Undefined 0 (2014) 1–0
IOS Press

Flexible Query Processing for SPARQL

Andrea Calì [a,b] Riccardo Frosini [a] Alexandra Poulovassilis [a] Peter T. Wood [a]
[a] *London Knowledge Lab, Birkbeck University of London, UK*
Email: {andrea,riccardo,ap,ptw}@dcs.bbk.ac.uk
[b] *Oxford-Man Institute of Quantitative Finance, University of Oxford, UK*

Abstract. Flexible querying techniques can enhance users' access to complex, heterogeneous datasets in settings such as RDF Linked Data where the user may not always know how a query should be formulated in order to retrieve the desired answers. This paper presents query processing algorithms for an extended fragment of SPARQL 1.1 incorporating regular path queries (property path queries), and query approximation and relaxation operators. Our flexible query processing approach is based on query rewriting and returns answers incrementally according to their "distance" from the exact form of the query. We formally show the soundness, completeness and termination properties of our query rewriting algorithm. We also present empirical results that show promising query processing performance for the extended language.

1. Introduction

Flexible querying techniques have the potential to enhance users' access to complex, heterogeneous datasets. In particular, users querying RDF Linked Data may lack full knowledge of the structure of the data, its irregularities, and the URIs used within it. Moreover, the schemas and URIs used can also evolve over time. This makes it difficult for users to formulate queries that precisely express their information retrieval requirements. Hence, providing users with flexible querying capabilities is essential.

SPARQL is the predominant language for querying RDF data and, in the latest extension of SPARQL 1.1, it supports property path queries (i.e. *regular path* queries) over the RDF graph. However, only exact matches to queries can be returned.

Example 1. *Suppose a user wishes to find events that took place in London on 12th December 2012 and poses the following query on the YAGO knowledge base*[1] *derived from multiple sources such as Wikipedia, WordNet and GeoNames:*

$$(x, on, \text{“12/12/12”}) \;\; AND \; (x, in, \text{“}London\text{”})$$

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

Approximating "on" by "happenedOnDate" (which does appear in YAGO) and "in" by "happenedIn" gives the following query:

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

This still returns no answers, since "happenedIn" does not connect event instances directly to literals such as "London". However, relaxing now $(x, happenedIn, \text{“}London\text{”})$ *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 above, another approximation step can be applied to it, inserting the property label that connects URIs to their labels and yielding the following query:

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

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

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

Example 2. *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 "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 to the user.

Moreover, YAGO does in fact store the coordinates of the "Battle of Waterloo" event, so if the query processor applies an approximation step that deletes the property "happenedIn", instead of adding "isLocatedIn", the resulting query

$$(\langle Battle\_of\_Waterloo\rangle, (hasLongitude|hasLatitude), x)$$

returns the desired answers.

In this paper we describe an extension of 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 first presented SPARQL^{AR} in [5], focussing on its syntax, semantics and complexity of query answering. We showed that the introduction of the query approximation and query relaxation operators does not increase the theoretical complexity of the language, and we provided complexity bounds for several language fragments. In this paper, we focus on our query processing algorithms for SPARQL^{AR}, examining their correctness and termination properties and presenting the results of a performance study over the YAGO dataset.

The rest of the paper is structured as follows. Section 2 describes related work on flexible querying for the Semantic Web, and on query approximation and relaxation more generally. Section 3 presents the necessary preliminary definitions, summarising the syntax, semantics and complexity of SPARQL^{AR}. Section 4 presents in detail our query processing approach for SPARQL^{AR}, which is based on query rewriting. We present our query processing algorithms, and formally show the soundness and completeness of our query rewriting algorithm, as well as its termination. Section 5 presents and discusses the results of a perfomance study over the YAGO dataset. Finally, Section 6 gives our concluding remarks and directions of further work.

2. Related work

There have been several previous proposals for applying flexible querying to the Semantic Web, mainly employing similarity measures to retrieve additional answers of possible relevance. For example, in [10] matching functions are used for constants such as strings and numbers, while in [13] an extension of SPARQL is developed called iSPARQL which uses three different matching functions to compute string similarity. In [7], the structure of the RDF data is exploited and a similarity measurement technique is proposed which matches paths in the RDF graph with respect to the query. Ontology driven similarity measures are proposed in [12,11,17] which use the RDFS ontology to retrieve extra answers and assign a score to them.

In [8] methods for relaxing SPARQL-like triple pattern queries automatically are presented. Query relaxations are produced by means of statistical language models for structured RDF data and queries. The query processing algorithms merge the results of different relaxations into a unified results list.

Recently a fuzzy approach has been used to extend the XPath query language [2] with the aim of providing mechanisms to assign priorities to queries and to rank query answers. These

techniques are based on fuzzy extensions of the boolean operators.

Flexible querying approaches for SQL have been discussed in [18] where the authors describe a system that enables a user to issue an SQL aggregation query, see results as they are produced, and adjust the processing as the query runs. This approach allows users to write flexible queries containing linguistic terms, observe the progress of their aggregation queries, and control execution on the fly.

An approximation technique for conjunctive queries on probabilistic databases has been investigated in [9]. The authors use propositional formulas for approximating the queries. Formulas and queries are connected in the following way: given an input database where every tuple is annotated by a distinct variable, each tuple t in the query answer is annotated by a formula over the input tuples that contributed to t.

Another flexible querying technique for relational databases is described in [4]. The authors present an extension to SQL (Soft–SQL) which permits so-called soft conditions. Such conditions tolerate degrees of under-satisfaction of query by exploiting the flexibility offered by fuzzy sets theory.

Finally, [16] 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 all the above work, our focus is on the SPARQL 1.1 language. In [5] we extended, for the first time, this language with query approximation and query relaxation operators, terming the extended language SPARQL^{AR}. Here, we present in detail our query processing algorithms for SPARQL^{AR}. Our query processing approach is based on query rewriting, whereby we incrementally generate a set of SPARQL 1.1 queries from the original SPARQL^{AR} query, evaluate these queries using existing technologies, and return answers ranked according to their "distance" from the original query. We examine the correctness and termination properties of our query rewriting algorithm and present the results of a performance study on the YAGO dataset.

3. Preliminaries

In this section we give definitions of the syntax and semantics of SPARQL extended with regular expression patterns (known as 'property paths' in the SPARQL documentation[2]) and with query approximation and relaxation operators. Our SPARQL^{AR} language and its syntax and semantics were first introduced in [5]. We summarise them again here, as necessary preliminaries to the rest of the paper. We also summarise here the complexity results from in [5], for completeness.

Definition 1 (Sets, triples and variables)**.** *Assume pairwise disjoint infinite sets U and L (of URIs and literals, respectively). 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. We assume also an infinite set V of* variables *that is disjoint from the above sets. We abbreviate any union of the sets U, L and V by concatenating their names; for instance, $UL = U \cup L$.*

Note that, in the above definition, we modify the definition of triples from [14] by omitting 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 [3].

Definition 2 (RDF-Graph)**.** *An* RDF-Graph *G is a directed graph (N, D, E) where: N is a finite set of nodes such that $N \subset UL$; D is a finite set of predicates such that $D \subset U$; E is a finite set of labelled, weighted edges of the form $\langle\langle s, p, o\rangle, c\rangle$ such that the the edge source (subject) $s \in N \cap U$, the edge target (object) $o \in N$, the edge label $p \in D$ and the edge weight $c \geq 0$.*

Note that, in the above definition, we modify the definition of an RDF-Graph [14] to add weights to the edges, for greater ease of formalising our flexible querying semantics. Initially, these weights are all 0.

Definition 3 (Ontology)**.** *An* ontology *K is a directed graph (N_K, E_K) where each node in N_K represents either a class or a property, 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, namely*

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

rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain *and* rdfs:range*, respectively.*

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. 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. 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 N_K. D is contained in the set of property nodes of N_K.

Definition 4 (Triple pattern)**.** *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.*

Note that again we modify the definition from [14] to exclude blank nodes.

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.*

3.1. Syntax of SPARQL^{AR} queries

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.

This definition of regular expression patterns is the same as that in [6]. Our query pattern syntax is also based on that of [6], but includes also our query approximation and relaxation operators APPROX and RELAX:

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 for conjuction but, for greater clarify, we use AND instead. Note also that ϵ and $\_$ cannot be specified in property paths in SPARQL 1.1.)

A SPARQL^{AR} 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 $Q' =$ SELECT$_{\overrightarrow{w}} Q$, the *head* of the query, $head(Q')$, is $\overrightarrow{w}$ if $\overrightarrow{w} \neq \emptyset$ and $var(Q)$ otherwise.

3.2. Semantics of SPARQL^{AR} queries

We extend the semantics of SPARQL with regular expression query patterns given in [6] in order to handle the weight/cost of edges in an RDF-Graph and the cost of the approximation and relaxation operators. In particular, we 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 finally 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$\}$.

3.2.1. Exact Semantics

The semantics of a triple pattern t that may include a regular expression pattern 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 $\mu \models R_1$;

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 $\overrightarrow{w} = vars(Q)$.

3.2.2. Query Relaxation

Our relaxation operator is based on that in [16] and relies on RDFS *entailment*. An RDFS graph K_1 entails an RDFS graph K_2, denoted $K_1 \models_{RDFS} K_2$, if K_2 can be derived 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, the *extended reduction* of an ontology K is required. Given an ontology K, its extended reduction $extRed(K)$ is computed as follows: (i) compute $cl(K)$; (ii) apply the 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 *direct* relaxations to be applied to queries (see below), corresponding to the 'smallest' relaxation steps. This is necessary for returning query answers to users incrementally in order of increasing cost. Also, K needs to be acyclic in order for direct relaxation to be well-defined. A triple pattern $\langle x, p, y\rangle$ *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 $\langle x', p', y'\rangle$ is derived from $\langle x, p, y\rangle$ by applying rule i from Figure 1. 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 $type^-$.

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.

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

Fig. 1. RDFS entailment rules

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

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

The semantics of the RELAX operator in SPARQL^{AR} are as follows:

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

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.

Example 3. *Consider the following portion $K = (N_K, E_K)$ of the YAGO ontology, where N_K is*

$$\{wasCreatedOnDate, startsExistingOnDate, wasBornOnDate, Date, Person, English\_politicians, politician\},$$

and E_K is

$$\{(wasCreatedOnDate, sp, startsExistingOnDate), (wasBornOnDate, sp, startsExistingOnDate), (wasCreatedOnDate, range, Date), (actedIn, domain, Person), (English\_politicians, sc, politician)\}$$

Suppose the user is looking for the birthday of all actors who played in the film Titanic:

```
SELECT * WHERE {
?x actedIn <Titanic> .
?x wasCreatedOnDate ?z }
```

The above query returns no answers, due to the fact that wasCreatedOnDate is not the correct predicate to use. By applying relaxation to the second triple pattern using rule (2) of RDFS entailment, it is possible to replace the predicate wasCreatedOnDate by startsExistingOnDate. Since K states that the predicate wasBornOnDate is a sub-property of startsExistingOnDate, the relaxed query will return the birthday of every actor.

As another example, suppose the user poses the following query:

```
SELECT * WHERE {
?x type <English_politicians> .
?x wasBornIn/isLocatedIn* <England>}
```

which returns every English politician born in England. By applying relaxation to the first triple pattern using rule (4) of RDFS entailment, it is possible to replace the class English_politicians by politicians. This relaxed query will return every politician who was born in England.

3.2.3. Query Approximation

For query approximation, we apply edit operations which transform a regular expression pattern P into a new expression pattern P'. Specifically, we apply the edit operations *deletion*, *insertion* and *substitution*, defined as follows (other possible edit operations are *transposition* and *inversion*, which we leave as future work):

$$A/p/B \rightsquigarrow (A/\epsilon/B) \quad \text{deletion}$$
$$A/p/B \rightsquigarrow (A/\_/B) \quad \text{substitution}$$
$$A/p/B \rightsquigarrow (A/\_/p/B) \quad \text{left insertion}$$
$$A/p/B \rightsquigarrow (A/p/\_/B) \quad \text{right insertion}$$

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. The application of an edit operation *op* has a cost c_{op} associated with it.

These rules can be applied to a URI p in order to approximate it to a regular expression P. We write $p \rightsquigarrow^* P$ if a sequence of edit operations can be applied 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.

The semantics of the APPROX operator in SPARQL^{AR} are as follows:

$$[[\text{APPROX}(x, p, y)]]_{G,K} = [[\langle x, p, y\rangle]]_G \cup \bigcup \{\langle \mu, cost + ecost(p, P)\rangle \mid p \rightsquigarrow^* P \wedge \langle \mu, cost\rangle \in [[\langle x, P, y\rangle]]_G\}$$

$$[[\text{APPROX}(x, P_1|P_2, y)]]_{G,K} = [[\text{APPROX}(x, P_1, y)]]_{G,K} \cup [[\text{APPROX}(x, P_2, y)]]_{G,K}$$

$$[[\text{APPROX}(x, P_1/P_2, y)]]_{G,K} = [[\text{APPROX}(x, P_1, z)]]_{G,K} \bowtie [[\text{APPROX}(z, P_2, y)]]_{G,K}$$

$$[[\text{APPROX}(x, P^*, y)]]_{G,K} = [[\langle x, \epsilon, y\rangle]]_G \cup [[\text{APPROX}(x, P, y)]]_{G,K} \cup \bigcup_{n\geq 1} \{\langle \mu, cost\rangle \mid \langle \mu, cost\rangle \in [[\text{APPROX}(x, P, z_1)]]_{G,K} \bowtie [[\text{APPROX}(z_1, P, z_2)]]_{G,K} \bowtie \cdots \bowtie [[\text{APPROX} \bowtie (z_n, P, y)]]_{G,K}\}$$

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.

Example 4. *Suppose that the user is looking for all discoveries made between 1700 and 1800 AD, and queries the YAGO dataset as follows:*

```
SELECT ?p ?z ?y WHERE{
?p discovered ?x . ?x discoveredOnDate ?y .
?x label ?z .
FILTER(?y >= 1700/1/1 and ?y <= 1800/1/1)}
```

Approximating the third triple pattern, it is possible to replace the predicate `label` *with* _*. The query will then return more information concerning that discovery, such as its preferred name (hasPreferredName) and the Wikipedia abstract (hasWikipediaAbstract).*

As another example, consider the following query, which is intended to return every German politician:

```
SELECT * WHERE{
?x isPoliticianOf ?y .
?x wasBornIn/isLocatedIn* <Germany>}
```

This query returns no answers since the relation isPoliticianOf only connects persons to states of the United States. If the first triple pattern is approximated by substituting the predicate isPoliticianOf by holdsPoliticalPosition, then the query will return the expected results.

3.3. Complexity of query answering

In [5] we studied the combined, data and query complexity of SPARQL^{AR}, extending the complexity results in [14,15,19] for simple SPARQL queries and in [1] for SPARQL with regular expression patterns to include our new flexible query constructs. The complexity of query evaluation is based on the following decision problem, which we denote EVALUATION: Given as input a graph $G = (N, D, E)$, an ontology K, a query Q and a pair $\langle \mu, cost\rangle$, is it the case that $\langle \mu, cost\rangle \in [[Q]]_{G,K}$? We showed the following results in [5]:

Theorem 1. *EVALUATION can be solved in time $O(|E| \cdot |Q|)$ for queries not containing regular expression patterns, and constructed using only the AND and FILTER operators.*

Theorem 2. *EVALUATION can be solved in time $O(|E| \cdot |Q|^2)$ for queries that may contain regular expression patterns and that are constructed using only the AND and FILTER operators.*

Theorem 3. *EVALUATION is NP-complete for queries that may contain regular expression patterns and that are constructed using only the AND and SELECT operators.*

Lemma 1. *EVALUATION of $[[APPROX(x, P, y)]]_{G,K}$ and $[[RELAX(x, P, y)]]_{G,K}$ can be accomplished in polynomial time.*

Theorem 4. *EVALUATION is NP-complete for queries that may contain regular expression patterns and that are constructed using the operators AND, FILTER, RELAX, APPROX and SELECT.*

Considering data complexity, the decision problem stated earlier becomes the following: Given as input a graph G, ontology K and a pair $\langle \mu, cost \rangle$, is it the case that $\langle \mu, cost \rangle \in [[Q]]_{G,K}$, with Q a fixed query?

Theorem 5. *EVALUATION is PTIME in data complexity for queries that may contain regular expression patterns and that are constructed using the operators AND, FILTER, RELAX, APPROX and SELECT.*

Results for query complexity follow from Lemma 1 and Theorems 1, 2 and 3. The complexity study of SPARQL^{AR} in [5] is summarised Figure 3, where the combined, data and query complexity are shown for specific language fragments and combinations of operators.

4. Query Processing

We evaluate SPARQL^{AR} queries by making use of a *query rewriting algorithm*, following a similar approach to [11,12,17]. In particular, given a query Q which may contain the APPROX and/or RELAX operators, we incrementally build a set of queries $\{Q_0, Q_1, \dots\}$ that do not contain these operators such that $\bigcup_i [[Q_i]]_{G,K} = [[Q]]_{G,K}$.

Our query rewriting algorithm — see Algorithm 2 below — starts by considering the query Q_0 which returns the exact answers to 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 apply one step of approximation (relaxation) to p, and we assign the cost of applying that approximation (relaxation) to the resulting query. From each query constructed in this way, we next generate a new set of queries applying a second step of approximation or relaxation. We continue to generate queries iteratively in this way. The cost of each query generated is the summed cost of the sequence of approximations or relaxations that have generated it. If the same query is generated more than once, only the one with the lowest cost is retained. Moreover, the set of queries generated is kept sorted by increasing cost. For practical reasons we limit the number of queries generated by bounding the cost of queries up to a maximum value c.

To compute the query answers — see Algorithm 1 — we apply an evaluation function, *eval*, to each query generated by the rewriting algorithm (in order of increasing 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, only the one with the lowest cost is retained. In Algorithm 1, *rewrite* is the query rewriting algorithm (Algorithm 2) and the set of mappings M is maintained in order of increasing cost.

In Algorithm 6, z, z_1 and z_2 are fresh new variables. The relaxTriplePattern function might generate regular expressions containing a URI $type^-$ which are matched to edges in E by reversing the subject and the object and using the property label *type*.

Example 5. *Consider the following ontology K (satisfying $K = extRed(K)$), which is a fragment of the YAGO knowledge base:*

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

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 $\langle Q', cost\rangle \in$ *rewrite(Q,c,K)* **do**
  foreach $\langle \mu, 0\rangle \in$ *eval(Q',G)* **do**
    $M := M \cup \{\langle \mu, cost\rangle\}$
return M;

Algorithm 2: Rewriting algorithm

input : Query Q_{AR}; approx/relax max cost c; Ontology K.
output: List of pairs query/cost sorted by cost.
Q_0 := remove APPROX and RELAX operators, and label triple patterns of Q_{AR};
queries:=$\{\langle Q_0, 0\rangle\}$; oldGeneration := $\{\langle Q_0, 0\rangle\}$;
while *oldGeneration* $\neq \emptyset$ **do**
  newGeneration := $\emptyset$;
  foreach $\langle Q, cost\rangle \in$ *oldGeneration* **do**
    foreach *labelled triple pattern* $\langle x, P, y\rangle$ *in* Q **do**
      $rew := \emptyset$;
      if $\langle x, P, y\rangle$ *is labelled with* A **then**
        $rew :=$ applyApprox(Q,$\langle x, P, y\rangle$);
      else if $\langle x, P, y\rangle$ *is labelled with* R **then**
        $rew :=$ applyRelax(Q,$\langle x, P, y\rangle$,K);
      foreach $\langle Q', cost'\rangle \in rew$ **do**
        if *(*$cost + cost' \leq c$*) and (*$\langle Q', cost + cost'\rangle \notin$ *newGeneration) and (if* $\langle Q', cost''\rangle \in$ *queries then* $cost + cost' < cost''$*)* **then**
          newGeneration := newGeneration $\cup \{\langle Q', cost + cost'\rangle\}$;
          queries:=addTo(queries, $\langle Q', cost + cost'\rangle$) ; `/* If` *queries* `contains` $\langle Q', cost''\rangle$`, this is replaced by` $\langle Q', cost + cost'\rangle$`. The elements of` *queries* `are also kept sorted by increasing cost. */`
  oldGeneration := newGeneration;
return queries;

Algorithm 3: applyApprox

input : Query Q; triple pattern $\langle x, P, y\rangle_A$.
output: Set of pairs query/cost S.
$S := \emptyset$;
foreach $\langle P', cost\rangle \in$ *approxRegex*(P) **do**
  $Q' :=$ replace $\langle x, P, y\rangle_A$ by $\langle x, P', y\rangle_A$ in Q;
  $S := S \cup \{\langle Q', cost\rangle\}$;
return S;

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^{AR} fragments.

Algorithm 4: approxRegex

input : Regular Expression P.
output: Set of pairs RegEx/cost T.
$T := \emptyset$;
if $P = p$ *where* p *is a URI* **then**
  $T := T \cup \{\langle \epsilon, cost_d \rangle\}$;
  $T := T \cup \{\langle \_, cost_s \rangle\}$;
  $T := T \cup \{\langle \_/p, cost_i \rangle\}$;
  $T := T \cup \{\langle p/\_, cost_i \rangle\}$;
else if $P = P_1/P_2$ **then**
  foreach $\langle P', cost \rangle \in approxRegex(P_1)$ **do**
    $T := T \cup \{\langle P'/P_2, cost \rangle\}$;
  foreach $\langle P', cost \rangle \in approxRegex(P_2)$ **do**
    $T := T \cup \{\langle P_1/P', cost \rangle\}$;
else if $P = P_1|P_2$ **then**
  foreach $\langle P', cost \rangle \in approxRegex(P_1)$ **do**
    $T := T \cup \{\langle P', cost \rangle\}$;
  foreach $\langle P', cost \rangle \in approxRegex(P_2)$ **do**
    $T := T \cup \{\langle P', cost \rangle\}$;
else if $P = P_1^*$ **then**
  foreach $\langle P', cost \rangle \in approxRegex(P_1)$ **do**
    $T := T \cup \{\langle (P_1^*)/P'/(P_1^*), cost \rangle\}$;
return T;

Algorithm 5: applyRelax

input : Query Q; triple pattern $\langle x, P, y\rangle_R$ of Q; Ontology K.
output: Set of pairs query/cost S.
$S := \emptyset$;
foreach $\langle\langle x', P', y'\rangle_R, cost\rangle \in relaxTriplePattern(\langle x, P, y\rangle)$ **do**
  $Q' :=$ replace $\langle x, P, y\rangle_R$ by $\langle x', P', y'\rangle_R$ in Q;
  $S := S \cup \{\langle Q', cost\rangle\}$;
return S;

Algorithm 6: relaxTriplePattern

input : Triple pattern $\langle x, P, y\rangle$; Ontology K.
output: Set of pairs triple pattern/cost T.
$T := \emptyset$;
if $P = p$ *where* p *is a URI* **then**
  foreach p' *such that* $\exists(p, sp, p') \in E_K$ **do**
    $T := T \cup \{\langle\langle x, p', y\rangle, cost_2\rangle\}$;
  foreach b *such that* $\exists(a, sc, b) \in E_K$ *and* $p = type$ *and* $y = a$ **do**
    $T := T \cup \{\langle\langle x, type, b\rangle, cost_4\rangle\}$;
  foreach b *such that* $\exists(a, sc, b) \in E_K$ *and* $p = type^-$ *and* $x = a$ **do**
    $T := T \cup \{\langle\langle b, type^-, y\rangle, cost_4\rangle\}$;
  foreach a *such that* $\exists(p, dom, a) \in E_K$ *and* y *is a URI or a Literal* **do**
    $T := T \cup \{\langle\langle x, type, a\rangle, cost_5\rangle\}$;
  foreach a *such that* $\exists(p, range, a) \in E_K$ *and* x *is a URI* **do**
    $T := T \cup \{\langle\langle a, type^-, y\rangle, cost_6\rangle\}$;
else if $P = P_1/P_2$ **then**
  foreach $\langle\langle x', P', z\rangle, cost\rangle \in relaxTriplePattern(\langle x, P_1, z\rangle)$ **do**
    $T := T \cup \{\langle\langle x', P'/P_2, y\rangle, cost\rangle\}$;
  foreach $\langle\langle z, P', y'\rangle, cost\rangle \in relaxTriplePattern(\langle z, P_2, y\rangle)$ **do**
    $T := T \cup \{\langle\langle x, P_1/P', y'\rangle, cost\rangle\}$;
else if $P = P_1|P_2$ **then**
  foreach $\langle\langle x', P', y'\rangle, cost\rangle \in relaxTriplePattern(\langle x, P_1, y\rangle)$ **do**
    $T := T \cup \{\langle\langle x', P', y'\rangle, cost\rangle\}$;
  foreach $\langle\langle x', P', y'\rangle, cost\rangle \in relaxTriplePattern(\langle x, P_2, y\rangle)$ **do**
    $T := T \cup \{\langle\langle x', P', y'\rangle, cost\rangle\}$;
else if $P = P_1^*$ **then**
  foreach $\langle\langle z_1, P', z_2\rangle, cost\rangle \in relaxTriplePattern((\langle z_1, P_1, z_2\rangle)$ **do**
    $T := T \cup \{\langle\langle x, P_1^*/P'/P_1^*, y\rangle, cost\rangle\}$;
  foreach $\langle\langle x', P', z\rangle, cost\rangle \in relaxTriplePattern((\langle x, P_1, z\rangle)$ **do**
    $T := T \cup \{\langle\langle x', P'/P_1^*, y\rangle, cost\rangle\}$;
  foreach $\langle\langle z, P', y'\rangle, cost\rangle \in relaxTriplePattern((\langle z, P_1, y\rangle)$ **do**
    $T := T \cup \{\langle\langle x, P_1^*/P', y'\rangle, cost\rangle\}$;
return T;

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

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

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:

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

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).

4.1. Correctness of the Rewriting Algorithm

We now discuss the soundness, completeness and termination of the rewriting algorithm. As we stated earlier, this takes as input a cost that limits the number of queries generated. Therefore the classic definitions of soundness and completeness need to be modified. To handle this, we use an operator $CostProj(M, c)$ to select mappings with a cost less than or equal to a given value c from a set M of pairs of the form $\langle \mu, cost \rangle$. We denote by $rew(Q)_c$ the set of queries generated by the rewriting algorithm from an initial query Q which have cost less than or equal to c.

Definition 8 (Containment)**.** *Given a graph G, an ontology K, and queries Q and Q', $[[Q]]_{G,K} \subseteq [[Q']]_{G,K}$ if for each pair $\langle \mu, c \rangle \in [[Q]]_{G,K}$ there exists a pair $\langle \mu, c \rangle \in [[Q']]_{G,K}$.*

Definition 9 (Soundness)**.** *The rewriting of Q, $rew(Q)_c$, is* sound *if the following holds: $\bigcup_{Q' \in rew(Q)_c} [[Q']]_{G,K} \subseteq CostProj([[Q]]_{G,K}, c)$ for every graph G and ontology K.*

Definition 10 (Completeness)**.** *The rewriting of Q, $rew(Q)_c$, is* complete *if the following holds: $CostProj([[Q]]_{G,K}, c) \subseteq \bigcup_{Q' \in rew(Q)_c} [[Q']]_{G,K}$ for every graph G and ontology K.*

To show the soundness and completeness of the query rewriting algorithm we will require the following lemmas:

Lemma 2. *Given four sets of evaluation results M_1, M_2, M_1' and M_2' such that $M_1 \subseteq M_1'$ and $M_2 \subseteq M_2'$, it holds that:*

$$M_1 \cup M_2 \subseteq M_1' \cup M_2' \quad (1)$$
$$M_1 \bowtie M_2 \subseteq M_1' \bowtie M_2' \quad (2)$$

Proof. (1) From the definition of union, it follows that $M_1' \cup M_2'$ contains every mapping from M_1 and M_2, and therefore the statement holds.

(2) From the definition of join, $M_1 \bowtie M_2$ contains a mapping $\mu_1 \cup \mu_2$ for every pair of compatible mappings $\langle \mu_1, cost_1 \rangle \in M_1$ and $\langle \mu_2, cost_2 \rangle \in M_2$. Since M_1' and M_2' also contain μ_1 and μ_2, respectively, then $M_1' \bowtie M_2'$ will also contain $\mu_1 \cup \mu_2$. □

Observation 1. *By the semantics of RELAX and APPROX, we observe that given a triple pattern $\langle x, P, y \rangle$, $[[\langle x, P, y \rangle]]_{G,K} \subseteq [[APPROX(x, P, y)]]_{G,K}$ and $[[\langle x, P, y \rangle]]_{G,K} \subseteq [[RELAX(x, P, y)]]_{G,K}$ for every graph G and ontology K.*

Lemma 3. *Given queries Q_1 and Q_2, graph G and ontology K the following equations hold:*

$$CostProj([[Q_1]]_{G,K} \bowtie [[Q_2]]_{G,K}, c) = CostProj(CostProj([[Q_1]]_{G,K}, c) \bowtie CostProj([[Q_2]]_{G,K}, c), c)$$
$$CostProj([[Q_1]]_{G,K} \cup [[Q_2]]_{G,K}, c) =$$

$$CostProj([[Q_1]]_{G,K}, c) \cup$$
$$CostProj([[Q_2]]_{G,K}, c)$$

Proof. Considering the right hand side (RHS) of the first equation, we know that each pair $\langle \mu, cost \rangle$ in the RHS has $cost \leq c$ and is equal to $\langle \mu_1, cost_1 \rangle \bowtie \langle \mu_2, cost_2 \rangle$, where $cost_1 \leq c$, $cost_2 \leq c$, $\langle \mu_1, cost_1 \rangle \in [[Q_1]]_{G,K}$ and $\langle \mu_2, cost_2 \rangle \in [[Q_2]]_{G,K}$. Therefore, the pair $\langle \mu, cost \rangle$ must also be contained in the left hand side (LHS) of the equation. Conversely, for each pair $\langle \mu, cost \rangle$ in the LHS, we know that $cost \leq c$ and that there must exist a pair $\langle \mu_1, cost_1 \rangle \in [[Q_1]]_{G,K}$ and a pair $\langle \mu_2, cost_2 \rangle \in [[Q_2]]_{G,K}$ such that $\langle \mu_1, cost_1 \rangle \bowtie \langle \mu_2, cost_2 \rangle = \langle \mu, cost \rangle$. Moreover, since $cost = cost_1 + cost_2$ we know that $cost_1 \leq c$ and $cost_2 \leq c$. Therefore, we can conclude that $\langle \mu, cost \rangle$ must also be contained in the RHS of the equation.

For the second equation it is easy to verify that every pair $\langle \mu, cost \rangle$ is in $CostProj([[Q_1]]_{G,K} \cup [[Q_2]]_{G,K}, c)$ if and only if it is contained either in $CostProj([[Q_1]]_{G,K}, c)$ or in $CostProj([[Q_2]]_{G,K}, c)$, or in both. □

Theorem 3. *The Rewriting Algorithm is sound and complete.*

Proof. For ease of reading, in this proof we will replace the operators APPROX and RELAX with A and R respectively. We divide the proof into three parts: (1) The first part shows that for $c \geq 0$ and relaxed or approximated triple patterns of the form $\langle x, p, y \rangle$, the functions approxRegex and relaxTriplePattern generate sound and complete triple patterns. (2) The second part of the proof shows that the algorithm is sound and complete for approximated and relaxed triple patterns containing any regular expression. (3) Finally, we show that the algorithm is sound and complete for general queries Q, i.e. we show that the following holds for any query Q, graph G and ontology K:

$$\text{CostProj}([[Q]]_{G,K}, c) \subseteq \bigcup_{Q' \in rew(Q)_c} [[Q']]_{G,K} \subseteq \text{CostProj}([[Q]]_{G,K}, c)$$

(1) In this first part we show that for any triple pattern $\langle x, p, y \rangle$ and cost $c \geq 0$ the following holds:

$$\text{CostProj}([[A/R(x,p,y)]]_{G,K}, c) = \bigcup_{t' \in rew(A/R(x,p,y))_c} [[t']]_{G,K}$$

We show this by induction on the cost c. For the base case of $c = 0$ we need to show that:

$$\text{CostProj}([[A/R(x,p,y)]]_{G,K}, 0) = \bigcup_{t' \in rew(A/R(x,p,y))_0} [[t']]_{G,K} \quad (3)$$

On the LHS, since the costs of applying APPROX and RELAX have cost greater than zero, the CostProj operator will only return the exact answers of the query, in other words it will exclude the answers generated by the APPROX and RELAX operators. On the RHS, the rewriting algorithm will not return queries with associated cost greater than 0 and therefore will just return the original query unchanged. This, when evaluated, will therefore also return the exact answers of the query. So (3) holds.

When c is greater than 0 we consider two different cases, one for APPROX and the other for RELAX:

(a) Approximation. For approximation, we show the following by induction on the cost c:

$$\text{CostProj}([[A(x,p,y)]]_{G,K}, c) = \bigcup_{t' \in rew(A(x,p,y))_c} [[t']]_{G,K} \quad (4)$$

The induction hypothesis is that

$$\text{CostProj}([[A(x,p,y)]]_{G,K}, c) = \bigcup_{t' \in rew(A(x,p,y))_c} [[t']]_{G,K} \quad (5)$$

for $c = i\alpha + j\beta + k\gamma$ for all $i, j, k \geq 0$, where α, β, γ are the cost of the insertion, deletion and substitution edit operations, respectively. We have already shown the base case of $i = j = k = 0$. We now show that (5) is true when one of, i, j or k is incremented by 1.

Considering the RHS of equation (4), when we apply one step of approximation to a triple pattern the algorithm generates a set of triple patterns. These triple patterns will be recursively rewritten by the algorithm. Therefore, by applying every possible edit operation to the original triple pattern, we have that:

$$\bigcup_{t' \in rew(A(x,p,y))_c} [[t']]_{G,K} =$$
$$[[\langle x,p,y\rangle]]_{G,K} \cup$$
$$\bigcup_{t' \in rew(A(x,\_/p,y))_{c-\alpha}} [[t']]_{G,K} \cup$$
$$\bigcup_{t' \in rew(A(x,p/\_,y))_{c-\alpha}} [[t']]_{G,K} \cup$$
$$\bigcup_{t' \in rew(A(x,\epsilon,y))_{c-\beta}} [[t']]_{G,K} \cup$$
$$\bigcup_{t' \in rew(A(x,\_,y))_{c-\gamma}} [[t']]_{G,K}$$

Considering the LHS of equation (4), again by the semantics of approximation, we have that:

$$\text{CostProj}([[A(x,p,y)]]_{G,K}, c) =$$
$$[[\langle x,p,y\rangle]]_{G,K} \cup$$
$$\text{CostProj}([[A(x,\_/p,y)]]_{G,K}, c-\alpha) \cup$$
$$\text{CostProj}([[A(x,p/\_,y)]]_{G,K}, c-\alpha) \cup$$
$$\text{CostProj}([[A(x,\epsilon,y)]]_{G,K}, c-\beta) \cup$$
$$\text{CostProj}([[A(x,\_,y)]]_{G,K}, c-\gamma)$$

Combining the last two into a single equation, we therefore need to show that:

$$[[\langle x,p,y\rangle]]_{G,K} \cup$$
$$\bigcup_{t' \in rew(A(x,\_/p,y))_{c-\alpha}} [[t']]_{G,K} \cup$$
$$\bigcup_{t' \in rew(A(x,p/\_,y))_{c-\alpha}} [[t']]_{G,K} \cup$$
$$\bigcup_{t' \in rew(A(x,\epsilon,y))_{c-\beta}} [[t']]_{G,K} \cup$$
$$\bigcup_{t' \in rew(A(x,\_,y))_{c-\gamma}} [[t']]_{G,K}$$
$$=$$
$$[[\langle x,p,y\rangle]]_{G,K} \cup$$
$$\text{CostProj}([[A(x,\_/p,y)]]_{G,K}, c-\alpha) \cup$$
$$\text{CostProj}([[A(x,p/\_,y)]]_{G,K}, c-\alpha) \cup$$
$$\text{CostProj}([[A(x,\epsilon,y)]]_{G,K}, c-\beta) \cup$$
$$\text{CostProj}([[A(x,\_,y)]]_{G,K}, c-\gamma)$$

Given Lemma 2, it is sufficient to show that all the following equations hold:

$$[[\langle x,p,y\rangle]]_{G,K} = [[\langle x,p,y\rangle]]_{G,K} \tag{6}$$

$$\bigcup_{t' \in rew(A(x,\_/p,y))_{c-\alpha}} [[t']]_{G,K} = \text{CostProj}([[A(x,\_/p,y)]]_{G,K}, c-\alpha) \tag{7}$$

$$\bigcup_{t' \in rew(A(x,p/\_,y))_{c-\alpha}} [[t']]_{G,K} = \text{CostProj}([[A(x,p/\_,y)]]_{G,K}, c-\alpha) \tag{8}$$

$$\bigcup_{t' \in rew(A(x,\epsilon,y))_{c-\beta}} [[t']]_{G,K} = \text{CostProj}([[A(x,\epsilon,y)]]_{G,K}, c-\beta) \tag{9}$$

$$\bigcup_{t' \in rew(A(x,\_,y))_{c-\gamma}} [[t']]_{G,K} = \text{CostProj}([[A(x,\_,y)]]_{G,K}, c-\gamma) \tag{10}$$

Equation (6) is trivially true. Equations (9) and (10) hold since on the LHS, $rew(A(x,\epsilon,y))_c$ and $rew(A(x,\_,y))_c$ contain only (x,ϵ,y) and $(x,\_,y)$ respectively, for any $c \geq 0$, and on the RHS, by the semantics of approximation, we know that $[[A(x,\epsilon,y)]]_{G,K} = [[x,\epsilon,y]]_{G,K}$ and $[[A(x,\_,y)]]_{G,K} = [[x,\_,y]]_{G,K}$.

For equation (7), considering the semantics of approximation with concatenation of paths, the LHS of the equation can be rewritten in the following way since we know that we will not apply any step of approximation to $A(x,\_,z)$.:

$$[[\langle x,\_,z\rangle]]_{G,K}) \bowtie (\bigcup_{t' \in rew(A(z,p,y))_{c-\alpha}} [[t']]_{G,K})$$

Applying Lemma 3 we can rewrite the RHS of (7) to:

$$\text{CostProj}(\text{CostProj}([[A(x,\_,z)]]_{G,K}, c-\alpha) \bowtie \text{CostProj}([[A(z,p,y)]]_{G,K}, c-\alpha), c-\alpha)$$

It is possible to drop the outer CostProj since the query $[[A(x,\_,z)]]_{G,K}$ returns only mappings with associated cost 0, obtaining:

$$\text{CostProj}([[A(x,\_,z)]]_{G,K}, c-\alpha) \bowtie \text{CostProj}([[A(z,p,y)]]_{G,K}, c-\alpha)$$

Therefore we need to show that the following holds:

$$[[\langle x,\_,z\rangle]]_{G,K}) \bowtie (\bigcup_{t' \in rew(A(z,p,y))_{c-\alpha}} [[t']]_{G,K}) = \text{CostProj}([[A(x,\_,z)]]_{G,K}, c-\alpha) \bowtie \text{CostProj}([[A(z,p,y)]]_{G,K}, c-\alpha)$$

Given Lemma 2 it is sufficient to show that:

$$[[\langle x,\_,z\rangle]]_{G,K} = \text{CostProj}([[A(x,\_,z)]]_{G,K}, c-\alpha) \tag{11}$$

$$\bigcup_{t' \in rew(A(z,p,y))_{c-\alpha}} [[t']]_{G,K} = \text{CostProj}([[A(z,p,y)]]_{G,K}, c-\alpha) \tag{12}$$

Equation (11) holds by similar reasoning to equation (10). Equation (12) holds by the induction hypothesis.

Equation (8) can be shown to hold by similar reasoning to equation (7). We conclude that equation (4) holds for every $c \geq 0$.

(b) Relaxation. For relaxation, we show the following by induction on the cost c:

$$\text{CostProj}([[R(x,p,y)]]_{G,K}, c) = \bigcup_{t' \in rew(R(x,p,y))_c} [[t']]_{G,K} \tag{13}$$

The induction hypothesis is that

$$\text{CostProj}([[R(x,p,y)]]_{G,K}, c) = \bigcup_{t' \in rew(A(x,p,y))_c} [[t']]_{G,K} \tag{14}$$

for $c = i\alpha + j\beta + k\gamma + l\delta$ for all $i, j, k, l \geq 0$, where α, β, γ, δ are the costs of the four relaxation operations arising from rules 2, 4, 5 and 6, respectively, of Figure 1. We have already shown the base case of $i = j = k = l = 0$. We now show that (14) holds when one of, i, j, k or l is incremented by 1. Similarly to the reasoning for approximation in part (a), we need to show that:

$$\begin{gathered}
[[\langle x,p,y\rangle]]_{G,K} \cup \\
\text{CostProj}([[R(x,p',y)]]_{G,K}, c-\alpha) \cup \\
\text{CostProj}([[R(x,type,a)]]_{G,K}, c-\beta) \cup \\
\text{CostProj}([[R(a,type^-,x)]]_{G,K}, c-\beta) \cup \\
\text{CostProj}([[R(x,type,a)]]_{G,K}, c-\gamma) \cup \\
\text{CostProj}([[R(a,type^-,x)]]_{G,K}, c-\delta) \\
= \\
[[\langle x,p,y\rangle]]_{G,K} \cup \\
\bigcup_{t' \in rew(R(x,p',y))_{c-\alpha}} [[t']]_{G,K} \cup \\
\bigcup_{t' \in rew(R(x,type,a))_{c-\beta}} [[t']]_{G,K} \cup \\
\bigcup_{t' \in rew(R(a,type^-,x))_{c-\beta}} [[t']]_{G,K} \cup \\
\bigcup_{t' \in rew(R(x,type,a))_{c-\gamma}} [[t']]_{G,K} \cup \\
\bigcup_{t' \in rew(R(a,type^-,x))_{c-\delta}} [[t']]_{G,K}
\end{gathered}$$

Given Lemma 2 it is sufficient to show that the following hold::

$$[[\langle x,p,y\rangle]]_{G,K} = [[\langle x,p,y\rangle]]_{G,K} \tag{15}$$

$$\text{CostProj}([[R(x,p',y)]]_{G,K}, c-\alpha) = \bigcup_{t' \in rew(R(x,p',y))_{c-\alpha}} [[t']]_{G,K} \tag{16}$$

$$\text{CostProj}([[R(x,type,a)]]_{G,K}, c-\beta) = \bigcup_{t' \in rew(R(x,type,a))_{c-\beta}} [[t']]_{G,K} \tag{17}$$

$$\text{CostProj}([[R(a,type^-,x)]]_{G,K}, c-\beta) = \bigcup_{t' \in rew(R(a,type^-,x))_{c-\beta}} [[t']]_{G,K} \tag{18}$$

$$\text{CostProj}([[R(x,type,a)]]_{G,K}, c-\gamma) = \bigcup_{t' \in rew(R(x,type,a))_{c-\gamma}} [[t']]_{G,K} \tag{19}$$

$$\text{CostProj}([[R(a,type^-,x)]]_{G,K}, c-\delta) = \bigcup_{t' \in rew(R(a,type^-,x))_{c-\delta}} [[t']]_{G,K} \tag{20}$$

Equation (15) is trivially true. Equations (16-20) can be rewritten as the general case of the induction hypothesis for some $c \geq 0$. Therefore equations (16-20) hold by the induction hypothesis. We conclude that equation (13) holds for every $c \geq 0$.

(2) Now we need to show that approxRegex and relaxTriplePattern are sound and complete for triple patterns containing any regular expression. In part (1) we have demonstrated soundness and completeness for triple patterns containing a single predicate, p:

$$\begin{gathered}
\text{CostProj}([[A/R(x,p,y)]]_{G,K}, c) = \\
\bigcup_{t' \in rew(A/R(x,p,y))_c} [[t']]_{G,K}
\end{gathered}$$

This is our base case. We now show soundness and completeness by structural induction, considering the three different operators used to construct a regular expression: concatenation, disjunction and Kleene-Closure.

(a) Concatenation. The induction hypothesis is that the following equations hold for any regular expressions P_1 and P_2:

$$\text{CostProj}([[A/R(x,P_1,y)]]_{G,K}, c) = \bigcup_{t' \in rew(A/R(x,P_1,y))_c} [[t']]_{G,K} \tag{21}$$

$$\text{CostProj}([[A/R(x,P_2,y)]]_{G,K},c) = \bigcup_{t' \in rew(A/R(x,P_2,y))_c} [[t']]_{G,K} \quad (22)$$

We now show that the following holds:

$$\text{CostProj}([[A/R(x,P_1/P_2,y)]]_{G,K},c) = \bigcup_{t' \in rew(A/R(x,P_1/P_2,y))_c} [[t']]_{G,K} \quad (23)$$

When the approxRegex and relaxTriplePattern functions are passed as input a triple pattern of the form $A/R(x,P_1/P_2,y)$, this is split into two triple patterns: $A/R(x,P_1,z)$ and $A/R(z,P_2,y)$. Both of these triple patterns are passed recursively to the approxRegex and relaxTriplePattern functions which return two sets of triple patterns that will be joined with the AND operator. Therefore the RHS of equation (23) can be written in the following way:

$$\text{CostProj}(\bigcup_{t' \in rew(A/R(x,P_1,z))_c}[[t']]_{G,K} \bowtie \bigcup_{t' \in rew(A/R(z,P_2,y))_c}[[t']]_{G,K}, c)$$

Given the semantics of approximation and relaxation with concatenation of paths, the LHS of equation (23) can be written as follows:

$$\text{CostProj}([[A/R(x,P_1,z)]]_{G,K} \bowtie [[A/R(z,P_2,y)]]_{G,K}, c)$$

which by Lemma 3 is equal to:

$$\text{CostProj}(\text{CostProj}([[A/R(x,P_1,z)]]_{G,K},c) \bowtie \text{CostProj}([[A/R(z,P_2,y)]]_{G,K},c),c)$$

We therefore need to show that:

$$\text{CostProj}(\text{CostProj}([[A/R(x,P_1,z)]]_{G,K},c) \bowtie \text{CostProj}([[A/R(z,P_2,y)]]_{G,K},c),c) = \text{CostProj}(\bigcup_{t' \in rew(A/R(x,P_1,z))_c}[[t']]_{G,K} \bowtie \bigcup_{t' \in rew(A/R(z,P_2,y))_c}[[t']]_{G,K}, c)$$

It is possible to drop the outer CostProj operators on both sides of the above equation. Applying Lemma 2 it is sufficient to show that:

$$\text{CostProj}([[A/R(x,P_1,z)]]_{G,K},c) = \bigcup_{t' \in rew(A/R(x,P_1,z))_c}[[t']]_{G,K}$$
$$\text{CostProj}([[A/R(z,P_2,y)]]_{G,K},c) = \bigcup_{t' \in rew(A/R(z,P_2,y))_c}[[t']]_{G,K}$$

These equations hold by the induction hypothesis. Therefore equation (23) holds.

(b) Disjunction. Similarly to concatenation, our induction hypothesis is that equations (21) and (22) hold for any regular expressions P_1 and P_2. We now show that the following equation holds:

$$\text{CostProj}([[A/R(x,P_1|P_2,y)]]_{G,K},c) = \bigcup_{t' \in rew(A/R(x,P_1|P_2,y))_c} [[t']]_{G,K} \quad (24)$$

When the approxRegex and relaxTriplePattern functions are passed as input a triple pattern of the form $A/R(x,P_1|P_2,y)$, this is split into two triple patterns: $A/R(x,P_1,y)$ and $A/R(x,P_2,y)$. Both of these triple patterns are passed recursively to the approxRegex and relaxTriplePattern functions which will return two sets of triple patterns that will be combined with the UNION operator. Therefore the RHS of equation (24) can be written as follows:

$$\bigcup_{t' \in rew(A/R(x,P_1,y))_c}[[t']]_{G,K} \cup \bigcup_{t' \in rew(A/R(x,P_2,y))_c}[[t']]_{G,K}$$

Given the semantics of approximation and relaxation with disjunction of paths, we can write the LHS of equation (24) as follows:

$$\text{CostProj}([[A/R(x,P_1,y)]]_{G,K} \cup [[A/R(x,P_2,y)]]_{G,K}, c)$$

which by Lemma 3 is equal to:

$$\text{CostProj}([[A/R(x,P_1,y)]]_{G,K},c) \cup \text{CostProj}([[A/R(x,P_2,y)]]_{G,K},c)$$

We therefore need to show that:

$$\text{CostProj}([[A/R(x,P_1,y)]]_{G,K},c) \cup \text{CostProj}([[A/R(x,P_2,y)]]_{G,K},c) = \bigcup_{t' \in rew(A/R(x,P_1,y))_c}[[t']]_{G,K} \cup \bigcup_{t' \in rew(A/R(x,P_2,y))_c}[[t']]_{G,K}$$

By Lemma 2 it is sufficient to show that:

$$\text{CostProj}([[A/R(x,P_1,y)]]_{G,K},c) = \bigcup_{t' \in rew(A/R(x,P_1,y))_c}[[t']]_{G,K}$$
$$\text{CostProj}([[A/R(x,P_2,y)]]_{G,K},c) = \bigcup_{t' \in rew(A/R(x,P_2,y))_c}[[t']]_{G,K}$$

These equations hold by the induction hypothesis. Therefore equation (24) holds.

(c) Kleene-Closure. Our induction hypothesis in this case is that

$$\text{CostProj}([[A/R(x, P^n, y)]]_{G,K}, c) = \bigcup_{t' \in rew(A/R(x,P^n,y))_c} [[t']]_{G,K}$$

for any regular expression P and any $n \geq 0$, where P^n denotes the regular expression $P/P/\ldots/P$ in which P appears n times. For the base case of $n = 0$, where $P^n = \epsilon$, the equation is trivially true since $rew(A(x, \epsilon, y))_c$ contains only the query (x, ϵ, y). We now show that the following holds:

$$\text{CostProj}([[A/R(x, P^{n+1}, y)]]_{G,K}, c) = \bigcup_{t' \in rew(A/R(x,P^{n+1},y))_c} [[t']]_{G,K} \quad (25)$$

The approxRegex function rewrites an approximated triple pattern on the RHS of the equation in the following way: $A(x, P^i/P/P^j, y))$ for arbitrarily chosen i, j satisfying $i + j = n$. It then splits this into three triple patterns, $A(x, P^i, z_1)$, $A(z_1, P, z_2)$ and $A(z_2, P^j, y)$. Therefore the RHS of (25) becomes:

$$\begin{aligned}\text{CostProj}(&\bigcup_{t' \in rew(A(x,P^i,z_1))_c} [[t']]_{G,K} \bowtie \\ &\bigcup_{t' \in rew(A(z_1,P,z_2))_c} [[t']]_{G,K} \bowtie \\ &\bigcup_{t' \in rew(A(z_2,P^j,y))_c} [[t']]_{G,K}, c)\end{aligned} \quad (26)$$

We have added the CostProj operator in order to follow the behaviour of the algorithm that excludes queries with associated cost greater than c.

Knowing that $\mathcal{L}(P^i/P/P^j) = \mathcal{L}(P^{n+1})$ and by the semantics of approximation with concatenation of paths, we can write the LHS as:

$$\text{CostProj}([[A(x, P^i, z_1)]]_{G,K} \bowtie [[A(z_1, P, z_2)]]_{G,K} \bowtie [[A(z_2, P^j, y)]]_{G,K}, c)$$

Applying Lemma 3, this can be rewritten as:

$$\begin{aligned}\text{CostProj}(\text{CostProj}([[A(x, P^i, z_1)]]_{G,K}, c) \bowtie \\ \text{CostProj}([[A(z_1, P, z_2)]]_{G,K}, c) \bowtie \\ \text{CostProj}([[A(z_2, P^j, y)]]_{G,K}, c), c)\end{aligned} \quad (27)$$

Combining (26) and (27) and removing the outer CostProj operator on both hand sides we therefore need to show that:

$$\begin{aligned}&\text{CostProj}([[A(x, P^i, z_1)]]_{G,K}, c) \bowtie \\ &\text{CostProj}([[A(z_1, P, z_2)]]_{G,K}, c) \bowtie \\ &\text{CostProj}([[A(z_2, P^j, y)]]_{G,K}, c) = \\ &\bigcup_{t' \in rew(A(x,P^i,z_1))_c} [[t']]_{G,K} \bowtie \\ &\bigcup_{t' \in rew(A/R(z_1,P,z_2))_c} [[t']]_{G,K} \bowtie \\ &\bigcup_{t' \in rew(A/R(z_2,P^j,y))_c} [[t']]_{G,K}\end{aligned}$$

By Lemma 2 it is sufficient to show that:

$$\text{CostProj}([[A(x, P^i, z_1)]]_{G,K}, c) = \bigcup_{t' \in rew(A(x,P^i,z_1))_c} [[t']]_{G,K} \quad (28)$$

$$\text{CostProj}([[A(z_1, P, z_2)]]_{G,K}, c) = \bigcup_{t' \in rew(A(z_1,P,z_2))_c} [[t']]_{G,K} \quad (29)$$

$$\text{CostProj}([[A(z_2, P^j, y)]]_{G,K}, c) = \bigcup_{t' \in rew(A(z_2,P^j,y))_c} [[t']]_{G,K} \quad (30)$$

Equations (28,29,30) hold by the induction hypothesis since i and j are both less than n, therefore equation (25) holds.

The same reasoning applies for the relaxTriplePattern function applied to a relaxed triple pattern on the RHS of (25) with the difference that it rewrites the triple pattern in 3 different ways: $R(x, P^i/P/P^j, y)$ (for arbitrarily chosen i, j satisfying $i+j = n$), $R(x, P/P^n, y)$ and $R(x, P^n/P, y)$. It is possible to apply the same steps of the proof as for approxRegex, noticing that $\mathcal{L}(P/P^n) = \mathcal{L}(P^{n+1})$ and $\mathcal{L}(P^n/P) = \mathcal{L}(P^{n+1})$.

(3) General queries. We now show that the algorithm is sound and complete for any query that may contain approximation and relaxation. As the base case we have the case of a query comprising a single triple pattern, which has been shown in part (2) of the proof:

$$\text{CostProj}([[A/R(x, P, y)]]_{G,K}, c) = \bigcup_{t' \in rew(A/R(x,P,y))_c} [[t']]_{G,K}$$

Consider now a query $Q = t$ AND Q' with t being an arbitrary triple pattern of the query Q. The induction hypothesis is that:

$$\text{CostProj}([[Q']]_{G,K}, c) = \bigcup_{Q'' \in rew(Q')_c} [[Q'']]_{G,K} \tag{31}$$

We now show that the following holds.

$$\text{CostProj}([[Q]]_{G,K}, c) = \bigcup_{Q'' \in rew(Q)_c} [[Q'']]_{G,K} \tag{32}$$

The LHS of equation (32) is equivalent to the following by the semantics of the AND operator:

$$\text{CostProj}([[t]]_{G,K} \bowtie [[[Q']]_{G,K}, c)$$

Applying Lemma 3 we can rewrite this as follows:

$$\begin{gathered}\text{CostProj}(\text{CostProj}([[t]]_{G,K}, c) \bowtie \\ \text{CostProj}([[]Q']]_{G,K}, c), c)\end{gathered} \tag{33}$$

For the RHS of equation (32) we have to consider two different cases: either t is a simple triple pattern or it contains the RELAX or APPROX operators. If we consider the former case then we rewrite the RHS of equation (32) to:

$$[[t]]_{G,K} \bowtie \bigcup_{Q'' \in rew(Q')_c} [[Q'']]_{G,K} \tag{34}$$

Combining (33) and (34) we need to show that:

$$\begin{gathered}\text{CostProj}(\text{CostProj}([[t]]_{G,K}, c) \bowtie \\ \text{CostProj}([[]Q']]_{G,K}, c), c) = [[t]]_{G,K} \bowtie \\ \bigcup_{Q'' \in rew(Q')_c} [[Q'']]_{G,K}\end{gathered}$$

We are able to drop the outer CostProj operator and the CostProj applied to the triple pattern t since $[[t]]_{G,K}$ returns mappings with cost 0. The resulting equation is as follows:

$$\begin{gathered}[[t]]_{G,K} \bowtie \text{CostProj}([[]Q']]_{G,K}, c) = [[t]]_{G,K} \bowtie \\ \bigcup_{Q'' \in rew(Q')_c} [[Q'']]_{G,K}\end{gathered}$$

Applying Lemma 2 it is sufficient to show that:

$$[[t]]_{G,K} = [[t]]_{G,K} \tag{35}$$

$$\text{CostProj}([[Q']]_{G,K}, c) = \bigcup_{Q'' \in rew(Q')_c} [[Q'']]_{G,K} \tag{36}$$

Equation (35) is trivially true and equation (36) holds by the induction hypothesis. Therefore equation (32) holds in the case of t being a simple triple pattern.

If t contains the APPROX or RELAX operators then the RHS of (32) is:

$$\bigcup_{t' \in rew(t)_c} [[t']]_{G,K} \bowtie \bigcup_{Q'' \in rew(Q')_c} [[Q'']]_{G,K}$$

Therefore, combining the latter with (33), we have:

$$\begin{gathered}\text{CostProj}(\text{CostProj}([[t]]_{G,K}, c) \bowtie \\ \text{CostProj}([[]Q']]_{G,K}, c), c) = \\ \text{CostProj}((\bigcup_{t' \in rew(t)_c} [[t']]_{G,K}) \bowtie \\ (\bigcup_{Q'' \in rew(Q')_c} [[Q'']]_{G,K}), c)\end{gathered}$$

(We have added the CostProj operator on the RHS of the equation in order to follow the behaviour of the algorithm that excludes queries with associated cost greater than c). Removing the CostProj from both hand sides of the equation and applying Lemma 2, it is sufficient to show that:

$$\text{CostProj}([[t]]_{G,K}, c) = \bigcup_{t' \in rew(t)_c} [[t']]_{G,K} \tag{37}$$

$$\text{CostProj}([[]Q']]_{G,K}, c) = \bigcup_{Q'' \in rew(Q')_c} [[Q'']]_{G,K} \tag{38}$$

Equation (37) holds since approxRegex and relaxTriplePattern are sound and complete as shown in step (2) of the proof. Equation (38) holds by the induction hypothesis. Therefore equation (32) holds in the case of t containing the APPROX and RELAX operators.

□

4.2. Termination of the Rewriting Algorithm

We are able to show that the rewriting algorithm terminates after a finite number of steps:

Theorem 4. *Given a query Q, ontology K and maximum query cost c, the Rewriting Algorithm terminates after at most $\lceil c/c' \rceil$ iterations, where c' is the lowest cost of an edit or relaxation operation, assuming that $c' > 0$.*

Proof. The algorithm terminates when the set *oldGeneration* is empty. At the end of each cycle, *oldGeneration* is assigned the value of *newGeneration*. During each cycle, elements are added to *newGeneration* only when new queries are generated and have cost less than c or already generated queries are generated again at a lesser cost (also less than c).

On each cycle of the algorithm, each query generated by *applyApprox* or *applyRelax* has cost at least c' plus the cost of the query from which it is generated. Since we start from query Q_0 which has cost 0, every query generated during the nth cycle will have cost greater than or equal to $n \cdot c'$. When $n \cdot c' > c$ the algorithm will not add any queries to *newGeneration*. Therefore, the algorithm will stop after at most $\lceil c/c' \rceil$ iterations. □

5. Experimental Results

We have implemented our query evaluation algorithms described above and have conducted empirical trials over the YAGO dataset and the Lehigh University Benchmark (LUBM)[3]. Our empirical results using the LUBM are described in [5] where we ran a small set of queries comprising 1 to 4 triple patterns on increasing sizes of datasets, with and without the APPROX/RELAX operators. In all cases, the approxed/relaxed versions of the queries returned more answers than the exact query. Response times were good for most of the queries. One query had a slow response time due to one of the queries generated by the query rewriting algorithm at distance 1 containing two triple patterns, one of which matched a large number of triples most of which did not match with the other triple pattern.

For the rest of this section, we focus on our empirical trials over the YAGO dataset. YAGO contains over 120 million triples (4.83 GB in Turtle format) which we downloaded and stored in a TDB database (TDB is a Jena component used for RDF storage and querying). The size of the TDB database is 9.70 GB, and the nodes of the YAGO graph are stored in 1.1 GB file. The TDB database provides three indexes: on subject, predicate, object; on predicate, object, subject; and on object, subject, predicate.

We ran our experiments on an HP Pavillion with a 2Ghz i7 quad-core processor and 4 GB of RAM. The query processing algorithms were implemented in Java, and Jena was used for SPARQL query evaluation on the TDB database. We executed 10 queries over the database, comprising increasing numbers of triple patterns (1 up to 10):

```
Q1 = SELECT ?a WHERE
{ RELAX(?a rdf:type <location>)}

Q2 = SELECT ?n WHERE
{ ?a rdfs:label ?n .
  RELAX(?a <happenedIn> <Berlin>)
}

Q3 = SELECT ?n ?d WHERE
{ ?a rdfs:label ?n .
  RELAX(?a <happenedIn> <Berlin>) .
  ?a <happenedOnDate> ?d
}

Q4 = SELECT ?n ?m WHERE
{ ?a rdfs:label ?n .
  ?a <livesIn> ?b .
  ?a <actedIn> ?m .
  RELAX(?m <isLocatedIn> ?b)
}

Q5 = SELECT ?n1 ?n2 WHERE
{ ?a rdfs:label ?n1 .
  ?b rdfs:label ?n2 .
  RELAX(?a <isMarriedTo> ?b).
  APPROX(?a <livesIn>/<isLocatedIn>* ?p).
  APPROX(?b <livesIn>/<isLocatedIn>* ?p)
}

Q6 = SELECT ?n  WHERE
{ APPROX(?a <actedIn>/<isLocatedIn>
                        <Australia>) .
 ?a rdfs:label ?n .
 RELAX(?a rdf:type <actor>) .
 ?city <isLocatedIn> <China> .
 ?a <wasBornIn> ?city .
 APPROX(?a <directed>/<isLocatedIn>
```

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

```
                    United_States>)
 }

Q7 = SELECT ?n1 ?n2 WHERE
{ APPROX(?a rdf:type <event>) .
  RELAX(?a <happenedIn> ?b ).
  ?p <wasBornIn> ?b .
  ?p <wasBornOnDate> ?d .
  RELAX(?a <happenedOnDate> ?d) .
  ?a rdfs:label ?n1 .
  ?p rdfs:label ?n2
}

Q8 = SELECT ?c ?n ?p ?l ?d WHERE
{ ?a <hasFamilyName> ?n .
  ?a rdfs:label ?c .
  ?a <hasWonPrize> ?p .
  ?a <wasBornIn> ?l .
  RELAX(?a <wasBornOnDate> ?d) .
  APPROX(?a rdf:type <scientist>) .
  ?a <isMarriedTo> ?b1 .
  ?a <isMarriedTo> ?b2
}Filter (?b1!=?b2)

Q9 = SELECT ?c ?n ?p ?l ?d WHERE
{ ?a <hasFamilyName> ?n .
  ?a rdfs:label ?c .
  ?a <hasWonPrize> ?p .
  ?a <wasBornIn> ?l .
  ?a <wasBornOnDate> ?d .
  RELAX(?a rdf:type <scientist>) .
  ?a <isMarriedTo> ?b .
  ?b <wasBornOnDate> ?d .
  RELAX(?l <isLocatedIn>* <Germany>)
}

Q10 = SELECT ?n ?n1 ?n2 WHERE
{ ?a rdfs:label ?n .
  RELAX(?a rdf:type <actor> ).
  APPROX(?a <wasBornIn> ?city) .
  ?a  <actedIn> ?m1 .
  ?m1 <isLocatedIn>  <Australia> .
  ?a  <directed> ?m2 .
  ?m2 <isLocatedIn> <Australia>.
  APPROX(?city <isLocatedIn>
              <United_States>) .
  ?m1 rdfs:label ?n1 .
  ?m2 rdfs:label ?n2
}
```

For each query ran both the exact form of the query (without any APPROX or RELAX operators) and the version of the query as specified above. For the latter queries, we set the cost of applying all the edit operations of approximation to 1 and likewise the cost of applying all the RDFS entailment rules of relaxation to 1, and requested answers of maximum cost 2. We ran each query 6 times, ignored the first timing as a cache warm-up, and took the mean of the other 5 timings. The results are shown in Figures 4 and 5.

The reader will notice that we used the APPROX operator only on triple patterns containing a regular expression in which at least one constant appears. This is due to the fact that if we apply APPROX to simple triple patterns of the form $(?x, p, ?y)$, the rewriting algorithm will generate: $(?x, \_, ?y)$, which returns the whole YAGO database, and $(?x, \epsilon, ?y)$ which returns every node of the YAGO database.

Query Q_1 returns every location stored in YAGO. The query returns only 3 more answers in its relaxed form compared to its exact form. This is due to the fact that only one query is generated by the rewriting algorithm. Increasing the maximum cost does not result in the rewriting algorithm generating any more queries, and no other answers would be returned at higher cost.

Query Q_2 returns every event that happened in Berlin. When the second triple pattern is relaxed the rewriting algorithm generates a query that returns every event in YAGO. This explains the long execution time of the relaxed version of Q_2 compared to its exact form.

Query Q_3 returns every event that happened in Berlin along with its date. Query Q_4 returns every actor who acted in movies located in the same city as the actor lived in. Queries Q_3 and Q_4 return additional answers in their relaxed form compared to their exact form. Performance for Q_3 and Q_4 is reasonable.

Query Q_5 returns every married couple who live in the same city. The long execution time of both the exact and the approxed/relaxed form of this query is due to the presence of Kleene-closure in two of the query conjuncts.

Query Q_6 returns every Chinese actor who acted in American movies and directed Australian movies. Query Q_7 returns every event and person such that the person was born in the same place and on the same day that the event occurred.

Query Q_8 returns every scientist who has married twice and has won a prize. When the rewriting algorithm is applied to Q_6, Q_7 and Q_8 it generates many queries that return no answers or that return answers already computed. Such queries, in particular Q_8, could be optimised by using a cache that stores the answers to triple patterns executed in previous queries.

Query Q_9 returns every scientist who was born in Germany, won a prize, and was married to a woman with the same date of birth. This query does not return any answers even when relaxed. Increasing the maximum cost to 3, the rewriting algorithm generates 140 queries which also return no answer. Similarly to query Q_8, it would be possible to improve the execution time of this query by caching the answers of triple patterns executed in previous queries. In particular, knowing that a subset of the conjuncts of query Q_9 can return no answers, it would be possible to infer that the relaxed form of Q_9 cannot return any answer for any maximum cost.

Finally, query Q_{10} returns every actor that directed and acted in Australian movies and was born in the United States. The exact answers of this query take many hours to calculate due to the size of the YAGO dataset. Even though it is similar to query Q_6, Q_{10} returns not only the actor/director but also the movies which he acted in/directed. Similarly, the approxed/relaxed form of query Q_{10} takes even more time to be executed.

The overall results show that the evaluation of SPARQL^{AR} queries through a query rewriting approach is reasonable. The difference between the execution time of the exact form and the approxed/relaxed form of the queries is acceptable for queries with less than 7 conjuncts. The performance of some queries could be improved through appropriate optimisation techniques, such as path indexes and caching, which is an area of ongoing investigation. An advantage of adopting a query rewriting evaluation approach for SPARQL^{AR} is that existing optimisation techniques for SPARQL query evaluation can be reused.

6. Conclusions

In this paper we have presented query processing algorithms for an extended fragment of the SPARQL 1.1 language, incorporating approximation and relaxation operators. Our query processing approach is based on query rewriting whereby, given a query Q containing the APPROX and/or RELAX operators, we incrementally generate a set of queries $\{Q_0, Q_1, \ldots\}$ that do not contain these operators such that $\bigcup_i [[Q_i]]_{G,K} = [[Q]]_{G,K}$, and we return results according to their "distance" from the exact form of Q.

We have formally shown the soundness, completeness and termination of our query rewriting algorithm. Our empirical studies also show promising query processing performance.

An advantage of adopting a query rewriting approach is that existing techniques for SPARQL query optimisation and evaluation can be reused to evaluate the queries generated by our rewriting algorithm. Our ongoing work involves investigating optimisations to the rewriting algorithm itself, since it can generate a large number of queries. Specifically, we are studying query containment for SPARQL^{AR} and how query costs impact on this. Following this investigation, we plan to implement optimisations for the rewriting algorithm. 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 that $[[Q_1]]_{G,K} \subseteq [[Q_2]]_{G,K}$, in which case $[[Q]]_{G,K} = [[Q_1]]_{G,K}$.

Other ongoing work comprises a deeper empirical investigation of query evaluation performance and of possible optimisations. Another direction of research is extension of our approximation and relaxation operators, query evaluation and query optimisation techniques to flexible federated query processing for SPARQL 1.1.

References

[1] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *Web Semant.*, 7(2):57–73, Apr. 2009.

[2] J. M. Almendros-Jiménez, A. Luna, and G. Moreno. Fuzzy XPath queries in XQuery. In *On the Move to Meaningful Internet Systems: OTM 2014 Conference Proceedings - Confederated International Conferences: CoopIS, and ODBASE 2014, Amantea, Italy, October 27-31*, pages 457–472, 2014.

[3] C. Bizer, R. Cyganiak, and T. Heath. How to publish Linked Data on the Web. Web page, 2007. Revised 2008. Accessed 22/02/2010.

[4] G. Bordogna and G. Psaila. Customizable flexible querying in classical relational databases. In J. Galindo, editor, *Handbook of Research on Fuzzy*

	Q_1	Q_2	Q_3	Q_4	Q_5
Exact	6491/0.430	34/0.039	28/0.228	2350/0.675	109170/325.486
A/R	6494/0.461	25759/284.890	3029/0.530	2390/0.709	539170/3631.224

Fig. 4. Queries (number of answers/time in seconds).

	Q_6	Q_7	Q_8	Q_9	Q_{10}
Exact	8/8.75	1/473.589	819/208.919	0/0.157	N/A
A/R	16/40.484	1/803.455	1003/2795.254	0/35.818	N/A

Fig. 5. Queries (number of answers/time in seconds).

Information Processing in Databases, pages 191–217. IGI Global, 2008.

[5] A. Calì, R. Frosini, A. Poulovassilis, and P. T. Wood. Flexible querying for SPARQL. In *On the Move to Meaningful Internet Systems: OTM 2014 Conference Proceedings - Confederated International Conferences: CoopIS, and ODBASE 2014, Amantea, Italy, October 27-31*, pages 473–490, 2014.

[6] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaida. 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.

[7] R. De Virgilio, A. Maccioni, and R. Torlone. A similarity measure for approximate querying over RDF data. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 205–213, New York, NY, USA, 2013. ACM.

[8] S. Elbassuoni, M. Ramanath, and G. Weikum. Query relaxation for entity-relationship search. In *Proceedings of the 8th Extended Semantic Web Conference on The Semanic Web: Research and Applications - Volume Part II*, ESWC'11, pages 62–76, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] R. Fink and D. Olteanu. On the optimal approximation of queries using tractable propositional languages. In *Proceedings of the 14th International Conference on Database Theory*, ICDT '11, pages 174–185, New York, NY, USA, 2011. ACM.

[10] A. Hogan, M. Mellotte, G. Powell, and D. Stampouli. Towards fuzzy query-relaxation for RDF. In E. Simperl, P. Cimiano, A. Polleres, O. Corcho, and V. Presutti, editors, *The Semantic Web: Research and Applications*, volume 7295 of *Lecture Notes in Computer Science*, pages 687–702. Springer Berlin Heidelberg, 2012.

[11] H. Huang and C. Liu. Query relaxation for star queries on RDF. In *Proceedings of the 11th International Conference on Web Information Systems Engineering*, WISE'10, pages 376–389, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] H. Huang, C. Liu, and X. Zhou. Computing relaxed answers on RDF databases. In *Proceedings of the 9th International Conference on Web Information Systems Engineering*, WISE '08, pages 163–175, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] C. Kiefer, A. Bernstein, and M. Stocker. The fundamentals of iSPARQL: a virtual triple approach for similarity-based semantic web tasks. In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference*, ISWC'07/ASWC'07, pages 295–309, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference*, ISWC'06, pages 30–43, Berlin, Heidelberg, 2006. Springer-Verlag.

[15] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, Sept. 2009.

[16] A. Poulovassilis and P. T. Wood. Combining approximation and relaxation in semantic web path queries. In *Proceedings of the 9th International Semantic Web Conference*, ISWC'10, pages 631–646, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] B. R. K. Reddy and P. S. Kumar. Efficient approximate SPARQL querying of web of linked data. In F. Bobillo, R. N. Carvalho, P. C. G. da Costa, C. d'Amato, N. Fanizzi, K. B. Laskey, K. J. Laskey, T. Lukasiewicz, T. Martin, M. Nickles, and M. Pool, editors, *URSW*, volume 654 of *CEUR Workshop Proceedings*, pages 37–48. CEUR-WS.org, 2010.

[18] M. Sassi, O. Tlili, and H. Ounelli. Transactions on Large-Scale Data- and Knowledge-Centered Systems V. chapter Approximate Query Processing for Database Flexible Querying with Aggregates, pages 1–27. Springer-Verlag, Berlin, Heidelberg, 2012.

[19] M. Schmidt. *Foundations of SPARQL Query Optimization*. PhD thesis, Albert-Ludwigs-Universitat Freiburg, 2009.