Flexible Querying for SPARQL

Andrea Calì[1,2], Riccardo Frosini[1], Alexandra Poulovassilis[1], and Peter T. Wood[1]

[1] Dept. of Computer Science and Inf. Systems, Birkbeck University of London, UK
[2] Oxford-Man Institute of Quantitative Finance, University of Oxford, UK
{andrea,riccardo,ap,ptw}@dcs.bbk.ac.uk

Abstract. Flexible querying techniques can be used to enhance users' access to heterogeneous data sets, such as Linked Open Data. This paper extends SPARQL 1.1 with approximation and relaxation operators that can be applied to regular expressions for querying property paths in order to find more answers than would be returned by the exact form of a user query. We specify the semantics of the extended language and we consider the complexity of query answering with the new operators, showing that both data and query complexity are not impacted by our extensions. We present a query evaluation algorithm that returns results incrementally according to their "distance" from the original query. We have implemented this algorithm and have conducted preliminary trials over the YAGO SPARQL endpoint and the Lehigh University Benchmark, showing promising performance for the language extensions.

1 Introduction

Flexible querying techniques are used to enhance access to information stored within information systems, including in terms of user interaction. In particular, users querying an RDF dataset are not always aware of how a query should be formulated in order to correctly retrieve the desired answers. This problem can be caused by a lack of knowledge about the schema of the dataset or about the URIs used in the dataset; moreover, both schema and URIs can change over time. For example, suppose a user wishes to find events which took place in London on 12th December 2012 and poses the query $(x, on, \text{“12/12/12”})$ AND $(x, in, \text{“}London\text{”})$. This returns no results from the YAGO knowledge base because there are no property edges named "on" or "in". Approximating "on" by "happenedOnDate" (which does appear in YAGO) and "in" by "happenedIn" still returns no answers, since "happenedIn" does not connect event instances directly to literals such as "London". However, relaxing $(x, happenedIn, \text{“}London\text{”})$ to $(x, type, Event)$ (using knowledge encoded in YAGO that the domain of "happenedIn" is $Event$) will return all events that occurred on 12th December 2012, including those occurring in London. Alternatively, instead of relaxing the second triple, another approximation step can be applied to $(x, happenedIn, \text{“}London\text{”})$, inserting the property edge $label$ that connects URIs to their labels and yielding the query

$$(x, happenedOnDate, \text{“12/12/12”}) \text{ AND } (x, happenedIn/label, \text{“}London\text{”})$$

R. Meersman et al. (Eds.): OTM 2014, LNCS 8841, pp. 473–490, 2014.

This query now returns every event that occurred on 12th December 2012 in London.

SPARQL is the most prominent RDF query language and, since the latest extension of SPARQL 1.1, it supports property path queries[1] (i.e. *regular path* queries). In this paper we investigate how to extend SPARQL 1.1 with query approximation and query relaxation operations such as those illustrated in the above examples, calling the extended language SPARQL^{AR}. We study the computational complexity of the query answering problem; in particular, we show that the introduction of the new operators does not increase the computational complexity of the original language. We provide tight complexity bounds for several SPARQL fragments; we study *data complexity* (with only the instance graph as input), *query complexity* (with only the query as input) and *combined complexity* (with both query and instance as input). Our complexity results are summarised in Figure 3 on page 485. We then provide a query answering algorithm based on query rewriting, and present and discuss some preliminary experimental results.

Example. Suppose the user wishes to find the geographic coordinates of the "Battle of Waterloo" event by posing the query $(\langle Battle\_of\_Waterloo\rangle, happenedIn/(hasLongitude|hasLatitude), x)$. We see that this query uses the property paths extension of SPARQL, specifically the concatenation (/) and disjunction (|) operators. In the query, the property edge "happenedIn" is concatenated with either "hasLongitude" or "hasLatitude", thereby finding a connection in the dataset between the event and its location (in our case Waterloo) and from the location to both its coordinates. This query does not return any answers from YAGO since YAGO does not store the geographic coordinates of Waterloo. However, by applying an approximation step, we can insert "isLocatedIn" after "happenedIn" which connects the URI representing Waterloo with the URI representing Belgium. The resulting query is

$$Battle\_of\_Waterloo, happenedIn/isLocatedIn/(hasLongitude|hasLatitude), x.$$

Since YAGO does have the geographic coordinates of Belgium, this query will return some answers that may be relevant for the user. Moreover, YAGO does store the coordinates of the "Battle of Waterloo" event, so if the query processor applies an approximation step that deletes the property edge "happenedIn", instead of adding "isLocatedIn", the resulting query $(\langle Battle\_of\_Waterloo\rangle, (hasLongitude|hasLatitude), x)$ returns the desired answers.

Related work. There have been different approaches to applying flexible querying to the Semantic Web. Most of these use similarity measures to retrieve additional relevant answers. An example of flexible querying with similarity measures can be found in [5], where the authors use matching functions for constants such as *strings* or *numeric values*. Similarly in [8] the authors have developed an extension of SPARQL called iSPARQL (imprecise SPARQL) which computes

[1] `http://www.w3.org/TR/sparql11-property-paths/`

string similarity matching using three different functions. A similarity measure technique which exploits the structure of the RDF dataset can be found in [4], where the authors navigate the RDF dataset as a graph in which every path is matched with respect to the query. Other techniques such as ontology driven similarity measures have been developed in [7,6,12]. These techniques use the RDFS ontology to retrieve extra answers and assign a score value to such answers. Finally, [11] shows how a conjunctive regular path query language can be effectively extended with approximation and relaxation techniques, using similar notions of approximation and relaxation as we use here.

In contrast to the above work, we focus here on the SPARQL 1.1 language. We extend, for the first time, this language with query approximation and relaxation operators, terming the extended language SPARQL^{AR}. We specify the semantics of SPARQL^{AR}, study the complexity of query answering, present a query evaluation algorithm returning answers ranked according to their "distance" from the original query, and present the results of a preliminary query performance study. Compared to [11], we focus here on SPARQL 1.1, derive new complexity results, provide a query rewriting algorithm, and present query performance results.

2 Preliminaries

In this section we give preliminary definitions needed to describe the syntax and semantics of SPARQL queries extended with regular expression patterns (known as 'property paths' in the SPARQL documentation) and flexible constructs, namely, query approximation and relaxation. For this kind of querying we will avoid blank nodes, since their use is discouraged for Linked Data because they represent a resource without specifying its name and are identified by an ID which may not be unique in the dataset [2]. Therefore we modify the definition of triples from [9].

Definition 1 (Sets, triples and variables). *Assume there are pairwise disjoint infinite sets U and L (URIs and literals). A tuple $\langle s, p, o\rangle \in U \times U \times (U \cup L)$ is called an RDF triple. In this tuple, s is the subject, p the predicate and o the object. We also assume an infinite set V of variables disjoint from the above sets. We abbreviate any union of these sets as, for instance, $UL = U \cup L$*

To accomodate our formalisation of flexible querying, we add weights to the edges of an RDF-Graph (changing again the definition from [9]). Initially these weights are all 0.

Definition 2 (RDF-Graph). *An RDF-Graph $G = (N, D, E)$ is defined as a finite set of nodes N such that $N \subset UL$, a finite set of predicates D used in the graph, where $D \subset U$, and a finite set of labelled weighted edges E, where each edge is of the form $\langle\langle s, p, o\rangle, c\rangle$ with subject $s \in N \cap U$, object $o \in N$, predicate $p \in D$ and c being the weight, or cost, of the edge.*

We discuss in the next section our query relaxation operator, which is based on the RDF-Schema (RDFS) data modelling vocabulary representing the ontology of an RDF dataset.

Definition 3 (RDF-Schema). *An ontology $K = (N_K, E_K)$ is a directed graph where each node in N_K represents either a class or a property, and each edge E_K is labelled with a symbol from the set $\{sc, sp, dom, range\}$. These edge labels in E_K encompass a fragment of the RDFS vocabulary, namely* rdfs:subClassOf*,* rdfs:subPropertyOf*,* rdfs:domain*,* rdfs:range*.*

In an RDF-graph $G = (N, D, E)$, each node in N represents an instance or a class and each edge in E a property. In an ontology $K = (N_K, E_K)$, each node in N_K represents a class (a "class node") or a property (a "property node"). The intersection of N and N_K is contained in the set of class nodes of K. D is contained in the set of property nodes of K. The predicate *type*, representing the RDF vocabulary rdf:type can be used in G to connect an instance of a class to a node representing that class.

Finally we define the notion of triple patterns, needed to construct queries, and mappings. Again we modify the definitions from [9] to exclude blank nodes.

Definition 4 (Triple patterns). *A triple pattern is a tuple $\langle x, z, y\rangle \in UV \times UV \times UVL$. Given a triple pattern $\langle x, z, y\rangle$, $var(\langle x, z, y\rangle)$ is the set of variables occurring in it.*

Definition 5 (Mapping). *A mapping μ from ULV to UL is a partial function $\mu : ULV \to UL$. We assume 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.*

2.1 Query Syntax

For regular expression patterns in SPARQL^{AR} we will use the definitions given in [3], conforming to the W3C syntax[2].

Definition 6 (Regular expression pattern). *A regular expression pattern $P \in RegEx(U)$ is defined as follows:*

$$P := \epsilon \mid \_ \mid p \mid (P_1|P_2) \mid (P_1/P_2) \mid P^*$$

where ϵ represents the empty pattern, $p \in U$ and $\_$ is a symbol that denotes the disjunction of all URIs in U.

The syntax of query patterns is based on that of [3] but also includes, in our case, the query approximation and relaxation operators APPROX and RELAX:

[2] `http://www.w3.org/TR/sparql11-property-paths/`

Definition 7 (Query Pattern). *A SPARQL^{AR} query pattern Q is defined as follows:*

$$Q := UV \times V \times UVL \mid UV \times RegEx(U) \times UVL \mid Q_1 \ AND \ Q_2 \mid Q \ FILTER \ R \mid RELAX(UV \times RegEx(U) \times UVL) \mid APPROX(UV \times RegEx(U) \times UVL)$$

where R is a SPARQL built-in condition and Q, Q_1, Q_2 are query patterns. We denote by $var(Q)$ the set of all variables occurring in Q.

In the W3C SPARQL syntax, a dot (.) is used as the AND operator, but we avoid it for clarity and use AND instead. Note also that ϵ and _ cannot be specified in property paths in SPARQL 1.1.

A SPARQL query has the form SELECT$_{\overrightarrow{w}}$ WHERE Q, with $\overrightarrow{w} \subseteq var(Q)$ (we may omit here the keyword WHERE for simplicity). Given $Q' = \text{SELECT}_{\overrightarrow{w}} Q$, the *head* of the query, $head(Q')$, is $\overrightarrow{w}$ if $\overrightarrow{w} \neq \emptyset$ and $var(Q)$ otherwise.

3 Semantics of SPARQL^{AR}

The semantics of SPARQL including regular expression query patterns is defined in [3]. For SPARQL^{AR} to handle the weight/cost of edges in an RDF-Graph and subsequently the cost of the approximation and relaxation operators (which we will describe in the following sections), we need to extend the notion of SPARQL query evaluation from returning a set of mappings to returning a set of pairs of the form $\langle \mu, cost \rangle$ where μ is a mapping and *cost* is its cost.

Two mappings μ_1 and μ_2 are said to be *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 where $var(\mu) = var(\mu_1) \cup var(\mu_2)$ and: for each x in $var(\mu_1) \cap var(\mu_2)$, we have $\mu(x) = \mu_1(x) = \mu_2(x)$; for each x in $var(\mu_1)$ but not in $var(\mu_2)$, we have $\mu(x) = \mu_1(x)$; and for each x in $var(\mu_2)$ but not in $var(\mu_1)$, we have $\mu(x) = \mu_2(x)$.

We next define the *union* and *join* of two sets of query evaluation results, M_1 and M_2:

$M_1 \cup M_2 = \{\langle \mu, cost \rangle \mid \langle \mu, cost_1 \rangle \in M_1$ or $\langle \mu, cost_2 \rangle \in M_2$ with $cost = cost_1$ if $\nexists cost_2 . \langle \mu, cost_2 \rangle \in M_2$, $cost = cost_2$ if $\nexists cost_1 . \langle \mu, cost_1 \rangle \in M_1$, and $cost = min(cost_1, cost_2)$ otherwise$\}$.
$M_1 \bowtie M_2 = \{\langle \mu_1 \cup \mu_2, cost_1 + cost_2 \rangle \mid \langle \mu_1, cost_1 \rangle \in M_1$ and $\langle \mu_2, cost_2 \rangle \in M_2$ with μ_1 and μ_2 compatible mappings$\}$.

3.1 Exact Semantics

The semantics of a triple pattern t that may include regular expression patterns as its second component, with respect to a graph G, denoted $[[t]]_G$, is defined recursively as follows:

$$[[\langle x, \epsilon, y\rangle]]_G = \{\langle \mu, 0\rangle \mid var(\mu) = var(\langle x, \epsilon, y\rangle) \wedge \exists c \in N \,.\, \mu(x) = \mu(y) = c\}$$
$$[[\langle x, z, y\rangle]]_G = \{\langle \mu, cost\rangle \mid var(\mu) = var(\langle x, z, y\rangle) \wedge \langle \mu(\langle x, z, y\rangle), cost\rangle \in E\}$$
$$[[\langle x, P_1|P_2, y\rangle]]_G = [[\langle x, P_1, y\rangle]]_G \cup [[\langle x, P_2, y\rangle]]_G$$
$$[[\langle x, P_1/P_2, y\rangle]]_G = [[\langle x, P_1, z\rangle]]_G \bowtie [[\langle z, P_2, y\rangle]]_G$$
$$[[\langle x, P^*, y\rangle]]_G = [[\langle x, \epsilon, y\rangle]]_G \cup [[\langle x, P, y\rangle]]_G \cup \bigcup_{n\geq 1} \{\langle \mu, cost\rangle \mid \langle \mu, cost\rangle \in [[\langle x, P, z_1\rangle]]_G \bowtie [[\langle z_1, P, z_2\rangle]]_G \bowtie \cdots \bowtie [[\langle z_n, P, y\rangle]]_G\}$$

where P, P_1, P_2 are regular expression patterns, x, y, z are in ULV, and $z, z_1, \ldots, z_n$ are fresh variables.

A mapping *satisfies a condition* R, denoted $\mu \models R$, as follows:

R is $x = c$: $\mu \models R$ if $x \in var(\mu)$, $c \in L$ and $\mu(x) = c$
R is $x = y$: $\mu \models R$ if $x, y \in var(\mu)$ and $\mu(x) = \mu(y)$
R is $isURI(x)$: $\mu \models R$ if $x \in var(\mu)$ and $\mu(x) \in U$
R is $isLiteral(x)$: $\mu \models R$ if $x \in var(\mu)$ and $\mu(x) \in L$
R is $R_1 \wedge R_2$: $\mu \models R$ if $\mu \models R_1$ and $\mu \models R_2$
R is $R_1 \vee R_2$: $\mu \models R$ if $\mu \models R_1$ or $\mu \models R_2$
R is $\neg R_1$: $\mu \models R$ if it is not the case that $\mu \models R_1$

Given the above definitions, the overall semantics of queries (excluding APPROX and RELAX) is as follows, where Q, Q_1, Q_2 are query patterns and the projection operator $\pi_{\overrightarrow{w}}$ selects only the subsets of the mappings relating to the variables in $\overrightarrow{w}$:

$$[[Q_1 \text{ AND } Q_2]]_G = [[Q_1]]_G \bowtie [[Q_2]]_G$$
$$[[Q \text{ FILTER } R]]_G = \{\langle \mu, cost\rangle \in [[Q]]_G \mid \mu \models R\}$$
$$[[\text{SELECT}_{\overrightarrow{w}} Q]]_G = \pi_{\overrightarrow{w}}([[Q]]_G)$$

We will omit the SELECT keyword from a query Q if $\overrightarrow{w} = vars(Q)$.

3.2 Query Relaxation

Our relaxation operator is based on that in [11] and relies on RDFS *entailment*. We give a summary here and refer the reader to that paper for full details.

An RDFS graph K_1 entails an RDFS graph K_2, denoted $K_1 \models_{RDFS} K_2$, if we can derive K_2 by applying the rules in Figure 1 iteratively to K_1. For the fragment of RDFS that we consider, $K_1 \models_{RDFS} K_2$ if and only if $K_2 \subseteq cl(K_1)$, with $cl(K_1)$ being the closure of the RDFS Graph K_1 under these rules.

In order to apply relaxation to queries, we need to define the *extended reduction* of an ontology K. Given an ontology K, its extended reduction $extRed(K)$ is computed as follows: (i) compute $cl(K)$; (ii) apply rules of Figure 2 in reverse until no longer applicable (applying a rule in reverse means deleting a triple deducible by the rule); (iii) apply rules 1 and 3 of Figure 1 in reverse until no longer

applicable. Henceforth, we assume that $K = extRed(K)$, which allows us to perform *direct* relaxations on queries (see below) that correspond to the 'smallest' relaxation steps. This is necessary if we are to return query answers to users incrementally in order of increasing cost. We also need K to be acyclic in order for direct relaxation to be well-defined. We say that a triple pattern $\langle x, p, y\rangle$

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

Fig. 1. RDFS entailment rules

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

Fig. 2. Additional rules for extended reduction of an RDFS ontology

directly relaxes to a triple pattern $\langle x', p', y'\rangle$, denoted $\langle x, p, y\rangle \prec_i \langle x', p', y'\rangle$, if $vars(\langle x, p, y\rangle) = vars(\langle x', p', y'\rangle)$ and applying rule i from Figure 1 we can derive $\langle x', p', y'\rangle$ from $\langle x, p, y\rangle$. There is a cost c_i associated with the application of a rule i. We note that since rule 6 changes the position of y, which we want to avoid when it comes to relaxing regular expression patterns (see below), we actually use $(d, type^-, y)$ as the consequent of rule 6; and we also allow a modified form of rule 4 where the triples involving *type* appear with their arguments in reverse order and *type* is replaced by $type^-$.

We say that a triple pattern $\langle x, p, y\rangle$ *relaxes to* a triple pattern $\langle x', p', y'\rangle$, denoted $\langle x, p, y\rangle \leq_K \langle x', p', y'\rangle$, if starting from $\langle x, p, y\rangle$ there is a sequence of direct relaxations that derives $\langle x', p', y'\rangle$. The relaxation cost of deriving $\langle x, p, y\rangle$ from $\langle x', p', y'\rangle$, denoted $rcost(\langle x, p, y\rangle, \langle x', p', y'\rangle)$, is the minimum cost of applying such a sequence of direct relaxations.

We now define the semantics of the RELAX operator in SPARQL^{AR} as follows:

$$\begin{aligned}
[[\text{RELAX}(x, p, y)]]_G &= [[\langle x, p, y\rangle]]_G \cup \{\langle \mu, cost + rcost(\langle x, p, y\rangle, \langle x', p', y'\rangle)\rangle \mid \\
&\qquad \langle x, p, y\rangle \leq_K \langle x', p', y'\rangle \wedge \langle \mu, cost\rangle \in [[\langle x', p', y'\rangle]]_G\} \\
[[\text{RELAX}(x, P_1|P_2, y)]]_G &= [[\text{RELAX}(x, P_1, y)]]_G \cup [[\text{RELAX}(x, P_2, y)]]_G \\
[[\text{RELAX}(x, P_1/P_2, y)]]_G &= [[\text{RELAX}(x, P_1, z)]]_G \bowtie [[\text{RELAX}(z, P_2, y)]]_G \\
[[\text{RELAX}(x, P^*, y)]]_G &= [[\langle x, \epsilon, y\rangle]]_G \cup [[\text{RELAX}(x, P, y)]]_G \cup \\
&\qquad \bigcup_{n \geq 1} \{\langle \mu, cost\rangle \mid \langle \mu, cost\rangle \in [[\text{RELAX}(x, P, z_1)]]_G \bowtie \\
&\qquad \bowtie [[\text{RELAX}(z_1, P, z_2)]]_G \bowtie \cdots \bowtie [[\text{RELAX}(z_n, P, y)]]_G\}
\end{aligned}$$

where P, P_1, P_2 are regular expression patterns, x, x', y, y' are in ULV, p, p' are in U, and z, z_1, ..., z_n are fresh variables.

3.3 Query Approximation

Regarding query approximation, we consider a set of edit operations which transform a regular expression pattern P into a new expression pattern P'. We focus here on the edit operations *deletion*, *insertion* and *substitution* (leaving other possible edit operations such as *transposition* and *inversion* for future work) which are defined as follows:

$$
\begin{array}{ll}
A/p/B \rightsquigarrow (A/\epsilon/B) & \text{deletion} \\
A/p/B \rightsquigarrow (A/\_/B) & \text{substitution} \\
A/p/B \rightsquigarrow (A/\_/p/B) & \text{left insertion} \\
A/p/B \rightsquigarrow (A/p/\_/B) & \text{right insertion}
\end{array}
$$

Here, A and B denote any regular expression and the symbol _ represents every URI from U — so the edit operations allow us to insert any URI and substitute a URI by any other in U. The application of an edit operation *op* has a cost c_{op} associated with it.

We can apply the above set of rules to a URI p in order to approximate it to a regular expression P. We write $p \rightsquigarrow^* P$ if we can apply a sequence of edit operations to p to derive P. The edit cost of deriving P from p, denoted $ecost(p, P)$, is the minimum cost of applying such a sequence of edit operations.

We now define the semantics of the APPROX operator in SPARQL^{AR} as follows:

$$
\begin{aligned}
[[\text{APPROX}(x,p,y)]]_G &= [[\langle x,p,y\rangle]]_G \cup \bigcup \{\langle \mu, cost + ecost(p,P)\rangle \mid \\
&\qquad p \rightsquigarrow^* P \wedge \langle \mu, cost\rangle \in [[\langle x,P,y\rangle]]_G\} \\
[[\text{APPROX}(x,P_1|P_2,y)]]_G &= [[\text{APPROX}(x,P_1,y)]]_G \cup [[\text{APPROX}(x,P_2,y)]]_G \\
[[\text{APPROX}(x,P_1/P_2,y)]]_G &= [[\text{APPROX}(x,P_1,z)]]_G \bowtie [[\text{APPROX}(z,P_2,y)]]_G \\
[[\text{APPROX}(x,P^*,y)]]_G &= [[\langle x,\epsilon,y\rangle]]_G \cup [[\text{APPROX}(x,P,y)]]_G \cup \\
&\qquad \bigcup_{n\geq 1} \{\langle \mu, cost\rangle \mid \langle \mu, cost\rangle \in [[\text{APPROX}(x,P,z_1)]]_G \bowtie \\
&\qquad \bowtie [[\text{APPROX}(z_1,P,z_2)]]_G \bowtie \cdots \bowtie [[\text{APPROX}(z_n,P,y)]]_G\}
\end{aligned}
$$

where P, P_1, P_2 are regular expression patterns, x, y are in ULV, p, p' are in U, and z, z_1, ..., z_n are fresh variables.

4 Complexity of Query Answering

In this section we study the combined, data and query complexity of SPARQL including regular expression patterns, the new APPROX and RELAX operators

and weighted edges in the RDF graph. Our work extends the complexity results in [9,10,13] for simple SPARQL queries and in [1] for SPARQL with regular expression patterns to include our new flexible query constructs in SPARQL^{AR}.

The complexity of query evaluation is based on the following decision problem, which we denote EVALUATION: given as input a graph G, a query Q and a pair $\langle \mu, cost \rangle$, is it the case that $\langle \mu, cost \rangle \in [[Q]]_G$?

Queries without regular expression patterns (i.e. where there are only triple patterns of the form $UV \times UV \times UVL$) and without the flexible query constructs, can be evaluated in polynomial time if they include only the operators AND and FILTER. This result can be achieved by adapting the algorithm given in [9,10,13], adding the cost of the mappings in our setting:

Theorem 1. *EVALUATION can be solved in time $O(|E| \cdot |Q|)$ for query pattern expressions constructed using only AND and FILTER operators.*

Proof. We give an algorithm for the EVALUATION problem that runs in polynomial time: first, for each i such that the triple pattern $\langle x, z, y \rangle_i$ is in Q, we verify that $\langle \mu(\langle x, z, y \rangle_i), cost_i \rangle \in E$ for some $cost_i$. If this is not the case, or if $\sum_i cost_i \neq cost$ we return False. Otherwise we check if μ satisfies the FILTER condition and return True or False accordingly. It is evident that the algorithm runs in polynomial time since verifying that $\langle \mu(\langle x, z, y \rangle_i), cost_i \rangle \in E$ can be done in time $|E|$.

When we add regular expression patterns to queries, the complexity of query evaluation increases slightly. To show this, we start by building an NFA $M_P = (S, T)$ that recognises $\mathcal{L}(P)$, the language denoted by the regular expression P, where S is the set of states (including s_0 and s_f representing the initial and final states respectively) and T is the set of transitions, each of cost 0. We then construct the weighted *product automaton*, H, of G and M_P as follows:

- The states of H are the Cartesian product of the set of nodes N of G and the set of states S of M_P.
- For each transition $\langle \langle s, p, s' \rangle, 0 \rangle$ in M_P and each edge $\langle \langle a, p, b \rangle, cost \rangle$ in E, there is a transition $\langle \langle s, a \rangle, \langle s', b \rangle, cost \rangle$ in H.

Then we check if there exists a path from $\langle s_0, \mu(x) \rangle$ to $\langle s_f, \mu(y) \rangle$ in H. In case there is more than one path, we select one with the minimum cost using Dijkstra's algorithm. Knowing that the number of nodes in H is equal to $|N| \cdot |S|$, the number of edges is at most $|E| \cdot |T|$, and that $|T| \leq |S|^2$, the evaluation can be performed in time $O(|E| \cdot |S|^2 + |N| \cdot |S| \cdot \log(|N| \cdot |S|))$. Noting that the query size is proportional to $|S|$, on the assumption that $|E| \geq |N|$ we therefore have:

Theorem 2. *EVALUATION can be solved in time $O(|E| \cdot |Q|^2)$ for query pattern expressions constructed using only AND, FILTER and regular expression patterns.*

Next, we discuss how adding the SELECT operator increases the complexity even if FILTER is excluded (i.e. including costs in the graph G and regular expression patterns, the complexity given by [9,10] does not change):

Theorem 3. *EVALUATION is NP-complete for query pattern expressions constructed using only AND and regular expression patterns, and including the projection operator SELECT.*

Proof. We first show that the evaluation problem is in NP. Given a pair $\langle \mu, cost \rangle$ and a query $\text{SELECT}_{\overrightarrow{w}} Q$ where Q does not include FILTER, we have to check whether $\langle \mu, cost \rangle$ is in $[[\text{SELECT}_{\overrightarrow{w}} Q]]_G$. We can guess a new mapping μ' such that $\pi_{\overrightarrow{w}}(\langle \mu', cost \rangle) = \langle \mu, cost \rangle$ and consequently check that $\langle \mu', cost \rangle \in [[Q]]_G$ (which can be done in polynomial time as we have seen in Theorem 2). The number of guesses is bounded by the number of variables in Q and values from G to which they can be mapped.

For NP-hardness we first define the problem of graph 3-colourability, which is known to be NP-complete: given a graph $\Gamma = (N_\Gamma, E_\Gamma)$ and three colours r, g, b, is it possible to assign a colour to each node in N_Γ such that no pair of nodes connected by an edge in E_Γ are of the same colour?

We next define the following RDF graph $G = (N, D, E)$:

$$\begin{aligned} N =& \{r, g, b, a\} \qquad D = \{a, p\} \\ E =& \{\langle\langle r, p, g\rangle, 0\rangle, \langle\langle r, p, b\rangle, 0\rangle, \langle\langle g, p, b\rangle, 0\rangle, \langle\langle g, p, r\rangle, 0\rangle, \\ & \langle\langle b, p, r\rangle, 0\rangle, \langle\langle b, p, g\rangle, 0\rangle, \langle\langle a, a, a\rangle, 0\rangle\} \end{aligned}$$

Now we construct the following query Q such that there is a variable x_i corresponding to each node n_i of Γ and there is a triple pattern of the form $\langle x_i, p, x_j \rangle$ in Q if and only if there is an edge (n_i, n_j) in Γ:

$$Q = \text{SELECT}_x((x_i, p, x_j) \text{ AND } \ldots \text{ AND } (x'_i, p, x'_j) \text{ AND } (a, a, x))$$

It is easy to verify that the graph Γ is colourable if and only if $\langle \mu, 0 \rangle \in [[Q]]_G$ with $\mu = \{x \to a\}$.

The following lemma will help us to show that adding the APPROX and RELAX operators will not increase the complexity.

Lemma 1. *EVALUATION of* $[[APPROX(x, P, y)]]_G$ *and* $[[RELAX(x, P, y)]]_G$ *can be done in polynomial time.*

Proof (Premise). Given a pair $\langle \mu, cost \rangle$ we have to verify in polynomial time that $\langle \mu, cost \rangle \in [[\text{APPROX}(x, P, y)]]_G$ or $\langle \mu, cost \rangle \in [[\text{RELAX}(x, P, y)]]_G$. We start by building an NFA $M_P = (S, T)$ as described earlier.

Proof (Approximation). An *approximate automaton* $A_P = (S, T')$ is constructed starting from M_P and adding the following additional transitions (similarly to the construction in [11]):

- For each state $s \in S$ there is a transition $\langle\langle s, \_, s\rangle, \alpha\rangle$, where α is the cost of insertion.
- For each transition $\langle\langle s, p, s'\rangle, 0\rangle$ in M_P, where $p \in D$, there is a transition $\langle\langle s, \epsilon, s'\rangle, \beta\rangle$, where β is the cost of deletion.

– For each transition $\langle\langle s, p, s'\rangle, 0\rangle$ in M_P, where $p \in D$, there is a transition $\langle\langle s, \_, s'\rangle, \gamma\rangle$, where γ is the cost of substitution.

We then form the weighted product automaton, H, of G and A_P as follows:

– The states of H will be the Cartesian product of the set of nodes N of G and the set of states S of A_P.
– For each transition $\langle\langle s, p, s'\rangle, cost_1\rangle$ in A_P and each edge $\langle\langle a, p, b\rangle, cost_2\rangle$ in E, there is a transition $\langle\langle s, a\rangle, \langle s', b\rangle, cost_1 + cost_2\rangle$ in H.
– For each transition $\langle\langle s, \epsilon, s'\rangle, cost\rangle$ in A_P and each node $a \in N$, there is a transition $\langle\langle s, a\rangle, \langle s', a\rangle, cost\rangle$ in H.
– For each transition $\langle\langle s, \_, s'\rangle, cost_1\rangle$ in A_P and each edge $\langle\langle a, p, b\rangle, cost_2\rangle$ in E, there is a transition $\langle\langle s, a\rangle, \langle s', b\rangle, cost_1 + cost_2\rangle$ in H.

Finally we check if there exists a path from $\langle s_0, \mu(x)\rangle$ to $\langle s_f, \mu(y)\rangle$ in H. Again, if there exists more than one path we select one with minimum cost using Dijkstra's Algorithm. Knowing that the number of nodes in H is $|N| \cdot |S|$ and that the number of edges in H is at most $(|E| + |N|) \cdot |S|^2$, the evaluation can therefore be computed in $O((|E| + |N|) \cdot |S|^2 + |N| \cdot |S| \cdot \log(|N| \cdot |S|))$.

Proof (Relaxation). Given an ontology $K = extRed(K)$ we build the *relaxed automaton* $R_P = (S', T', S_0, S_f)$ starting from M_P (similarly to the construction in [11]). S_0 and S_f represent the sets of initial and final states, and S' contains every state in S plus the states in S_0 and S_f. Initially S_0 and S_f contain s_0 and s_f respectively. Each initial and final state in S_0 and S_f is labelled with either a constant or the symbol $*$; in particular, s_0 is labelled with x if x is a constant or $*$ if it is a variable and similarly s_f is labelled with y if y is a constant or $*$ if it is a variable. An *incoming* (*outgoing*) *clone* of a state s is a new state s' such that s' is an initial or final state if s is, s' has the same set of incoming (outgoing) transitions as s, and has no outgoing (incoming) transitions. Initially T' contains all the transitions in T. We recursively add states to S_0 and S_f, and transitions to T' as follows until we reach a fixpoint:

– For each transition $\langle\langle s, p, s'\rangle, cost\rangle \in T'$ and $\langle p, sp, p'\rangle \in K$ add the transition $\langle\langle s, p', s'\rangle, cost + \alpha\rangle$ to T', where α is the cost of applying rule 2.
– For each transition $\langle\langle s, type, s'\rangle, cost\rangle \in T'$, $s' \in S_f$ and $\langle c, sc, c'\rangle \in K$ such that s' is annotated with c add an outgoing clone s'' of s' annotated with c' to S_f and add the transition $\langle\langle s, type, s''\rangle, cost + \beta\rangle$ to T', where β is the cost of applying rule 4.
– For each transition $\langle\langle s, type^-, s'\rangle, cost\rangle \in T'$, $s \in S_0$ and $\langle c, sc, c'\rangle \in K$ such that s is annotated with c add an incoming clone s'' of s annotated with c' to S_0 and add the transition $\langle\langle s'', type^-, s'\rangle, cost + \beta\rangle$ to T', where β is the cost of applying rule 4.
– For each $\langle\langle s, p, s'\rangle, cost\rangle \in T'$, $s' \in S_f$ and $\langle p, dom, c\rangle$ such that s' is annotated with a constant, add an outgoing clone s'' of s' annotated with c to S_f, and add the transition $\langle\langle s, type, s''\rangle, cost + \gamma\rangle$ to T', where γ is the cost of applying rule 5.

– For each $\langle\langle s, p, s'\rangle, cost\rangle \in T'$, $s \in S_0$ and $\langle p, range, c\rangle$ such that s is annotated with a constant, add an incoming clone s'' of s annotated with c to S_0, and add the transition $\langle\langle s'', type^-, s'\rangle, cost + \delta\rangle$ to T', where δ is the cost of applying rule 6.

(We note that because queries and graphs do not contain edges labelled *sc* or *sp*, rules 1 and 3 in Figure 1 are inapplicable as far as query relaxation is concerned.)

We then form the weighted product automaton, H, of G and R_P as follows:

– For each node $a \in N$ of G and each state $s \in S'$ of R_P, then $\langle s, a\rangle$ is a state of H if s is labelled with either $*$ or a, or is unlabelled.
– For each transition $\langle\langle s, p, s'\rangle, cost_1\rangle$ in R_P and each edge $\langle\langle a, p, b\rangle, cost_2\rangle$ in E such that $\langle s, a\rangle$ and $\langle s', b\rangle$ are states of H, then there is a transition $\langle\langle s, a\rangle, \langle s', b\rangle, cost_1 + cost_2\rangle$ in H.
– For each transition $\langle\langle s, type^-, s'\rangle, cost_1\rangle$ in R_P and each edge $\langle\langle a, type, b\rangle, cost_2\rangle$ in E such that $\langle s, b\rangle$ and $\langle s', a\rangle$ are states of H, then there is a transition $\langle\langle s, b\rangle, \langle s', a\rangle, cost_1 + cost_2\rangle$ in H.

Finally we check if there exists a path from $\langle s, \mu(x)\rangle$ to $\langle s', \mu(y)\rangle$ in H, where $s \in S_0$ and $s' \in S_f$. Again, if there exists more than one path we select one with minimum cost using Dijkstra's Algorithm. Knowing that the number of nodes in H is at most $|N| \cdot |S'|$ and the number of edges in H is at most $|E| \cdot |S'|^2$, the evaluation can therefore be computed in $O(|E| \cdot |S'|^2 + |N| \cdot |S'| \cdot \log(|N| \cdot |S'|))$.

Proof (Conclusion). We can conclude that both query approximation and query relaxation can be evaluated in polynomial time. In particular, the evaluation can be done in $O(|E|)$ time with respect to the data and in polynomial time with respect to the query.

Through the previous lemma we can conclude our combined complexity study with the following theorem:

Theorem 4. *EVALUATION is NP-complete for query pattern expressions constructed using regular expression patterns and the operators AND, FILTER, RELAX, APPROX and SELECT.*

With the next theorem, we consider the behaviour of our extended SPARQL language in terms of data complexity. In particular we extend what we have already shown in Theorems 3 and 4 by changing the decision problem stated earlier to the following: given as input a graph G and a pair $\langle\mu, cost\rangle$, is it the case that $\langle\mu, cost\rangle \in [[Q]]_G$, with Q a fixed query?

Theorem 5. *EVALUATION is PTIME in data complexity for query pattern expressions constructed using regular expression patterns, and the operators AND, FILTER, RELAX, APPROX and SELECT.*

Proof. In order to prove the theorem, we devise an algorithm that runs in polynomial time with respect to the size of the graph G. We start by building a new mapping μ' such that each variable $x \in var(\mu')$ appears in $var(Q)$ but not in $var(\mu)$, and to each we assign a different constant from ND. We then verify in polynomial time that $\langle \mu \cup \mu', cost \rangle$ is in $[[Q]]_G$. The number of mappings we can generate is $O(|ND|^{|var(Q)|})$. Since the query is fixed we can therefore say that the evaluation with respect to the data is in polynomial time.

Results for the query complexity of our extended SPARQL language follow directly from Lemma 1 and Theorems 1, 2 and 3. In fact, Lemma 1 and Theorems 1 and 2 show an algorithm that evaluates AND queries with APPROX, RELAX and regular expression patterns in polynomial time with respect to the query. The proof of Theorem 3 reduces the problem of EVALUATION from the graph 3-colourability problem, and since the graph constructed for the reduction does not change with respect to the input problem, we can affirm that query complexity remains NP-complete.

We summarise the complexity study of our extended SPARQL language in Figure 3, where we show the combined, data and query complexity for the language fragments identified by the operators included.

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

Fig. 3. Complexity of various SPARQL fragments

5 Query Evaluation

In this section we describe how to compute the relaxed and approximated answer of a SPARQL^{AR} query by making use of a query rewriting algorithm, following a similar approach to [6,7,12]. In particular, given a query Q with the APPROX and/or RELAX operators, our goal is to incrementally build a set of queries $\{Q_0, \ldots, Q_n\}$ that do not contain these operators such that $\bigcup_i [[Q_i]]_G = [[Q]]_G$. Moreover we need to produce answers in order of increasing cost.

The query rewriting algorithm starts by considering the query Q_0 which returns the exact answer of the query Q, i.e., ignoring the APPROX and RELAX operators. To keep track of which triple patterns need to be relaxed or approximated, we label such triple patterns with A for approximation and R for relaxation. For each triple pattern $\langle x_i, P_i, y_i \rangle$ in Q_0 labelled with A (R) and each URI p that appears in P_i, we construct a new query applying one step of approximation (relaxation) to p, and to each query we assign the cost of applying such a step. From each query constructed in this way, we next produce a new set of queries applying a second step of approximation or relaxation. The cost of each query is equal to the cost of the original query plus the cost of the sequence of approximations or relaxations applied to it. For practical reasons we limit the number of queries generated by bounding the cost of the queries up to a maximum value c.

To compute the query answers, we incrementally apply an evaluation function, *eval*, to each query generated by the rewriting algorithm in ranked order of the cost of the queries, and to each mapping returned by *eval* we assign the cost of the query. If we generate a particular mapping more than once we keep the one with the lowest cost. The *eval* function takes as input a query Q and a graph G and returns $[[Q]]_G$. The overall query evaluation algorithm is defined below where *rew* is the query rewriting algorithm and we assume that the set of mappings M is maintained in order of increasing cost (e.g. as a priority queue):

Algorithm 1. Flexible Query Evaluation

input : Query Q; approx/relax max cost c; Graph G; Ontology K.
output: List of pairs mapping/cost M sorted by cost.
$M := \emptyset$;
foreach $\langle Q', cost \rangle \in$ *rew(Q,c,K)* **do**
  foreach $\langle \mu, 0 \rangle \in$ *eval(Q',G)* **do**
    $M := M \cup \{\langle \mu, cost \rangle\}$
return M;

Example 1. Consider the following ontology K (satisfying $K = extRed(K)$), which is a fragment of the YAGO knowledge base[3] derived from multiple sources such as Wikipedia, WordNet and GeoNames:

$$K = (\{happenedIn, placedIn, happenedOnDate, Event\},$$
$$\{\langle happenedIn, sp, placedIn \rangle, \langle happenedIn, dom, Event \rangle\})$$

Suppose a user wishes to find every event which took place in London on 12th December 2012 and poses the following query Q:

APPROX(x, *happenedOnDate*, "12/12/12") AND RELAX(x, *happenedIn*, "*London*").

Without applying APPROX or RELAX, this query does not return any answers when evaluated on the YAGO endpoint (because "happenedIn" connects to URIs representing places and "London" is a literal, not a URI). After the first step of approximation and relaxation, the following queries are generated:

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

$Q_1 = (x, \epsilon, \text{"12/12/12"})_A$ AND $(x, happenedIn, \text{"London"})_R$
$Q_2 = (x, happenedOnDate/\_, \text{"12/12/12"})_A$ AND $(x, happenedIn, \text{"London"})_R$
$Q_3 = (x, \_/happenedOnDate, \text{"12/12/12"})_A$ AND $(x, happenedIn, \text{"London"})_R$
$Q_4 = (x, \_, \text{"12/12/12"})_A$ AND $(x, happenedIn, \text{"London"})_R$
$Q_5 = (x, happenedOnDate, \text{"12/12/12"})_A$ AND $(x, placedIn, \text{"London"})_R$
$Q_6 = (x, happenedOnDate, \text{"12/12/12"})_A$ AND $(x, type, Event)_R$

Each of these also returns empty results, with the exception of query Q_6 which returns every event occurring on 12/12/12 (including amongst them the events occurring in London that are of interest to the user).

We have conducted a theoretical study of the correctness and termination of the Rewriting Algorithm which can be found in the extended version of the paper (`http://www.dcs.bbk.ac.uk/~riccardo/ODBASE2014Extended.pdf`), where the Rewriting Algorithm itself is also specified in detail.

6 Experimental Results

We have implemented our query evaluation algorithm and have conducted preliminary trials over the YAGO SPARQL endpoint and the Lehigh University Benchmark (LUBM)[4]. Using the latter we generated 3 datasets: D_1 with 149,973 triples (13Mb in XML format), D_2 with 421,562 triples (44Mb), and D_3 with 673,416 triples (65Mb). The LUBM ontology[5] consists of 355 statements and describes a university domain. We ran our experiments on a Windows 7 computer with 4Gb of RAM and a quadcore-core i7 CPU at 2.0Ghz. The query evaluation algorithm was implemented in Java and we used Jena for the SPARQL query execution[6].

In Figure 4 we show the number of answers and the number of seconds it takes to answer a set of exact queries, Q_1 - Q_4. In Figure 5 we show both the number of answers and the number of seconds it takes to answer approximated and/or relaxed versions of the same queries, Q'_1 - Q'_4. For these experiments, the cost of all edit and relaxation operations was set to 1 (in practice the user may set different costs depending on the query or application). The maximum cost was also fixed at 1 for each of Q'_1 - Q'_4 (in practice the user would set a small cost – 0 or 1 – to begin with, explore the results returned, and iteratively request more results at greater cost as necessary).

Query Q_1, which returns every university and its head, is as follows:

```
SELECT  ?X ?Z WHERE {?X ub:headOf ?Z}
```

For Q'_1, we RELAX the triple pattern to find other people who work at a university. With the maximum cost set to 1, the rewriting algorithm generated only 1

[4] `http://swat.cse.lehigh.edu/projects/lubm/`
[5] `http://swat.cse.lehigh.edu/onto/univ-bench.owl`
[6] `https://jena.apache.org/`

more query. The new query generated 817 answers at cost 1 for the first dataset, 2263 answers at cost 1 for the second and finally 3606 answers for the third.

Query Q_2 returns people who work for department `Department0.University0` with their email address and phone number:

```
SELECT *
WHERE {?X ub:worksFor <http://www.Department0.University0.edu> .
    ?X ub:name ?Y1 . ?X ub:emailAddress ?Y2 . ?X ub:telephone ?Y3}
```

For Q_2', we APPROX the first triple pattern, allowing details to be returned of people who have other relationships with the department. Even with a maximum cost of 1, Q_2' returns many more answers than Q_2.

Query Q_3 returns every sub-organization of organizations affiliated with `University0`:

```
SELECT ?Y ?Z
WHERE {?Y ub:subOrganizationOf* ?Z .
    ?Z ub:affiliatedOrganizationOf <http://www.University0.edu>}
```

For Q_3' we RELAX the first triple pattern and APPROX the second one, allowing answers to be returned relating to organizations that have other relationships with `University0`.

Query Q_4 returns students along with courses they attend which are taught by their advisor:

```
SELECT ?X ?Z
WHERE {?X ub:advisor/ub:teacherOf ?Z . ?X ub:takesCourse ?Z}
```

For Q_4' we RELAX the first triple pattern and APPROX the second one.

Dataset	Q_1	Q_2	Q_3	Q_4
D_1	23/0.001s	34/0.004s	0/0.197s	331/0.129s
D_2	63/0.006s	34/0.005s	0/0.64s	883/0.296s
D_3	100/0.007s	34/0.006s	0/1.67s	1381/0.517s

Fig. 4. Exact queries (number of answers/time)

Dataset	Q_1'	Q_2'	Q_3'	Q_4'
D_1	840/0.015s	605/0.421s	743/0.924s	348/60.7s
D_2	2326/0.055s	605/1s	756/3.17s	925/451s
D_3	3706/0.105s	605/1.6s	767/4.44s	1461/1492s

Fig. 5. Approx/relax queries (number of answers/time)

We see that response times are good for Q_1' - Q_3', allowing many more answers to be returned than for Q_1 - Q_3 within a reasonable amount of time. However, response times for Q_4' are poor and show a large amount of time required to

return relatively few additional answers compared with Q_4. The rewriting algorithm generates three new queries at distance 1 starting from Q_4. One of these queries takes almost all the time shown in the figure (57 seconds on D_1, 435 seconds on D_2 and 1470 seconds on D_3). This query returns students and courses that they did not attend which were taught by their advisor:

```
SELECT ?X ?Z
WHERE {?X ub:advisor/ub:teacherOf ?Z . ?X !ub:takesCourse ?Z}
```

(Since SPARQL does not accept the symbol "_" in triple patterns, we exploit the symbol "!" instead. The predicate `!ub:takesCourse` is generated as a result of a substitution operation in which `ub:takesCourse` is substituted by "_". It is sufficient to use `!ub:takesCourse` to match every predicate other than `ub:takesCourse` since the latter will already have been matched.) Since the triple pattern `?X !ub:takesCourse ?Z` matches a large number of triples in the dataset and most of these do not match the first triple pattern, the query returns a limited number of answers in a very long time.

An advantage of our query rewriting approach is that existing techniques for SPARQL query optimisation and evaluation can be reused. Even though this initial experimental study is promising, we will investigate optimizing the rewriting algorithm since it can generate a large number of queries. Moreover, we will also investigate optimization techniques to deal with queries such as Q_4 which cannot be evaluated efficiently using a naive approach.

7 Concluding Remarks and Future Work

We have presented in this paper a study of flexible querying for an extended fragment of the SPARQL 1.1 language. We have shown that adding approximation and relaxation operators to this fragment does not increase the complexity of query answering, and that such operators can be useful in finding more answers than would be returned by the exact form of a user's query.

We have specified the semantics of our extended language, and have described a query evaluation algorithm based on query rewriting that returns results incrementally according to their "distance" from the original query. Our preliminary experimental studies show promising query performance. Regarding the relative ranking of relaxation and approximation, this depends on the user's query, the data and the user's knowledge of the data structuring: if the user uses appropriate predicates but they are too specialised, then RELAX may be more useful; if the user omits part of the necessary structure from their query or uses incorrect predicates, then APPROX may be more useful.

Our ongoing work involves a study of query containment in our extended SPARQL language, and how query costs influence this. Through an investigation of query containment we plan to devise optimizations for our rewriting algorithm. For example, it is possible to decrease the number of queries generated by the rewriting algorithm if $Q = Q_1$ AND Q_2 and $[[Q_1]]_G \subseteq [[Q_2]]_G$, then $[[Q]]_G = [[Q_1]]_G$. Our ongoing work also comprises a deeper empirical investigation of the performance of our query evaluation algorithm, and of possible

optimizations. Another direction of research is application of our approximation and relaxation operators, query evaluation and query optimization techniques to federated query processing for SPARQL 1.1.

References

1. Alkhateeb, F., Baget, J.-F., Euzenat, J.: Extending SPARQL with regular expression patterns (for querying RDF). Web Semant. 7(2), 57–73 (2009)
2. Bizer, C., Cyganiak, R., Heath, T.: How to publish linked data on the web. Web page (2007) (Revised 2008) (accessed February 22, 2010)
3. Chekol, M.W., Euzenat, J., Genevès, P., Layaïda, N.: PSPARQL Query Containment. Research report, EXMO - INRIA Grenoble Rhône-Alpes / LIG Laboratoire d'Informatique de Grenoble, WAM - INRIA Grenoble Rhône-Alpes / LIG Laboratoire d'Informatique de Grenoble (June 2011)
4. De Virgilio, R., Maccioni, A., Torlone, R.: A similarity measure for approximate querying over RDF data. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops, EDBT 2013, pp. 205–213. ACM, New York (2013)
5. Hogan, A., Mellotte, M., Powell, G., Stampouli, D.: Towards fuzzy query-relaxation for RDF. In: Simperl, E., Cimiano, P., Polleres, A., Corcho, O., Presutti, V. (eds.) ESWC 2012. LNCS, vol. 7295, pp. 687–702. Springer, Heidelberg (2012)
6. Huang, H., Liu, C.: Query relaxation for star queries on RDF. In: Chen, L., Triantafillou, P., Suel, T. (eds.) WISE 2010. LNCS, vol. 6488, pp. 376–389. Springer, Heidelberg (2010)
7. Huang, H., Liu, C., Zhou, X.: Computing relaxed answers on RDF databases. In: Bailey, J., Maier, D., Schewe, K.-D., Thalheim, B., Wang, X.S. (eds.) WISE 2008. LNCS, vol. 5175, pp. 163–175. Springer, Heidelberg (2008)
8. Kiefer, C., Bernstein, A., Stocker, M.: The fundamentals of iSPARQL: A virtual triple approach for similarity-based semantic web tasks. In: Aberer, K., et al. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 295–309. Springer, Heidelberg (2007)
9. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
10. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. 34(3), 16:1–16:45 (2009)
11. Poulovassilis, A., Wood, P.T.: Combining approximation and relaxation in semantic web path queries. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 631–646. Springer, Heidelberg (2010)
12. Reddy, B.R.K., Kumar, P.S.: Efficient approximate SPARQL querying of web of linked data. In: Bobillo, F., Carvalho, R.N., da Costa, P.C.G., d'Amato, C., Fanizzi, N., Laskey, K.B., Laskey, K.J., Lukasiewicz, T., Martin, T., Nickles, M., Pool, M. (eds.) URSW. CEUR Workshop Proceedings, vol. 654, pp. 37–48. CEUR-WS.org (2010)
13. Schmidt, M.: Foundations of SPARQL Query Optimization. PhD thesis, Albert-Ludwigs-Universitat Freiburg (2009)