.

The GUI layer supports user interaction with the system, allowing queries to be submitted, costs of the edit and relaxation operators to be set, datasets and ontologies to be selected, and query answers to be incrementally displayed to the user.

The System layer is responsible for the processing of the SPARQL^{AR} queries. It comprises three components: the Utilities, containing classes providing the core logic of the system; the Domain Classes, providing classes relating to the construction of SPARQL^{AR} queries; and the Query Evaluator in which query rewriting, optimisation and evaluation are undertaken.

The Data layer connects the system 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. The RDF datasets are stored as a TDB database[10] and the RDF schema can be stored in multiple RDF formats (e.g. Turtle, N-Triple, RDF/XML).

When a user query is submitted to the GUI, this invokes a method of the *SPARQL*^{AR} *Parser* to parse the query string and construct an object of the class *SPARQL*^{AR} *Query*. The GUI also invokes the *Data/Ontology Loader* which creates

[10]https://jena.apache.org/documentation/tdb/.

an object of the class *Data/Ontology Wrapper*, and the *Approx/Relax Constructor* which creates objects of the classes *Approx* and *Relax*.

Once these objects have been initialised, they are passed to the Query Evaluator by invoking the *Rewriting Algorithm*. This generates the set of SPARQL queries to be executed over the RDF dataset. The set of queries are passed to the *Evaluator*, which interacts with the *Optimiser* and the *Cache* to improve query performance. Specifically, the answers of parts of queries are computed and stored in the *Cache*, and these answers are retrieved from the *Cache* when the *Evaluator* needs these results. A SPARQL^{AR} query is first split into two parts: the triple patterns which do not have APPROX or RELAX applied to them (the exact part) and those which have (the A/R part). The exact part is first evaluated and the results are cached. The query rewriting algorithm is then applied to the A/R part. Each triple pattern generated is evaluated individually, as also are all possible pairs of triple patterns, and the answers for each evaluation are cached. To avoid memory overflow, an upper limit is placed on the size of the cache. Finally, the overall results of a SPARQL query are obtained by joining subquery results already cached with those obtained by evaluating the rest of the query.

The *Evaluator* uses the *Jena Wrapper* to invoke Jena library methods for executing SPARQL queries over the RDF dataset. The *Jena Wrapper* also gathers the query answers and passes them to the *Answer Wrapper*. Finally, the answers are displayed by the *Answers Window*, in ranked order.

A performance study using data generated from the Lehigh University Benchmark (LUBM)[11] is described in Calì et al. (2014). Three datasets were generated, the largest of which contained 673,416 triples (65 MB). A larger-scale performance study on the YAGO dataset is described in Frosini et al. (2017). YAGO contains over 120 million triples which were downloaded and stored in a Jena TDB database. The size of the TDB database was 9.70 GB, and the nodes of the YAGO graph were stored in a 1.1 GB file.

The overall results show that the evaluation of SPARQL^{AR} queries through a query rewriting approach is promising (see Calì et al. 2014; Frosini et al. 2017 for details). The difference between the execution time of the exact form and the APPROXed/RELAXed forms of the queries is acceptable for queries with fewer than five conjuncts. For most of the other queries that were trialled, the simple caching technique described above also brings down the run times of their APPROXed/RELAXed forms to more 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.

Our ongoing work involves investigating optimisations to the query rewriting algorithm, since this can generate a large number of queries. In particular, we are studying the query containment problem for SPARQL^{AR} and how query costs impact on this. For example, for a query $Q = Q_1$ AND Q_2 it is possible to decrease the number of queries generated by the rewriting algorithm if we know

[11] http://swat.cse.lehigh.edu/projects/lubm/.

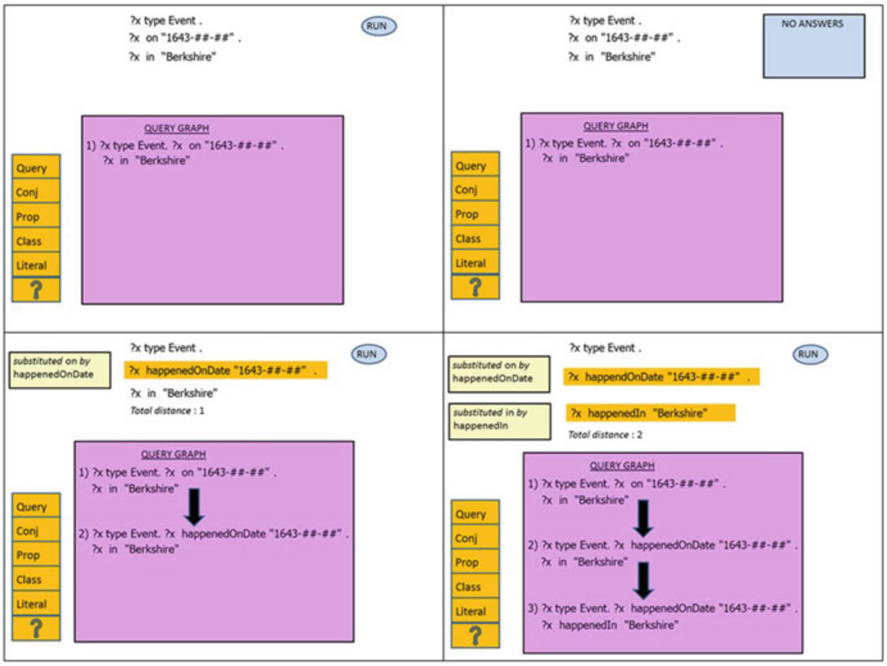
that $Q_1 \subseteq Q_2$, in which case we can evaluate Q_1 rather than Q. Other optimisations under investigation include using statistics about path frequencies in the data graph to reorder the evaluation of triple patterns so as to evaluate first those returning fewer results; and using summaries of the data graph to avoid evaluating subqueries that we know, after evaluation on the graph summary, cannot return any answers. Also planned is a detailed comparison of this query rewriting approach to query approximation and relaxation with the 'native' implementation of Omega described in the previous section.

4.5 Further Topics

4.5.1 User Interaction

In Sect. 4.3.1 we briefly discussed an application-specific prototype that provides a forms-based GUI for incrementally generating CRPQ queries, parts of which can optionally be approximated or relaxed, and for displaying ranked query results to the user. A detailed discussion of that prototype can be found in Poulovassilis et al. (2012). An area of future work identified in that paper was how such systems might provide explanations to the user of how the overall 'distance' of each query result has been derived, based on the application of a sequence of edit and relaxation operations each of some cost specified by the user.

One possible visualisation for such explanations, in a more generic setting, is the Query Graph illustrated in Figs. 4.12 and 4.13, which is based on an 'inverted' version of the relaxation graph for graph patterns discussed in Sect. 4.1. To illustrate, consider Example 4.3 from Sect. 4.1 which is enacted in successive screenshots in the two figures, moving from left-to-right and top-to-bottom. The user begins (Screen 1) by constructing their initial query, which is shown both in the main pane and in the Query Graph panel below. The user then presses the RUN button to run the query. However, no answers are returned (Screen 2). The user elects to edit the second triple pattern, by clicking on that pattern and then on the 'Conj[unct]' button, selecting 'substitution' from a drop-down list of edit operations displayed by the system, and then `happenedOnDate` from a list of properties suggested by the system (e.g. properties that are known to have domain `Event`). Screen 3 shows the new query and its distance from the original one, the updated Query Graph and—in the top-left—the edit operations applied so far. The user presses RUN but again no answers are returned. The user elects to edit now the third triple pattern, again selecting 'substitution' from a drop-down list of edit operations, and now `happenedIn` from the list of properties suggested by the system. Screen 4 shows the new query and the updated Query Graph. The user presses RUN but again no answers are returned. At this point, the user seeks help by clicking on the "?' button and system suggests three alternatives: (i) relaxation of the second triple pattern to `?x rdf:type Event`, (ii) relaxation of the third triple pattern



```
?x type Event .
?x on "1643-##-##" .
?x in "Berkshire"

QUERY GRAPH
1) ?x type Event. ?x on "1643-##-##" .
   ?x in "Berkshire"
```

```
?x type Event .
?x on "1643-##-##" .
?x in "Berkshire"

QUERY GRAPH
1) ?x type Event. ?x on "1643-##-##" .
   ?x in "Berkshire"
```

```
substituted on by happenedOnDate
?x type Event .
?x happenedOnDate "1643-##-##" .
?x in "Berkshire"
Total distance : 1

QUERY GRAPH
1) ?x type Event. ?x on "1643-##-##" .
   ?x in "Berkshire"
2) ?x type Event. ?x happenedOnDate "1643-##-##" .
   ?x in "Berkshire"
```

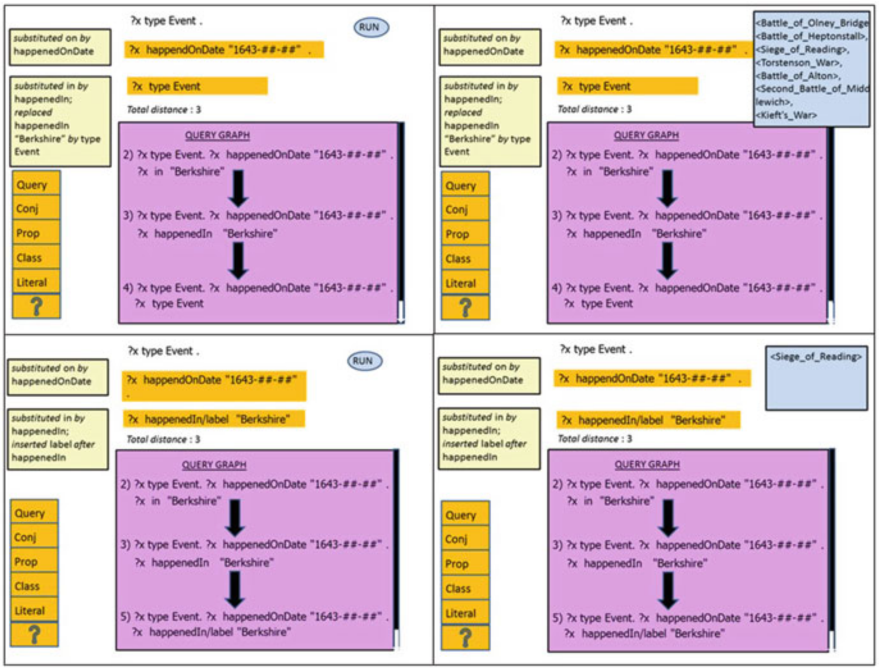
```
substituted on by happenedOnDate
substituted in by happenedIn
?x type Event .
?x happendOnDate "1643-##-##" .
?x happenedIn "Berkshire"
Total distance : 2

QUERY GRAPH
1) ?x type Event. ?x on "1643-##-##" .
   ?x in "Berkshire"
2) ?x type Event. ?x happenedOnDate "1643-##-##" .
   ?x in "Berkshire"
3) ?x type Event. ?x happenedOnDate "1643-##-##" .
   ?x happenedIn "Berkshire"
```

Fig. 4.12 User interaction and visualisation

to `?x rdf:type Event`, (iii) insertion of a literal-valued property to follow `happenedIn`. Suppose the user chooses option (ii). Screen 5 shows the new query and the updated Query Graph (scrolling down now in the expanded Query Graph pane). The user presses RUN and Screen 6 illustrates the results returned (all events taking place in 1643, at any location). The user decides these results are too diverse to be useful and backtracks to Query 3, where the system provides again alternatives (i)–(iii). Suppose the user now chooses option (iii). The system offers a list of literal-valued properties (those with domain `Place`, or a superclass, on the basis of knowledge that this is the range of `happenedIn`), and the user selects the property `label`. Screen 7 shows the new query and the updated Query Graph. The user presses RUN and Screen 8 shows the result returned, which is the one event recorded as occurring in Berkshire in 1643.

Allowing the user to visualise how queries are incrementally generated, what distance is associated with each query, and what results are returned, if any, can help the user decide whether the answers being returned are useful and to try out different edit/relaxation operations. Detailed design, implementation and evaluation of such interactive flexible querying facilities and visualisations for end-users are an area requiring further work.



?x type Event .

substituted on by happenedOnDate

?x happendOnDate "1643-##-##" .

substituted in by happenedIn; replaced happenedIn "Berkshire" by type Event

?x type Event

Total distance : 3

QUERY GRAPH

2) ?x type Event. ?x happenedOnDate "1643-##-##" . ?x in "Berkshire"

3) ?x type Event. ?x happenedOnDate "1643-##-##" . ?x happenedIn "Berkshire"

4) ?x type Event. ?x happenedOnDate "1643-##-##" . ?x type Event

<Battle_of_Olney_Bridge
<Battle_of_Heptonstall>,
<Siege_of_Reading>,
<Torstenson_War>,
<Battle_of_Alton>,
<Second_Battle_of_Midd
lewich>,
<Kieft's_War>

?x type Event .

substituted on by happenedOnDate

?x happendOnDate "1643-##-##" .

substituted in by happenedIn; inserted label after happenedIn

?x happenedIn/label "Berkshire"

Total distance : 3

QUERY GRAPH

2) ?x type Event. ?x happenedOnDate "1643-##-##" . ?x in "Berkshire"

3) ?x type Event. ?x happenedOnDate "1643-##-##" . ?x happenedIn "Berkshire"

5) ?x type Event. ?x happenedOnDate "1643-##-##" . ?x happenedIn/label "Berkshire"

<Siege_of_Reading>

Fig. 4.13 User interaction and visualisation

4.5.2 *More Query Flexibility*

There are several directions in which the approaches discussed in the previous sections can be extended. One area of ongoing work is to merge the APPROX and RELAX operators into one integrated FLEX operator that *simultaneously* applies edit and relaxation operations to a regular path query. This would allow greater ease of querying for users, in that they would not need to be aware of the ontology structure and to identify which conjuncts of their query may be amenable to relaxation and which to approximation. Another ongoing direction is to extend our languages with lexical and semantic *similarity measures*, in order to allow approximate matching of literals and resources.

To illustrate, suppose a History of Science researcher wishes to find scientists born in London. She is also interested in scientists living in or near London. However, she doesn't know how YAGO records that a person is a scientist, and poses the following query that makes use of the `hasGloss` property linking resources to textual descriptions, seeking to find the word ‘scientist’ or similar within such descriptions:

```
SELECT * WHERE {
    FLEX (?p yago:wasBornIn London) .
    ?p rdf:type ?c . ?c yago:hasGloss ?descr .
    FILTER sim (?descr, "scientist") > 0.7}
```

The system can find matches for ‘scientist’ within values of the `hasGloss` property (e.g. as in the descriptions ‘person with advanced knowledge of one or more sciences’, ‘an expert in the science of economics’), allowing relevant answers to be returned. Use of the FLEX operator also allows the system to substitute `wasBornIn` by `livesIn`, giving additional answers of relevance.

Similarity measures could also be applied by the system to distinguish between different alternatives when an edge label in a regular path query is being substituted by a different label. Having knowledge of the semantic similarly of properties—for example, exploiting dictionaries such as Wordnet—would allow the system to assign a finer-grained cost to edge label substitutions, thereby allowing finer ranking of the top-k answers and increasing their precision.

Similarly, it would be possible to assign a finer ranking to the application of relaxation operations where a specific property is replaced by superproperty, or a specific class by a superclass.

4.5.3 More Query Expressivity

Another direction of work is to extend the expressivity of the query languages beyond conjunctive regular path queries. In this direction, Hurtado et al. (2009a) investigated approximate matching of *extended regular path* (ERP) queries in which the regular expression R in a query conjunct (X, R, Y) can be associated with a path variable P—using the syntax $(X, R : P, Y)$—and these path variables can appear also in the query head, thereby allowing graph paths to be returned to the user as part of the query answer. It was shown that top-k approximate answers can be returned in polynomial time in the size of the graph and the query. Thus, for example, revisiting the transport network example in Sect. 4.2.1, the following query finds cities reachable from Santiago by train, directly or indirectly, or by combinations of other modes of transport, returning also the routes:

```
  ?P, ?X <- APPROX (u1, train+.name:?P ,?X)
```

The answers returned are:
`([train,u2,name],Talca),`
`([train,u2,train,u4,name],Chillan)`
at distance 0;
`([bus,u3,name],Valparaiso),`
`([airplane,u7,name],Temuco),`
`([airplane,u4,name],Chillan)`
at distance c_3;
`([airplane,u4,airplane,u5,name],Concepcion),`
`([airplane,u4,bus,u5,name],Concepcion),`

`([airplane,u4,bus,u6,name],Los Angeles)`
at distance $c_3 + c_4$; and so forth.

Another application of this kind of flexible querying, where it is useful to return paths in the query results, is described in Poulovassilis et al. (2015), which discusses the analysis of user–system interaction data as arising from exploratory learning environments. The interaction data is stored in Neo4j. Interaction events, and their types, are represented by nodes. Events are linked to their event-type by edges labelled OCCURRENCE_OF while successive events are linked to each other by edges labeled NEXT. Poulovassilis et al. (2015) give examples of how approximate matching of ERP queries over such data can allow pedagogical experts to investigate how students are undertaking exploratory learning tasks, and how feedback messages generated by the system are affecting students' behaviours, with the aim of designing improved support for students. For example, the following query (expressed in Neo4j's Cypher language[12]) finds pairs of events x, y such that x is an intervention (i.e. a message) generated by the system and y is the user's next action; the path between x and is returned, through the variable p, as are the event-type of x and y, through the variables v and w.

```
MATCH (x:Event)-[:OCCURRENCE_OF]->
                (v:EventType {event_cat:"intervention"}),
       p = (x:Event)-[:NEXT]->(y:Event),
      (y:Event)-[:OCCURRENCE_OF]->(w:EventType)
RETURN v.event_type as start_node_type,
       extract(n IN nodes(p) | n.id_fltask) as path_node_ids,
       w.event_type as end_node_type
```

Results returned include:

start_node_type	path_node_ids	end_node_type
highMessage	["344509","344510"]	ClickButton
highMessage	["344519","344520"]	ClickButton
highMessage	["344522","344523"]	ClickButton
highMessage	["344714","344715"]	ClickButton
highMessage	["344760","344761"]	ClickButton

If query approximation were supported in Cypher, then applying APPROX to the subquery `(x:Event)-[:NEXT]->(y:Event)` above, and enabling just insertion of edge labels, would generate the subquery `(x:Event)-[:NEXT*2]->`

[12]http://neo4j.com, accessed at 18/6/2015.

`(y:Event)`. Evaluation of the new query would return answers such as the following, all at distance 1 from the original query:

start_node_type	path_node_ids	end_node_type
highMessage	["344509","344510","346027"]	ClickButton
highMessage	["344519","344520","344521"]	PlatformEvent
highMessage	["344522","344523","344712"]	ClickButton
highMessage	["344714","344715","344716"]	ClickButton
highMessage	["344760","344761","344762"]	PlatformEvent

Following this, the subquery `(x:Event)-[:NEXT*2]->(y:Event)` could be automatically approximated again to `(x:Event)-[:NEXT*3]->(y:Event)`. Evaluation of the new query would return answers such as the following, now at edit distance 2 from the original query:

start_node_type	path_node_ids	end_node_type
highMessage	["344509","344510","346027","346028"]	FractionGenerated
highMessage	["344519","344520","344521","344522"]	highMessage
highMessage	["344522","344523","344712","344713"]	FractionGenerated
highMessage	["344714","344715","344716","344717"]	FractionChange
highMessage	["344760","344761","344762","344763"]	highMessage

By this point, the pedagogical expert is able to see that some high-level interruption messages (‘highMessage’) are leading students towards productive behaviours, such as generating a fraction—lines 1 and 3, or changing a fraction—line 4 (the specific learning environment to which this data relates aims to teach young learners about fractions). However, other messages are just resulting in more messages being generated by the system (lines 2 and 5), which may lead experts to explore the data further (e.g. to retrieve the messages associated with events 344519 and 344760) and possibly reconsider this part of the system’s design.

4.6 Related Work

A general overview of graph databases from the perspectives of graph characteristics, graph data management, applications and benchmarking can be found in Larriba-Pey et al. (2014). The work described in this chapter has considered only the simple graph data model introduced in Sect. 4.1, and a broader survey of graph data models can be found in Angles and Gutierrez (2008). Likewise, a survey of graph query languages can be found in Wood (2012), and we focus here on languages that support RPQs and on flexible query processing for graph-structured data.

Using regular expressions to specify path queries on graph-structured data has been studied for nearly 30 years, being introduced in the languages G, G+ and

Graphlog (Cruz et al. 1987; Consens and Mendelzon 1989; Mendelzon and Wood 1989, 1995) and taken up in several languages for other semi-structured data models (Abiteboul et al. 1997; Fernandez et al. 2000; Buneman et al. 2000). More recently, CRPQs are supported in NAGA (Kasneci et al. 2009), SPARQLeR (Kochut and Janik 2007), PSPARQL (Alkhateeb et al. 2009), G-SPARQL (Sakr et al. 2012) and SPARQL 1.1 (Harris and Seaborne 2013). Cypher, the declarative query language supported by the Neo4j graph DBMS, also supports a restricted form of regular path queries. The nSPARQL language (Pérez et al. 2008) extends SPARQL with *nested* regular expressions and shows that these enable query answers that encompass the semantics of the RDFS vocabulary by direct graph traversal, without materialising the closure of the graph. In addition to the automaton-based approach described in Sect. 4.3, other approaches proposed for evaluating (exact) CRPQs include translation into Datalog or recursive SQL (Consens and Mendelzon 1993; Wood 2012; Dey et al. 2013), search-based processing (Fan et al. 2011; Koschmieder and Leser 2012) and reachability indexing (Gubichev et al. 2013).

Recent work in the WAVEGUIDES project is investigating cost-based optimisation for SPARQL 1.1, focusing in the first instance on query optimisation for property paths (Yakovets et al. 2015), and this has potential application in the optimisation of approximated/relaxed CRPQs as well.

Early work on flexible querying for semi-structured data was undertaken by Kanza and Sagiv (2001), who considered matchings returning paths whose set of edge labels contain those appearing in the query; such semantics can be captured by transposition and insertion edit operations on edge labels. More generally, Grahne and Thomo (2001, 2006) explored approximate matching of single-conjunct regular path queries, using a weighted regular transducer to perform transformations to RPQs for approximately matching semi-structured data. This approach was extended in Hurtado et al. (2009b) to CRPQs. In other work, Grahne et al. (2007) introduced *preferential* RPQs where users can specify the relative importance of symbols appearing in the query by annotating them with weights.

The work in Barcelo et al. (2010, 2012) extends CRPQs to allow comparisons between path variables within the bodies of queries, as well as allowing path variables to appear in query heads, calling this extension *extended conjunctive regular path queries* (ECRPQs) (but not considering flexible querying). The work in Libkin and Vrgoc (2012) extends CRPQs to include manipulation also of the data values associated with nodes along a path. Other extensions to CRPQs are discussed in Wood (2012), for example with aggregation functions such as $count$, sum, max, min to allow finding properties of graphs that are useful for network analysis (e.g. in/out-degree of nodes, length of shortest paths between nodes, graph diameter). Extending these more expressive graph query languages with flexible querying capabilities is an open area. Also open is extending graph query languages for more complex graph models (e.g. property graphs, hyperedges, hypernodes—see Angles and Gutierrez 2008) with flexible queries.

There have been several proposals for flexibly querying Semantic Web data using *similarity measures* to retrieve additional relevant answers. For example, in iSPARQL (Kiefer et al. 2007) similarity measures are applied to resources; in

Hogan et al. (2012) similarity functions are applied to constants such as strings and numeric values; and in De Virgilio et al. (2013) a structural similarity approach is proposed that exploits the graph structure of the data. In other work, ontology-driven similarity measures are developed, using an RDFS ontology to retrieve additional answers and assign a score to them (Huang et al. 2008; Huang and Liu 2010; Reddy and Kumar 2010).

In Mandreoli et al. (2009) knowledge of the semantic relationships between graph nodes is used for approximate query matching, and Cedeno and Candan (2011) describe a framework for cost-aware querying of weighted RDF data through predicates that express flexible paths between nodes. Elbassuoni et al. (2009, 2011) propose extending SPARQL with keyword search capabilities, together with IR-style ranking of query answers. In Yang et al. (2014), a set of transformation functions are used to map attributes of nodes and edges appearing in a graph query to matches in the data graph, and a ranking model for query answers is learnt using automatically generated training instances and the query log.

Dolog et al. (2006, 2009) consider relaxing queries on RDF data based on user preferences; user preferences mined from the query log are also used for query relaxation in Meng et al. (2008); and flexible querying using preferences expressed as fuzzy sets is investigated in Buche et al. (2009).

Approximate *graph matching* has also been much studied (Zhang et al. 2010; Zhu et al. 2011; Zou et al. 2011; Fan et al. 2013; Ma et al. 2014), including adding regular expressions as edge constraints on the graph patterns to be matched (Fan et al. 2011) and ontology-based subgraph querying (Wu et al. 2013). This work has synergies with the flexible querying processing approaches discussed in this chapter, since the algorithms proposed could potentially be leveraged for improved query performance of approximated/relaxed CRPQs: this is currently an open area of research.

4.7 Concluding Remarks

We have given an overview of motivations, applications and implementation techniques for extending graph query languages with relaxation and approximation. Along the way we have highlighted directions of ongoing work, relating to providing additional flexibility through similarity matching, designing further logical and physical optimisations, and conducting more extensive performance studies. On the theory front, future work involves investigating the query containment problem for SPARQL^{AR} and the complexity implications of extending more expressive query languages with relaxation and approximation features. On the usability front, further work is required on designing user interfaces that allow users to control and visualise how flexible queries are incrementally generated and evaluated, so as to be able to decide whether the answers being returned are useful and to try out alternative relaxations or approximations.

Acknowledgements The author gratefully thanks Andrea Calì, Riccardo Frosini, Sergio Gutierrez-Santos, Carlos Hurtado, Manolis Mavrikis, Petra Selmer, Alex Wollenschlaeger and Peter Wood for our collaboration in the work described here.

References

Abiteboul S, Quass D, McHugh J, Widom J, Wiener J (1997) The LOREL query language for semistructured data. Int J Digit Libr 1(1):68–88

Aho AV, Hopcroft JE, Ullman JD (1974) The design and analysis of computer algorithms. Addison-Wesley, Reading

Alkhateeb F, Baget J, Euzenat J (2009) Extending SPARQL with regular expression patterns (for querying RDF). J Web Semantics 7(2):57–73

Almendros-Jimenez J, Luna A, Moreno G (2014) Fuzzy XPath queries in XQuery. In: Proceedings of OTM 2014, pp 457–472

Amer-Yahia S, Lakshmanan LVS, Pandit S (2004) FleXPath: flexible structure and full-text querying for XML. In: Proceedings of ACM SIGMOD 2004, pp 83–94

Angles R, Gutierrez C (2008) Survey of graph database models. ACM Comput Surv 40(1):1–39

Ayers R (1997) Databases for criminal intelligence analysis: knowledge representation issues. AI Soc 11:18–35

Babcock B, Chaudhuri S, Das G (2003) Dynamic sample selection for approximate query processing. In: Proceedings of ACM SIGMOD 2003, pp 539–550

Barcelo P, Hurtado CA, Libkin L, Wood PT (2010) Expressive languages for path queries over graph-structured data. In: Proceedings of PODS 2010, pp 3–14

Barcelo P, Libkin L, Lin AW, Wood PT (2012) Expressive languages for path queries over graph-structured data. ACM Trans Database Syst 37(4):1–46

Batini C, Lenzerini M, Navathe SB (1986) A comparative analysis of methodologies for database schema integration. ACM Comput Surv 18(4):323–364

Bordogna G, Psaila G (2008) Customizable flexible querying in classical relational databases. In: Handbook of research on fuzzy information processing in databases. IGI Global, Hershey, pp 191–217

Bosc P, Pivert O (1992) Some approaches for relational databases flexible querying. J Intell Inf Syst 1(3):323–354

Bosc P, Hadjali A, Pivert O (2009) Incremental controlled relaxation of failing flexible queries. J Intell Inf Syst 33(3):261–283

Bray T et al (eds) (2008) Extensible markup language (XML) 1.0, W3C Recommendation

Buche P, Dibie-Barthelemy J, Chebil H (2009) SPARQL querying of web data tables driven by an ontology. In: Proceedings of FQAS 2009

Buneman P, Fernandez M, Suciu D (2000) A query language and algebra for semistructured data based on structural recursion. VLDB J 9(1):76–110

Buratti G, Montesi D (2008) Ranking for approximated XQuery full-text queries. In: Proceedings of BNCOD 2008, pp 165–176

Calì A, Frosini R, Poulovassilis A, Wood PT (2014) Flexible querying for SPARQL. In: Proceedings of ODBASE 2014 (OTM Conferences), pp 473–490

Calvanese D, Giacomo GD, Lenzerini M, Vardi MY (2000) Containment of conjunctive regular path queries with inverse. In: Proceedings of KR 2000, pp 176–185

Cedeno J, Candan KS (2011) R2DF framework for ranked path queries over weighted RDF graphs. In: Proceedings of WIMS 2011

Chakrabarti K, Garofalakis M, Rastogi R, Shim K (2001) Approximate query processing using wavelets. VLDB J 10(2–3):199–223

Chen AC, Gao S, Karampelas P, Alhajj R, Rokne J (2011) Finding hidden links in terrorist networks by checking indirect links of different sub-networks. In: Counterterrorism and open source intelligence, pp 143–158

Chu W, Yang H, Chiang K, Minock M, Chow G, Larson C (1996) CoBase: a scalable and extensible cooperative information system. J Intell Inf Syst 6(2/3):223–259

Consens M, Mendelzon AO (1989) Expressing structural hypertext queries in GraphLog. In: Proceedings of ACM hypertext 1989, pp 269–292

Consens M, Mendelzon AO (1993) Low complexity aggregation in Graphlog and Datalog. Theor Comput Sci 116(1–2):95–116

Cruz IF, Mendelzon AO, Wood PT (1987) A graphical query language supporting recursion. In: Proceedings of SIGMOD 1987, pp 323–330

de Freitas S, Harrison I, Magoulas G, Papamarkos G, Poulovassilis A, van Labeke N, Mee A, Oliver M (2008) L4All: a web-service based system for lifelong learners. In: The learning grid handbook: concepts, technologies and applications. The future of learning, vol 2. IOS Press, Amsterdam

De Virgilio R, Maccioni A, Torlone R (2013) A similarity measure for approximate querying over RDF data. In: Proceedings of EDBT/ICDT 2013 workshops, pp 205–213

Deo N (2004) Graph theory with applications to engineering and computer science. PHI Learning, New Delhi

Dey S, Cuevas-Vicenttin V, Kohler S, Gribkoff E (2013) On implementing provenance-aware regular path queries with relational query engines. In: Proceedings of EDBT 2013, pp 214–223

Dolog P, Stuckenschmidt H, Wache H (2006) Robust query processing for personalized information access on the semantic web. In: Proceedings of FQAS 2006

Dolog P, Stuckenschmidt H, Wache H, Diederich J (2009) Relaxing RDF queries based on user and domain preferences. J Intell Inf Syst 33(3):239–260

Droste M, Kuich W, Vogler H (2009) Handbook of weighted automata. Springer, Berlin

Eckhardt A, Hornicak E, Vojtas P (2011) Evaluating top-k algorithms with various sources of data and user preferences. In: Proceedings of FQAS 2011, pp 258–269

Elbassuoni S, Ramanath M, Schenkel R, Sydow M, Weikum G (2009) Language model-based ranking for queries on RDF-graphs. In: Proceedings of CIKM 2009, pp 977–986

Elbassuoni S, Ramanath M, Weikum G (2011) Query relaxation for entity-relationship search. In: Proceedings of ESWC 2011 (Part 2)

Fan W, Li J, Ma S, Tang N, Wu Y (2011) Adding regular expressions to graph reachability and pattern queries. In: Proceedings of ICDE 2011, pp 39–50

Fan W, Wang X, Wu Y (2013) Diversified top-k graph pattern matching. PVLDB 6(13):1510–1521

Fernandez M, Suciu D (1998) Optimizing regular path expressions using graph schemas. In: Proceedings of ICDE 1998, pp 14–23

Fernandez M, Florescu D, Levy A, Suciu D (2000) Declarative specification of web sites with strudel. VLDB J 9(1):38–55

Finger J, Polyzotis N (2009) Robust and efficient algorithms for rank join evaluation. In: Proceedings of SIGMOD 2009

Fink R, Olteanu D (2011) On the optimal approximation of queries using tractable propositional languages. In: Proceedings of ICDT 2011, pp 174–185

Frosini R, Calì A, Poulovassilis A, Wood PT (2017) Flexible query processing for SPARQL. Semantic Web 8(4):533–563

Galindo J, Medina J, Pons O, Cubero C (1998) A server for fuzzy SQL queries. In: Proceedings of FQAS 1998, pp 164–174

Goble CA, Stevens R (2008) State of the nation in data integration for bioinformatics. J Biomed Inform 41(5):687–693

Gottlob G, Leone N, Scarcello F (2001) The complexity of acyclic conjunctive queries. J ACM 43(3):431–498

Grahne G, Thomo A (2001) Approximate reasoning in semi-structured databases. In: Proceedings of KRDB 2001

Grahne G, Thomo A (2006) Regular path queries under approximate semantics. Ann Math Artif Intell 46(1–2):165–190
Grahne G, Thomo A, Wadge WW (2007) Preferentially annotated regular path queries. In: Proceedings of ICDT 2007, pp 314–328
Gubichev A, Bedathur S, Seufert S (2013) Sparqling kleene: fast property paths in rdf-3x. In: Proceedings of 1st international workshop on graph data management experiences and systems (GRADES'13)
Gutierrez C, Hurtado C, Mendelzon AO (2004) Foundations of Semantic Web Databases. In: Proceedings of PODS 2004, pp 95–106
Halevy A, Rajaraman A, Ordille J (2006) Data integration: the teenage years. In: Proceedings of VLDB 2006, pp 9–16
Harris S, Seaborne A (eds) (2013) SPARQL 1.1 Query Language, W3C Recommendation
Hayes P (ed) (2004) RDF Semantics, W3C Recommendation
Heer J, Agrawala M, Willett M (2008) Generalized selection via interactive query relaxation. In: Proceedings of CHI 2008, pp 959–968
Hill J, Torson J, Guo B, Chen Z (2010) Toward ontology-guided knowledge-driven XML query relaxation. In: Proceedings of 2nd international conference on computational intelligence, modelling and simulation (CIMSiM) 2010, pp 448–453
Hogan A, Mellotte M, Powell G, Stampouli D (2012) Towards fuzzy query relaxation for RDF. In: Proceedings of ISWC 2012, pp 687–702
Huang H, Liu C (2010) Query relaxation for star queries on RDF. In: Proceedings of WISE 2010, pp 376–389
Huang H, Liu C, Zhou X (2008) Computing relaxed answers on RDF databases. In: Proceedings of WISE 2008, pp 163–175
Hurtado CA, Poulovassilis A, Wood PT (2008) Query relaxation in RDF. J Data Semantics X:31–61
Hurtado CA, Poulovassilis A, Wood PT (2009a) Finding top-k approximate answers to path queries. In: Proceedings of FQAS 2009, pp 465–476
Hurtado CA, Poulovassilis A, Wood PT (2009b) Ranking approximate answers to semantic web queries. In: Proceedings of ESWC 2009, pp 263–277
Ilyas I, Aref W, Elmagarmid A (2004) Supporting top-k join queries in relational databases. VLDB J 13:207–221
Ioannidis Y, Poosala V (1999) Histogram-based approximation of set-valued query-answers. In: Proceedings of VLDB 1999, pp 174–185
Kanza Y, Sagiv Y (2001) Flexible queries over semistructured data. In: Proceedings of ACM PODS 2001, pp 40–51
Kasneci G, Ramanath M, Suchanek F, Weikum G (2009) The YAGO-NAGA approach to knowledge discovery. ACM SIGMOD Rec 37(4):41–47
Kiefer C, Bernstein A, Stocker M (2007) The fundamentals of iSPARQL: a virtualtriple approach for similarity-based semantic web tasks. In: Proceedings of ISWC 2007
Kochut K, Janik M (2007) Extended SPARQL for semantic association discovery. In: Proceedings of ESWC 2007, pp 145–159
Koschmieder A, Leser U (2012) Regular path queries on large graphs. In: Proceedings of SSDBM 2012, pp 177–194
Lacroix Z, Murthy H, Naumann F, Raschid L (2004) Links and paths through life sciences data sources. In: Proceedings of DILS 2004, pp 203–211
Larriba-Pey J, Martinez-Bazan N, Dominguez-Sal D (2014) Introduction to graph databases. In: Proceedings of reasoning web 2014, pp 171–194
Leser U, Trissl S (2009) Graph management in the life sciences. In: Encyclopedia of database systems, pp 1266–1271
Libkin L, Vrgoc D (2012) Regular path queries on graphs with data. In: Proceedings of ICDT 2012, pp 74–85
Liu C, Li J, Yu J, Zhou R (2010) Adaptive relaxation forquerying heterogeneous XML data sources. Inf Syst 35(6):688–707

Ma S, Cao Y, Fan W, Huai J, Wo T (2014) Strong simulation: capturing topology in graph pattern matching. ACM Trans Database Syst 39(1):1–46

Mandreoli F, Martoglia R, Villani G, Penzo W (2009) Flexible query answering on graph-modeled data. In: Proceedings of EDBT 2009, pp 216–227

Martin MS, Gutierrez C, Wood PT (2011) A social networks query and transformation language. In: Proceedings of AMW 2011, pp 631–646

Mendelzon AO, Wood PT (1989) Finding regular simple paths in graph databases. In: Proceedings of VLDB 1989, pp 185–193

Mendelzon AO, Wood PT (1995) Finding regular simple paths in graph databases. SIAM J Comput 24(6):1235–1258

Meng X, Ma ZM, Yan L (2008) Providing flexible queries over web databases. In: Knowledge-based intelligent information and engineering systems, pp 601–606

Mishra C, Koudas N (2009) Interactive query refinement. In: Proceedings of EDBT 2009, pp 862–873

Munoz S, Pérez J, Gutierrez C (2007) Minimal deductive systems for RDF. In: Proceedings of ESWC 2007, pp 53–67

Na S, Park S (2005) A process of fuzzy query on new fuzzy object oriented data model. In: Proceedings of DEXA 2005, pp 500–509

Pérez J, Arenas M, Gutierrez C (2006) Semantics and complexity of SPARQL. In: Proceedings of ISWC 2006, pp 30–43

Pérez J, Arenas M, Gutierrez C (2008) nSPARQL: a navigational language for RDF. In: Proceedings of ISWC 2008, pp 66–81

Poulovassilis A, Wood PT (2010) Combining approximation and relaxation in semantic web path queries. In: Proceedings of ISWC 2010, pp 631–646

Poulovassilis A, Selmer P, Wood PT (2012) Flexible querying of lifelong learner metadata. IEEE Trans Learn Technol 5(2):117–129

Poulovassilis A, Gutierrez-Santos S, Mavrikis M (2015) Graph-based modelling of students' interaction data from exploratory learning environments. In: Proceedings of GEDM 2015 (at Educational Data Mining 2015), pp 46–51

Poulovassilis A, Selmer P, Wood PT (2016) Approximation and relaxation of semantic web path queries. J Web Semant 40:1–21

Reddy BRK, Kumar PS (2010) Efficient approximate SPARQL querying of web of linked data. In: Proceedings of URSW 2010, pp 37–48

Sakr S, Elnikety S, He Y (2012) G-SPARQL: a hybrid engine for querying large attributed graphs. In: Proceedings of CIKM 2012, pp 335–344

Sarma D et al (2008) Bootstrapping pay-as-you-go data integration systems. In: Proceedings of SIGMOD 2008, pp 861–874

Sassi M, Tlili O, Ounelli H (2012) Approximate query processing for database flexible querying with aggregates. Trans Large-Scale Data- Knowl Centered Syst V:1–27

Selmer P (2016) Flexible querying of graph-structured data. PhD thesis, Birkbeck, University of London

Selmer P, Poulovassilis A, Wood PT (2015) Implementing flexible operators for regular path queries. In: Proceedings of GraphQ 2015 (EDBT/ICDT Workshops), pp 149–156

Siepen J et al (2008) ISPIDER Central: an integrated database web-server for proteomics. Nucleic Acids Res (Web-Server-Issue) 36:485–490

Suthers D (2015) From contingencies to network-level phenomena: multilevel analysis of activity and actors in heterogeneous networked learning environments. In: Proceedings of LAK 2015, pp 368–377

Theobald M, Schenkel R, Weikum G (2005) An efficient and versatile query engine for TopX search. In: Proceedings of VLDB 2005, pp 625–636

Vanhatalo J, Völzer H, Leymann F, Moser S (2008) Automatic workflow graph refactoring and completion. In: Proceedings of ICSOC 2008. Springer, Berlin, pp 100–115

Wood PT (2012) Query languages for graph databases. ACM SIGMOD Rec 41(1):50–60

Wu B, Ye Q, Yang S, Wang B (2009) Group CRM: a new telecom CRM framework from social network perspective. In: Proceedings of 1st ACM international workshop on complex networks meet information and knowledge management (CNIKM'09), pp 3–10

Wu Y, Yan X, Yang S (2013) Ontology-based subgraph querying. In: Proceedings of ICDE 2013, pp 697–708

Yakovets N, Godfrey P, Gryz J (2015) Towards query optimization for SPARQL property paths. arXiv preprint arXiv:150408262

Yang S, Wu Y, Sun H, Yan X (2014) Schemaless and structureless graph querying. Proc VLDB Endowment 7(7):565–576

Zhang S, Yang J, Jin W (2010) SAPPER: subgraph indexing and approximate matching in large graphs. PVLDB 3(1):1185–1194

Zhou X, Gaugaz J, Balke WT, Nejdl W (2007) Query relaxation using malleable schemas. In: Proceedings of ACM SIGMOD 2007, pp 545–556

Zhu L, Ng WK, Cheng J (2011) Structure and attribute index for approximate graph matching in large graphs. Inf Syst 36(6):958–972

Zou L, Mo J, Chen L, Ozsu MT, Zhao D (2011) gStore: answering SPARQL queries via subgraph matching. PVLDB 4(8):482–493