Undefined 0 (2015) 1–0
IOS Press

Flexible Query Processing for SPARQL

Riccardo Frosini [a] Andrea Calì [a,b] Alexandra Poulovassilis [a] Peter T. Wood [a]
[a] *London Knowledge Lab, Birkbeck, University of London, UK*
Email: {riccardo,andrea,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 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 a fragment of SPARQL 1.1 incorporating regular path queries (property path queries), extended with 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 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 desirable.

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, it does not support notions of query approximation and relaxation (apart from the OPTIONAL operator).

Example 1. *Suppose a user wishes to find events that took place in London on 15th September 1940 and poses the following query on the YAGO knowledge base*[1]*, which is derived from multiple sources such as Wikipedia, WordNet and GeoNames:*

$$(x, on, \text{“15/09/1940”})\ AND\ (x, in, \text{“}London\text{”})$$

(The above is not a complete SPARQL query, but is sufficient to illustrate the problem we address.) This query returns no results because there are no property edges named "on" or "in" in YAGO.

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

$$(x, happenedOnDate, \text{“15/09/1940”})\ AND\ (x, happenedIn, \text{“}London\text{”})$$

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 15th September 1940, including those occurring in London. In this particular instance only one answer is returned which is the event "Battle of Britain", but other events*

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

could in principle have been returned. So the query exhibits better recall than the original query, but possibly low precision.

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:

$$(x, happenedOnDate, \text{“15/09/1940”})\ AND\ (x, happenedIn/label, \text{“}London\text{”})$$

This query now returns the only event that occurred on 15th September 1940 in London, that is "Battle of Britain". It exhibits both better recall than the original query and also high precision.

Example 2. *Suppose the user wishes to find the geographic coordinates of the "Battle of Waterloo" event by posing the query*

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

in which angle brackets delimit a URI. 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 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

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

This query returns 16 answers that may be relevant to the user, since YAGO does store the geographic coordinates of some (unspecified) locations in Belgium, increasing recall but with possibly low precision.

Moreover, YAGO does in fact store directly the coordinates of the "Battle of Waterloo" event, so if we apply 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, showing both high precision and high recall

In this paper we describe an extension of a fragment of SPARQL 1.1 with *query approximation* and *query relaxation* operations that automatically generate rewritten queries 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 review and extend these results to a larger SPARQL language fragment. We also explore in more detail the theoretical and performance aspects of 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 theoretical foundation of our approach, 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. We include a discussion in Section 4.4 on how users may be helped in formulating queries and interpreting results in a system which includes query approximation and relaxation. Section 5 presents and discusses the results of a performance study over the YAGO dataset. Finally, Section 6 gives our concluding remarks and directions for 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 [14] 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 [12,11,20] 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 proposed to extend the XPath query language with the aim of providing mechanisms to assign priorities to queries and to rank query answers [2]. These techniques are based on fuzzy extensions of the Boolean operators.

Flexible querying approaches for SQL have been discussed in [21] 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.

In [18] the authors show 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.

Finally, in [23] the authors describe and provide technical details of the implementation of a flexible querying evaluator for conjunctive regular path queries, extending the work in [18].

In contrast to all the above work, our focus is on the SPARQL 1.1 language. In [5] we extended, for the first time, a fragment of this language with query approximation and query relaxation operators, terming the extended language SPARQL^{AR}. Here, we add the UNION operator to SPARQL^{AR} and derive additional complexity results. Moreover, 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 we present the results of a performance study on the YAGO dataset.

3. Theoretical Foundation

In this section we give definitions of the syntax and semantics of SPARQL^{AR} summarising and extending the syntax and semantics from [5], and also the complexity results from that paper. We begin with some necessary definitions.

Definition 1 (Sets, triples and variables)**.** *We 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 U and L. 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 [16] 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 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 *is a non-negative number.*

Note that, in the above definition, we modify the definition of an RDF-Graph from [16] to add weights to the edges, which are needed to formalise our flexible querying semantics. Initially, these weights are all 0.

We next define the ontology of an RDF dataset, using a fragment of the RDF-Schema (RDFS) vocabulary.

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

In an RDF-graph $G = (N, D, E)$, we assume that each node in N represents an instance or a class and each edge in E a property (even though, more generally, RDF does not distinguish between instances, classes and properties; in fact, in RDF it is possible to use a property as a node of the graph). 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 [16] 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 $P_1, P_2 \in RegEx(U)$ *are also regular expression patterns,* 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:*

$$\begin{gathered} Q := UV \times UV \times UVL \mid UV \times RegEx(U) \times UVL \mid \\ Q_1 \ AND \ Q_2 \mid Q_1 \ UNION \ Q_2 \mid Q \ FILTER \ R \mid \\ RELAX(UV \times RegEx(U) \times UVL) \mid \\ APPROX(UV \times RegEx(U) \times UVL) \end{gathered}$$

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

(In the W3C SPARQL syntax, a dot (.) is used for conjunction but, for greater clarity, 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 Q', $head(Q')$, 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 applying the approximation and relaxation operators. These costs are used to rank the answers. In particular, when we introduce the APPROX and RELAX operators below these costs determine the ranking of answers returned to the user, with exact answers (of cost 0) being returned first, followed by answers with increasing costs.

We extend the notion of SPARQL query evaluation from returning a set of mappings to returning a set of pairs of the form $\langle \mu, c \rangle$, where μ is a mapping and c is a non-negative integer that indicates the cost of the answers arising from this mapping.

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 such that $var(\mu) = var(\mu_1) \cup var(\mu_2)$ 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:

$M_1 \cup M_2 = \{\langle \mu, c \rangle \mid \langle \mu, c_1 \rangle \in M_1$ or $\langle \mu, c_2 \rangle \in M_2$ with $c = c_1$ if $\nexists c_2 . \langle \mu, c_2 \rangle \in M_2$, $c = c_2$ if $\nexists c_1 . \langle \mu, c_1 \rangle \in M_1$, and $c = min(c_1, c_2)$ otherwise$\}$.
$M_1 \bowtie M_2 = \{\langle \mu_1 \cup \mu_2, c_1 + c_2 \rangle \mid \langle \mu_1, c_1 \rangle \in M_1$ and $\langle \mu_2, c_2 \rangle \in M_2$ 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:

$$[[\langle x, \epsilon, y \rangle]]_G = \{\langle \mu, 0 \rangle \mid var(\mu) = var(\langle x, \epsilon, y \rangle) \wedge \exists c \in N \,.\, \mu(x) = \mu(y) = c\}$$

$$[[\langle x, z, y \rangle]]_G = \{\langle \mu, c \rangle \mid var(\mu) = var(\langle x, z, y \rangle) \wedge \langle \mu(\langle x, z, y \rangle), c \rangle \in E\}$$

$$[[\langle x, P_1 | P_2, y \rangle]]_G = [[\langle x, P_1, y \rangle]]_G \cup [[\langle x, P_2, y \rangle]]_G$$

$$[[\langle x, P_1 / P_2, y \rangle]]_G = [[\langle x, P_1, z \rangle]]_G \bowtie [[\langle z, P_2, y \rangle]]_G$$

$$[[\langle x, P^*, y \rangle]]_G = [[\langle x, \epsilon, y \rangle]]_G \cup [[\langle x, P, y \rangle]]_G \cup \bigcup_{n \geq 1} \{\langle \mu, c \rangle \mid \langle \mu, c \rangle \in [[\langle x, P, z_1 \rangle]]_G \bowtie [[\langle z_1, P, z_2 \rangle]]_G \bowtie \cdots \bowtie [[\langle z_n, P, y \rangle]]_G\}$$

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 $x = a$: $\mu \models R$ if $x \in var(\mu)$, $a \in LU$ and $\mu(x) = a$;
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 $\overrightarrow{w}$:

$$[[Q_1 \text{ AND } Q_2]]_G = [[Q_1]]_G \bowtie [[Q_2]]_G$$

$$[[Q_1 \text{ UNION } Q_2]]_G = [[Q_1]]_G \cup [[Q_2]]_G$$

$$[[Q \text{ FILTER } R]]_G = \{\langle \mu, c \rangle \in [[Q]]_G \mid \mu \models R\}$$

$$[[\text{SELECT}_{\overrightarrow{w}} Q]]_G = \pi_{\overrightarrow{w}}([[Q]]_G)$$

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 [18] and relies on a fragment of the RDFS entailment rules known as ρDF [15]. 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. Notice that if K_1 is finite then also $cl(K_1)$ is finite.

Applying a rule means adding a triple that is deducible by the rule to G or K. Specifically, if there are two triples t, t' that match the antecedent of a rule, then it is possible to insert the triple implied by the consequent of the rule. For example, the triple pattern $\langle x, startsExistingOnDate, y\rangle$ can be deduced from $\langle x, wasBornOnDate, y\rangle$ and $\langle wasBornOnDate, sp, startsExistingOnDate\rangle$ by applying rule 2.

In order to apply relaxation to queries, the *extended reduction* of an ontology K is required [13]. 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 more rules can be applied; (iii) apply rules 1 and 3 of Figure 1 in reverse until no more rules can be applied. Applying a rule in reverse means removing a triple deducible by the rule from G or K. Specifically, if there are two triples t and t' that match the antecedent of a rule then it is possible to remove a triple that can be derived from t and t' by that rule.

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 associating an unambiguous cost to queries, so that query answers can then be returned to users incrementally in order of increasing cost.

If we did not use the extended reduction of the ontology K, then the relaxation steps applied would not necessarily be the "smallest". For example, consider the following ontology $K = \{(b, dom, c), (a, sp, b), (a, dom, c)\}$, where $K \neq extRed(K)$. If we relax the triple pattern (x, a, y) with respect to K, then as a first step we could apply rule 5 to generate $(x, type, c)$. However, the same triple pattern can be generated with 2 steps of relaxation by applying rule 1 first and then rule 5 of Figure 1.

As a further condition, we require that the ontology K is acyclic in order for relaxed queries to have unambiguous costs (a detailed analysis can be found in [13]).

Example 3. *Given the following cyclic ontology* $K = (\langle a, sp, b\rangle, \langle b, sp, a\rangle, \langle a, dom, c\rangle, \langle b, dom, c\rangle)$ *then* $cl(K) = K \cup (\langle a, sp, a\rangle, \langle b, sp, b\rangle)$. *By applying steps (ii) and (iii) above we could generate two possible ontologies* $K' = (\langle a, sp, b\rangle, \langle b, sp, a\rangle, \langle b, dom, c\rangle)$ *and* $K'' = (\langle a, sp, b\rangle, \langle b, sp, a\rangle, \langle a, dom, c\rangle)$ *that are extended reductions of* K.

Consider now the query $Q = RELAX(x, a, y)$ *which can be relaxed to* (x, b, y) *with* K'. *Applying a second step of relaxation we obtain* $(x, type, c)$. *If instead we used ontology* K'', *a first step of relaxation would immediately generate* $(x, type, c)$. *Therefore, having non-acyclic ontologies might generate the same triple pattern at two different relaxation distances from the original triple pattern, depending on the reduced ontology.*

Following the terminology of [13], a triple pattern $\langle x, p, y\rangle$ *directly relaxes* to a triple pattern $\langle x', p', y'\rangle$ with respect to an ontology $K = extRed(K)$, denoted $\langle x, p, y\rangle \prec_i \langle x', p', y'\rangle$, 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.

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.

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

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

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

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

$$\bowtie \cdots \bowtie [[\text{RELAX}(z_n, P, y)]]_{G,K}\}$$

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

$$\{hasFamilyName, hasGivenName, label, actedIn, Actor, English\_politicians, politician\},$$

and E_K is

$$\{(hasFamilyName, sp, label), (hasGivenName, sp, label), (actedIn, domain, actor), (English\_politicians, sc, politician)\}$$

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

```
SELECT * WHERE {
?x actedIn <Tea_with_Mussolini> .
?x hasFamilyName ?z }
```

The above query returns 4 answers. However, some actors have only a single name (for example Cher), or have their full name recorded using the "label" property directly. By applying relaxation to the second triple pattern using rule (2), we can replace the predicate $hasFamilyName$ by "label". This causes the relaxed query to return also the given names of actors in that film recorded through the property "hasGivenName" (hence returning Cher), as well as actors' full names recorded through the property "label": a total of 255 results.

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), it is possible to replace the class $English\_politicians$ by $politicians$. This relaxed query will return every politician who was born in England, giving possibly additional answers of relevance to the user.

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):

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

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. The application of an edit operation op has a non-negative 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:

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

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.

Example 5. *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 substitute the predicate "label" by "_". The query will then return more information concerning that discovery, such as its preferred name ($hasPreferredName$) and the Wikipedia abstract ($hasWikipediaAbstract$), improving recall and maintaining good precision.

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 predicate "$isPoliticianOf$" only connects persons to states of the United States in YAGO. If the first triple pattern is approximated by substituting the predicate "$isPoliticianOf$" with "_", then the query will return the expected results, matching the correct predicate to retrieve the desired answers, which is "$holdsPoliticalPosition$". It will also retrieve all the other persons that are born in Germany (thus showing improved recall, but lower precision).

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

3.3. Complexity of query answering

We now study the combined, data and query complexity of SPARQL^{AR}, extending the complexity results from [16,17,22] for simple SPARQL queries, from [1] for SPARQL with regular expression patterns to include our new flexible query constructs, and from [5] to include now UNION in SPARQL^{AR}.

The complexity of query evaluation is based on the following decision problem, which we denote EVALUATION: Given as input a graph $G = (N, D, E)$, an ontology K, a query Q and a pair $\langle \mu, cost\rangle$, is it the case that $\langle \mu, cost\rangle \in [[Q]]_{G,K}$? Considering data complexity, the decision problem 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? Finally, the decision problem for query complexity is the following: Given as input an ontology K, a query Q and a pair $\langle \mu, cost\rangle$, is it the case that $\langle \mu, cost\rangle \in [[Q]]_{G,K}$, with G a fixed graph?

We have the following results, the proofs of which are given in the Appendix.

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.*

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.*

The complexity study of SPARQL^{AR} in [5] is summarised in the first six lines of Table 1, where the combined, data and query complexity are shown for specific language fragments and combinations of operators.

The results for query complexity follow from Lemma 1 and Theorems 1, 2 and 3.

We next show three new complexity results which extend those of [5], summarised in the last two lines of Table 1:

Theorem 6. *EVALUATION is in NP for queries containing AND, UNION, FILTER, APPROX, RELAX and regular expression patterns.*

Theorem 7. *EVALUATION is NP-complete for queries that may contain regular expression patterns and that are constructed using the operators AND, UNION, FILTER, RELAX, APPROX and SELECT.*

Theorem 8. *EVALUATION is PTIME in data complexity for queries that may contain SELECT and regular expression patterns, and that are constructed using the operators AND, UNION, FILTER, RELAX and APPROX.*

The results for query complexity follow from Lemma 1 and Theorems 6 and 7.

We conclude our complexity study confirming that adding the UNION operator to SPARQL^{AR} does not increase the overall complexity. EVALUATION is NP-complete for queries that contain regular expression patterns and that are constructed using the operators AND, UNION, FILTER, RELAX, APPROX and SELECT.

3.4. OPTIONAL operator

The OPTIONAL operator can be added to a SPARQL query in order to retrieve information only when it is available. In other words, it allows optional matching of query patterns. If in query Q the OPTIONAL operator is applied to query pattern Q'', that is $Q = Q'$ OPTIONAL $\{Q''\}$, then $[[Q]]_G$ will return all the mappings in $[[Q']]_G \bowtie [[Q'']]$ plus all the mappings in $[[Q']]$ that are not compatible with any mappings in $[[Q'']]$.

It is possible to add the OPTIONAL operator to SPARQL^{AR}, allowing APPROX and RELAX to be applied to triple patterns occurring within an OPTIONAL clause, with the same semantics as specified earlier. However, the complexity of SPARQL with the OPTIONAL clause is PSPACE-complete [16]. Therefore, by our earlier results, the complexity of SPARQL^{AR} would also increase similarly.

4. Query Processing

We evaluate SPARQL^{AR} queries by making use of a *query rewriting algorithm*, following a similar approach to [11,12,20]. 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 $\bigcup_i [[Q_i]]_{G,K} = [[Q]]_{G,K}$.

We present the algorithm in Section 4.1, proving its correctness in Section 4.2 and termination in Section 4.3. Some practical issues relating to how users might benefit from a flexible querying system such as this are discussed in Section 4.4.

4.1. Query Rewriting

Our query rewriting algorithm (Algorithm 2 below) starts by considering the query Q_0 which re-

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

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

turns 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.

The function *toCQS* ("to conjunctive query set") takes as input a query Q, and returns a set of pairs $\langle Q_i, 0\rangle$ such that $\bigcup_i [[Q_i]]_G = [[Q]]_G$ and no Q_i contains the UNION operator. The function *toCQS* exploits the following equality:

$$
\begin{aligned}
&[[(Q_1 \text{ UNION } Q_2) \text{ AND } Q_3]]_G = \\
&([[Q_1]]_G \cup [[Q_2]]_G) \bowtie [[Q_3]]_G = \\
&([[Q_1]]_G \bowtie [[Q_3]]_G) \cup ([[Q_2]]_G \bowtie [[Q_3]]_G) = \\
&([[Q_1 \text{ AND } Q_3]]_G) \cup ([[Q_2 \text{ AND } Q_3]]_G)
\end{aligned}
$$

We assign to the variable *oldGeneration* the set of queries returned by $toCQS(Q_0)$. For each query Q' in the set *oldGeneration*, each triple pattern $\langle x_i, P_i, y_i\rangle$ in Q' 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. The applyApprox and applyRelax functions invoked by Algorithm 2 are shown as Algorithms 3 and 5, respectively. From each query constructed in this way, we next generate a new set of queries by 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.

In Algorithm 2, the *addTo* operator accepts two arguments: the first is a collection C of query/cost pairs, while the second is a single query/cost pair $\langle Q, c\rangle$. The operator adds $\langle Q, c\rangle$ to C. If C already contains a pair $\langle Q, c'\rangle$ such that $c' \geq c$, then $\langle Q, c'\rangle$ is replaced by $\langle Q, c\rangle$ in C.

To compute the query answers (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.

The applyApprox and applyRelax functions, respectively, invoke the functions approxRegex and replaceTriplePattern, shown as Algorithms 4 and 6. In Algorithm 6, z, z_1 and z_2 are fresh new variables. The relaxTriplePattern function

Algorithm 1: Flexible Query Evaluation

input : Query Q; approx/relax max cost c; Graph G; Ontology K.
output: List M of mapping/cost pairs, 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 query/cost pairs, sorted by cost.
Q_0 := remove APPROX and RELAX operators, and label triple patterns of Q_{AR};
queries:=$toCQS(Q_0)$;
oldGeneration := $toCQS(Q_0)$;
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 $cost + cost' \leq c$ **then**
          newGeneration := addTo(newGeneration,$\langle Q', cost + cost'\rangle$) ;
          queries:=addTo(queries, $\langle Q', cost + cost'\rangle$) ; `/* 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 S of query/cost pairs.
$S := \emptyset$;
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;

Algorithm 4: approxRegex

input : Regular Expression P.
output: Set T of RegEx/cost pairs.
$T := \emptyset$;
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 S of query/cost pairs.
$S := \emptyset$;
foreach $\langle \langle x', P', y' \rangle_R, cost \rangle \in relaxTriplePattern(\langle x, P, y \rangle, K)$ **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;

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*. The predicate $type^-$ is generated when we apply rule 6 of Figure 1 to a triple pattern. Given a triple pattern $\langle x, a, y \rangle$ where x is a constant and is y a variable, and an ontology statement $\langle a, range, d \rangle$, we can deduce the triple pattern $\langle y, type, d \rangle$. If instead the predicate a appears in a triple pattern containing a regular expression such as $\langle x, a/b, z \rangle$ (which is equivalent to $\langle x, a, y \rangle$ AND $\langle y, b, z \rangle$), then we cannot simply replace it with $\langle y, type, d \rangle$ as the regular expression would be broken apart and two triple patterns would result. By using $\langle d, type^-, y \rangle$, we correctly construct the triple pattern $\langle d, type^-/b, z \rangle$.

In the following example, we illustrate how the rewriting algorithm works by showing the queries it generates, starting from a SPARQL^{AR} query.

Example 6. *Consider the following ontology K (satisfying $K = extRed(K)$), which is a fragment*

Algorithm 6: relaxTriplePattern

input : Triple pattern $\langle x, P, y\rangle$; Ontology K.
output: Set T of triple pattern/cost pairs.
$T := \emptyset$;
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;

of the YAGO knowledge base:

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

Suppose a user wishes to find every event which took place in London on 15th September 1940 and poses the following query Q:

$$APPROX(x, happenedOnDate, \text{“15/09/1940”})\ AND\ RELAX(x, happenedIn, \text{“London”}).$$

As pointed out in Example 1, 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:

$$Q_1 = (x, \epsilon, \text{“15/09/1940”})_A \ AND \ (x, happenedIn, \text{“}London\text{”})_R$$
$$Q_2 = (x, happenedOnDate/\_, \text{“15/09/1940”})_A \ AND \ (x, happenedIn, \text{“}London\text{”})_R$$
$$Q_3 = (x, \_/happenedOnDate, \text{“15/09/1940”})_A \ AND \ (x, happenedIn, \text{“}London\text{”})_R$$
$$Q_4 = (x, \_, \text{“12/12/12”})_A \ AND \ (x, happenedIn, \text{“}London\text{”})_R$$
$$Q_5 = (x, happenedOnDate, \text{“15/09/1940”})_A \ AND \ (x, placedIn, \text{“}London\text{”})_R$$
$$Q_6 = (x, happenedOnDate, \text{“15/09/1940”})_A \ AND \ (x, type, Event)_R$$

Each of these also returns empty results, with the exception of query Q_6 which returns every event occurring on 15/09/1940 (YAGO contains only one such event, namely “Battle of Britain”).

4.2. 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 and corollary.

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:*

$$M_1 \cup M_2 \subseteq M_1' \cup M_2' \tag{1}$$
$$M_1 \bowtie M_2 \subseteq M_1' \bowtie M_2' \tag{2}$$

The following result follows from Lemma 2:

Corollary 1. *Given four sets of evaluation results M_1, M_2, M_1' and M_2' such that $M_1 = M_1'$ and $M_2 = M_2'$, it holds that:*

$$M_1 \cup M_2 = M_1' \cup M_2' \tag{3}$$
$$M_1 \bowtie M_2 = M_1' \bowtie M_2' \tag{4}$$

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

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

Theorem 9. *The Rewriting Algorithm is sound and complete.*

4.3. Termination of the Rewriting Algorithm

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

Theorem 10. *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 $c' > 0$.*

4.4. Practical Considerations

In the previous sections we have concentrated on theoretical aspects of SPARQL^{AR}. In practice SPARQL^{AR} would be used as part of a framework allowing users to search RDF data in a flexible way. A front-end could allow users to pose queries using keywords or natural language. Such user queries could then be translated into SPARQL (cf. [19]).

Suppose a user poses the following question to the system: "What event happened in London on 15/09/1940?". This question could be translated to the query $((x, happenedOnDate, \text{“15/09/1940”})$ AND $(x, happenedIn, \text{“}London\text{”}))$ from Example 6. The system could automatically apply APPROX to the first triple pattern and RELAX to the second triple pattern (as shown in the example) by determining that $happenedIn$ appears in the given ontology, whereas $happenedOnDate$ does not. As we saw earlier, applying approximation and relaxation to this query produces the answer that the user is looking for.

Applying the APPROX operator without an upper bound on cost will eventually return every connected node of the RDF graph. This negatively affects the precision of answers but ensures 100% recall. When applying the substitution operation of APPROX to a triple pattern, we do not specify the predicate that needs to be replaced, but instead insert the wildcard (_) that allows the insertion/replacement of any predicate. Of course, this is a drawback in terms of the precision of answers retrieved. However, for each predicate in a query, it is possible to specify a set of predicates that are semantically similar to it in order to increase the precision of the retrieved answers. A similarity matching algorithm, based either on syntactic or semantic similarity, could be used to compare the predicates of the RDF dataset. The semantic similarity could exploit dictionaries such as WordNet[2]. Moreover, we could assign different costs for substitution by different (sets of) predicates, depending on how similar they are to the original predicate. This would allow for a finer ranking of the answers.

Similarly, it is possible to add a finer ranking of the answers arising from the RELAX operator. When we relax a triple pattern to derive a direct relaxation, we make use of a triple from the ontology K. Instead of assigning a cost to each rule of Figure 1, we could assign a cost to each triple in K, reflecting domain experts' views of the semantic closeness of concepts. Therefore, the direct relaxation would have a cost depending on which triple in K is used.

Finally, in order to help users interpret answers to their queries, the system could provide information about which rewritten query returned which answers. Showing only queries to users might not be particularly helpful, especially if the original query was simply in the form of keywords. Instead, showing the sequence of steps by which the original terms used by the user were approximated or relaxed could help them decide whether the answers returned were meaningful or not.

5. Experimental Results

We have implemented the query evaluation algorithms described above in Java, using Jena for SPARQL query evaluation. Figure 3 illustrates the system architecture, which consists of three layers: the GUI layer, the System layer, and the Data layer. The GUI layer supports user interaction with the system, allowing queries to be submitted, costs of the edit and relaxation operators to be set, data sets and ontologies to be selected, and query answers to be incrementally displayed to the user. The System layer comprises three components: the Utilities, containing classes providing the core logic of the system; the Domain Classes, providing classes relating to the construction of SPARQL^{AR} queries; and the Query Evaluator in which query rewriting, optimisation and evaluation are undertaken. The Data layer connects the system to the selected RDF dataset and ontology using the JENA API; RDF datasets are stored as a TDB database[3] and RDF schemas can be stored in multiple RDF formats (e.g. Turtle, N-Triple, RDF/XML).

User queries are submitted to the GUI, which invokes a method of the *SPARQL^{AR} Parser* to parse the query string and construct an object of the class *SPARQL^{AR} Query*. The GUI also invokes

[2] `https://wordnet.princeton.edu/`

[3] `https://jena.apache.org/documentation/tdb/.`

the *Data/Ontology Loader* which creates an object of the class *Data/Ontology Wrapper*, and the *Approx/Relax Constructor* which creates objects of the classes *Approx* and *Relax*. Once these objects have been initialised, they are passed to the Query Evaluator by invoking the *Rewriting Algorithm*. This generates the set of SPARQL queries to be executed over the RDF dataset. The set of queries is passed to the *Evaluator*, which interacts with the *Optimiser* and the *Cache* to improve query performance — we discuss the Optimiser and the Cache in Section 5.2. The *Evaluator* uses the *Jena Wrapper* to invoke Jena library methods for executing SPARQL queries over the RDF dataset. The *Jena Wrapper* also gathers the query answers and passes them to the *Answer Wrapper*. Finally, the answers are displayed by the *Answers Window*, in ranked order.

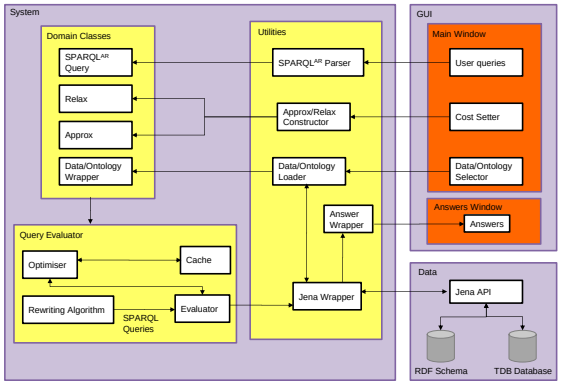

Fig. 3. SPARQL^{AR} system architecture

We have conducted empirical trials over the YAGO dataset and the Lehigh University Benchmark (LUBM)[4]. 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.

For the rest of this section, we focus on our empirical trials over the YAGO dataset, firstly without any optimisations, and then in Section 5.2 with an optimised query evaluator. YAGO contains over 120 million triples (4.83 GB in Turtle format) which we downloaded and stored in a TDB database. The size of the TDB database is 9.70 GB, and the nodes of the YAGO graph are stored in a 1.1 GB file.

We ran our experiments on a Windows PC with a 2.4Ghz i5 dual-core processor and 8 GB of RAM. We executed 10 queries over the database, comprising increasing numbers of triple patterns (1 up to 10), listed below. The aim of this performance study was to further gauge the practical feasibility of our techniques and to discover major performance bottlenecks requiring further investigation. A more comprehensive and detailed performance study is planned for future work.

```
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>
                     United_States>) }
```

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

```
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 }
```

The reader will notice that we used the APPROX operator only on triple patterns containing a regular expression in which either the subject or object is a constant. 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 the following two triple patterns: $(?x, \_, ?y)$ which returns every triple in the database, and $(?x, \epsilon, ?y)$ which returns every node in the database.

For each query Q_1 to Q_{10}, we 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 each edit operation of approximation and each RDFS entailment rule of Figure 1 to one, and requested answers of maximum cost two. We ran each query 6 times, ignored the first timing as a Jena cache warm-up, and took the mean of the other 5 timings. We restart our system each time we run a query; this avoids the possibility that the warm-up caching of a previous query enhances the execution performance of other queries.

The numbers of answers returned by each query, for both the exact form and the APPROX/RELAX (A/R) form, are shown in Tables 2 and 3, along with the number of rewritten queries in each case (# of queries). Tables 4 and 5 list the execution times for the exact queries, the A/R queries and the A/R queries with a simple caching optimisation implemented (optimised A/R). We discuss the results without this optimisation in the next subsection, and the results with this optimisation applied in Section 5.2.

5.1. Initial results

Query Q_1 returns every location stored in YAGO. The rewriting algorithm generates only the following additional query

```
SELECT ?a WHERE
{?a rdf:type <Resource>}
```

which returns only 3 answers. 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, while query Q_4 returns every actor who acted in movies located in the same city where the actor lived. Queries Q_3 and

Q_4 return additional answers in their relaxed form compared to their exact form. Both queries exhibit reasonable performance.

Query Q_5 returns all married couples who live in the same city. The long execution time of the exact form of this query is due to the presence of Kleene-closure in two of the query conjuncts. Moreover, the rewriting algorithm generates 95 queries which are time-consuming to evaluate due to the presence of not only of the Kleene-closure but also the wild-card symbol "_". We were not able to complete the execution of query Q_5 with approximation and relaxation. It might be possible to overcome this problem by replacing "_" with a selected disjunction of predicates. Such predicates would be chosen using knowledge of the graph structure. For example, when we approximate the triple pattern $\langle ?a, livesIn/isLocatedIn*, ?p\rangle$ in query Q_5, we generate $\langle ?a, livesIn/\_/isLocatedIn*, ?p\rangle$ using the insertion edit operator and, during query evaluation, the symbol _ is replaced with a disjunction of all the predicates in YAGO. We could instead replace _ with a disjunction of the predicates that are known to connect $livesIn$ and $isLocatedIn$, i.e. the predicates p such that there is a path $livesIn/p/isLocatedIn$ in YAGO. This type of optimisation is currently being investigated.

Query Q_6 returns every Chinese actor who played in American films and directed Australian films. The A/R version of the query takes many hours to evaluate (the rewriting algorithm generates 154 queries) due to the _ symbol we use for the insertion and substitution approximation operations. Applying the optimisation technique described in Section 5.2 below decreases the execution time dramatically and returns results in a more reasonable time. Similarly to query Q_5, it would also be possible to replace the _ symbol with a selected disjunction of predicates, making the query more likely to return answers more quickly still.

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. When the rewriting algorithm is applied to Q_7, it generates many queries that contain no answers or that contain answers already computed. The first version of caching that we have implemented (described in Section 5.2) is not sophisticated enough to help with the A/R version of Q_7, for the reason explained in Section 5.2 and further work is required here.

Query Q_8 returns every scientist who has married twice and has won a prize. The rewriting algorithm generates 17 queries from Q_8. The long execution time of the A/R form of Q_8 is due to use of the "_" symbol. The running time of the query is improved significantly by the optimisation described in Section 5.2.

Query Q_9 returns every scientist who was born in Germany, has won a prize, and was married to someone with the same date of birth. This query returns no answers. The execution of the A/R form of the query takes many hours, due to the Kleene-closure. However, the running time of the query is again dramatically improved by the optimisation described in Section 5.2.

Finally, query Q_{10} returns every actor who directed and acted in Australian movies and was born in the United States. The exact form of this query returns no answers. The rewriting algorithm generates 47 queries and the A/R form of the query takes a very long time to evaluate. Once again, the optimisation described in Section 5.2 gives a significant reduction in the running time.

5.2. Optimised evaluation

In Tables 4 and 5 we also show the query execution times for the A/R forms of all the queries using an optimised query evaluator. The optimisation is based on a caching technique, in which we pre-compute some of the answers in advance. The Optimiser module stores the cached answers in memory using the Java class HashSet[5] which enables answers to be retrieved efficiently. Algorithm 7 shows the optimised evaluation. We leave it as future work to investigate other join strategies, such as sort-merge join or sideways information passing.

In Algorithm 7, we start by splitting a query into two parts: the triple patterns which do not have APPROX or RELAX applied to them (which we call the *exact* part) and those which have (which we call the A/R part). We first evaluate the exact part of the query and store the results. We then apply the rewriting algorithm to the A/R part. Each triple pattern of the latter is evaluated in-

[5]Java documentation: `https://docs.oracle.com/javase/6/docs/api/java/util/HashSet.html`

Table 2
Numbers of answers (Exact and A/R) and numbers of rewritten queries (A/R).

	Q_1	Q_2	Q_3	Q_4	Q_5
Exact	6491	116	106	8546	585150
A/R	6494	60614	6867	8586	N/A
# of queries	2	5	5	2	95

Table 3
Numbers of answers (Exact and A/R) and numbers of rewritten queries (A/R).

	Q_6	Q_7	Q_8	Q_9	Q_{10}
Exact	28	5	1540	0	0
A/R	14431	N/A	22540	0	0
# of queries	154	36	17	29	47

Table 4
Query execution time (in seconds).

	Q_1	Q_2	Q_3	Q_4	Q_5
Exact	0.321	0.008	0.009	1.512	7670
A/R	0.340	66.32	0.81	1.571	N/A
optimised A/R	0.440	60.4	2.31	1.01	N/A

Table 5
Query execution time (in seconds).

	Q_6	Q_7	Q_8	Q_9	Q_{10}
Exact	0.123	5	0.173	1.23	323.100
A/R	N/A	N/A	272.875	N/A	N/A
optimised A/R	60.23	N/A	12.475	0.08	100.4

dividually; all possible pairs of triple patterns are also evaluated. The answers to each are stored in *cache*, a data structure that contains these partial evaluation results. To avoid memory overflow, we place an upper limit on the size of *cache*. We then compute the answers of the A/R part with the *newEval* function which exploits the answers already computed and stored in *cache*. In other words, if part of the query has been already computed, it retrieves the answers and joins them with the part of the query that has not been executed. Finally, we join the answers of the exact part of the query with those of the A/R part.

For query Q_1 the optimised evaluation slightly worsens the computation time. This is due to the extra time spent by the evaluation algorithm in undertaking the caching. In fact, in general for single-conjunct queries the optimisation does not speed up the computation[6].

For queries Q_2 and Q_4 the optimised evaluation decreases the execution time somewhat. For query Q_2, since the number of answers is rather large, it is hard to compute all these answers in a shorter amount of time even with the optimised evaluation.

The optimised evaluation of query Q_3 performs worse than the simple evaluation. The main reason is that the exact sub-query returns a large number of answers, namely, every event along with its label and date. These answers are stored in the

[6]In the final version of the system, the optimisation module would be disabled for single-conjunct queries.

Algorithm 7: Flexible Query Evaluation – Optimised

input : Query Q; approx/relax max cost c; Graph G; Ontology K.
output: List M of mapping/cost pairs, sorted by cost.
$\pi_{\overrightarrow{w}}$:= head of Q;
Q_E := sub query of Q with only the triples that are not approxed nor relaxed.;
M_s := eval(Q_E,G);
if M_s *is empty* **then**
  return $\emptyset$
cache := $\emptyset$; `/* set of pairs of query/evaluation results */`
Q_{AR} := sub-query of Q comprising only the approxed and relaxed triples;
$M := \emptyset$;
foreach $\langle Q', cost\rangle \in$ *rewrite*(Q_{AR},c,K) **do**
  if *cache is not full* **then**
    foreach *pair* $t, t' \in Q'$ **do**
      $cache := cache \cup \langle t, eval(t, G)\rangle$;
      $cache := cache \cup \langle t \text{ AND } t', eval(t \text{ AND } t', G)\rangle$
  foreach $\langle \mu, 0\rangle \in newEval(Q', preEval, G)$ **do**
    $M := M \cup \{\langle \mu, cost\rangle\}$;
return $\pi_{\overrightarrow{w}}(M \bowtie M_s)$;

cache and then retrieved for the final join with the relaxed triple pattern.

Queries Q_6 and Q_8 can now be computed in a reasonable amount of time. Q_9 can also be computed with the optimised algorithm. The time taken is less than 0.1 seconds due to the fact that its exact part returns no answers, making the computation of the rest of the query redundant.

Query Q_{10} can now be computed and, in fact, the optimised algorithm managed to run the A/R form of the query faster than its exact form. It is possible that the Jena SPARQL evaluator does not perform optimally for this particular query, but splitting the query into multiple parts and joining the results separately, as we do in our optimisation technique, improves the evaluation time considerably.

We are still unable to execute the A/R forms of queries Q_5 and Q_7. For query Q_5, the long evaluation time is due to the presence of the Kleene-closure and the wild-card symbol "_". On the other hand query Q_7 cannot be computed because of the join structure of the exact part of the query, which is the following:

```
?p <wasBornIn> ?b .
?p <wasBornOnDate> ?d .
?p rdfs:label ?n2.
?a rdfs:label ?n1 .
```

We can see that the variable ?p appears in the first three triple patterns, while the variables of the last triple pattern do not appear anywhere else in the query. Therefore, Jena has to compute a Cartesian product between the 2954875 answers retrieved for the last triple pattern and the 500000 answers retrieved for the first three triple patterns. More sophisticated optimisation techniques need to be investigated to improve the performance of these queries.

The overall results show that the evaluation of SPARQL^{AR} queries through a query rewriting approach is promising. The difference between the execution time of the exact form and the A/R form of the queries is acceptable for queries with fewer than 5 conjuncts. For most of the other queries, the simple optimisation technique described above also brings down the running times of the A/R forms to more reasonable levels. Clearly, for more complex queries, more sophisticated optimisation techniques need to be investigated and developed.

6. Conclusions

In this paper we have presented query processing algorithms for an extended fragment of the SPARQL 1.1 language, incorporating approxima-

tion 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 show promising query processing performance, but also that further optimisations are required.

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 the query containment problem 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 $[[Q]]_{G,K} = [[Q_1]]_{G,K}$.

Another area of ongoing work involves the construction of synopses (or data guides) of RDF-datasets in order to speed up query evaluation. In our context, such a synopsis is a graph S constructed from an RDF-dataset G that will have the following property: if we consider G and S as automata, then $\mathcal{L}(G) \subseteq \mathcal{L}(S)$. Hence, given a query Q, if $eval(Q, S) = \emptyset$ then $eval(Q, G) = \emptyset$. Since the synopsis S will be considerably smaller than G, we can evaluate Q over S for each query Q generated by the rewriting algorithm; if $eval(Q, S)$ returns no answer, then we do not need to execute Q over G. Moreover, the synopsis S can be exploited in order to remove the _ symbol that is generated when we apply APPROX to a triple pattern of a query. Given a triple pattern $\langle x, P, y\rangle$ from query Q, we compute $A = M_P \cap S$, where M_P is the automaton that recognises $\mathcal{L}(P)$. Subsequently, we replace $\langle x, P, y\rangle$ with $\langle x, P_A, y\rangle$ in Q, where P_A is a property path that does not contain the symbol _ such that $\mathcal{L}(P_A) = \mathcal{L}(A)$.

Another direction of research is the extension of our approximation and relaxation operators, query evaluation and query optimisation techniques to flexible federated query processing for SPARQL 1.1. Finally, also planned is a detailed comparison of the query rewriting approach to query approximation and relaxation presented here with the "native" implementation of similar operators described in [23].

Acknowledgements. Andrea Calì acknowledges partial support by the EPSRC project "Logic-based Integration and Querying of Unindexed Data" (EP/E010865/1)".

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 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. A. Hurtado, A. Poulovassilis, and P. T. Wood. Query relaxation in RDF. *Journal on Data Semantics*, X:31–61, 2008.

[14] 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.

[15] S. Muñoz, J. Pérez, and C. Gutierrez. Minimal deductive systems for RDF. In *Proceedings of the 4th European Conference on The Semantic Web: Research and Applications*, ESWC '07, pages 53–67, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] 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.

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

[18] 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.

[19] C. Pradel, O. Haemmerlé, and N. Hernandez. Natural language query interpretation into SPARQL using patterns. In *Fourth International Workshop on Consuming Linked Data - COLD 2013*, pages pp. 1–12, Sydney, AU, 2013. Kent State UNiversity. Thanks to Ceur-ws editor. The definitive version is available at http://ceur-ws.org/Vol-1034/.

[20] 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.

[21] M. Sassi, O. Tlili, and H. Ounelli. Approximate query processing for database flexible querying with aggregates. In A. Hameurlain, J. Küng, and R. Wagner, editors, *Transactions on Large-Scale Data- and Knowledge-Centered Systems V*, pages 1–27. Springer-Verlag, Berlin, Heidelberg, 2012.

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

[23] P. Selmer, A. Poulovassilis, and P. T. Wood. Implementing flexible operators for regular path queries. In *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT), Brussels, Belgium, March 27th, 2015.*, pages 149–156, 2015.

Appendix

Proof of Theorem 1.

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 $|E|$. □

Proof of Theorem 2.

Proof. 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 $O(|E| \cdot |S|^2 + |N| \cdot |S| \cdot \log(|N| \cdot |S|))$. □

Proof of Theorem 3

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:

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

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

Proof of Lemma 1

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. □

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 [18]):

- 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 $O((|E| + |N|) \cdot |S|^2 + |N| \cdot |S| \cdot \log(|N| \cdot |S|))$. □

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 [18]). 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 $O(|E| \cdot |S'|^2 + |N| \cdot |S'| \cdot \log(|N| \cdot |S'|))$. □

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. □

Proof of Theorem 4.

Proof. Follows straightforwardly from Theorem 3 and Lemma 1. □

Proof of Theorem 5.

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. □

Proof of Theorem 6

Proof. We give an NP algorithm, EvaluationCost shown as Algorithm 8, for the EVALUATION problem for a generic query Q containing AND, UNION and regular expression patterns. The EvaluationCost algorithm takes as input a

mapping μ, a graph G and a query Q and returns a cost c. Given an evaluation $\langle \mu, c' \rangle$, and a query Q, then the EvaluationCost algorithm returns c if $\{\langle \mu, c \rangle\} \in [[Q]]_G$ and $NULL$ otherwise. Finally, we need to check that c is equal to c'.

It is easy to see in the EvaluationCost algorithm that the non-deterministic step occurs when the condition $Q = Q_1$ AND Q_2 is satisfied, in which case we need to guess a decomposition of the mapping μ into μ_1 and μ_2. The number of guesses is bounded by the number of possible decompositions of μ (which is finite).

Algorithm 8: EvaluationCost

```
input : Query Q with no variables, a
        mapping μ, a graph G
output: A cost value c, or NULL
if Q = t then
    if there exists a cost such that
    {⟨μ, cost⟩} ∈ [[t]]_G, where t is a simple
    triple pattern or an APPROX/RELAX
    then
        return cost;
    else
        return NULL
else if Q = Q1 AND Q2 then
    Guess a decomposition μ = μ1 ∪ μ2;
    if EvaluationCost(Q1,μ1,G) ≠ NULL
    and EvaluationCost(Q2,μ2,G) ≠ NULL
    then
        return EvaluationCost(Q1,μ1,G) +
        EvaluationCost(Q2,μ2,G);
    else
        return NULL
else if Q = Q1 UNION Q2 then
    if EvaluationCost(Q1,μ,G) = NULL then
        return EvaluationCost(Q2,μ,G);
    else if EvaluationCost(Q2,μ,G) = NULL
    then
        return EvaluationCost(Q1,μ,G);
    else if EvaluationCost(Q1,μ,G) ≤
    EvaluationCost(Q2,μ,G) then
        return EvaluationCost(Q1,μ,G);
    else
        return EvaluationCost(Q2,μ,G);
```

□

Proof of Theorem 7.

Proof. We first show that the evaluation problem is in NP. Given a pair $\langle \mu, cost \rangle$ and a query SELECT$_{\overrightarrow{w}}Q$, where Q contains AND, UNION and SELECT, 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 NP time as we have seen in Theorem 6). 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 reduce from the 3-SAT problem, which is known to be NP-complete: Given a Boolean formula $\phi = C_1 \wedge \cdots \wedge C_n$ as input, where each clause C_i is a disjunct of exactly three literals, is ϕ satisfiable? Each literal l_{i1}, l_{i2} and l_{i3} of C_i is either a variable v_k or a negated variable $\bar{v}_k$, with $k \in \{1, \ldots, m\}$.

We start by constructing the following graph:

$$G = (\{a, 0, 1\}, \{f, t\}, \{\langle\langle a, f, 0\rangle, 0\rangle, \langle\langle a, t, 1\rangle, 0\rangle\})$$

We then construct the query $Q = \text{SELECT}_z(T_1$ AND $\cdots T_n$ AND $\langle a, t, z \rangle)$, where each symbol $T_i = \{V_{i1}$ UNION V_{i2} UNION $V_{i3}\}$ corresponds to a clause C_i in ϕ, and each V_{ij} either corresponds to a triple of the form $\langle a, t, x_k \rangle$ if l_{ij} is a variable v_k, or corresponds to a triple of the form $\langle a, f, x_k \rangle$ if l_{ij} is a negated variable $\bar{v}_k$.

It can be verified that the formula ϕ is satisfiable if and only if $\langle \mu, 0 \rangle \in [[Q]]_G$ with $\mu = \{z \to 1\}$. □

Proof of Theorem 8

Proof. In Theorem 6, the non-deterministic steps (i.e. the decomposition of the mapping μ into μ_1 and μ_2 to verify that $\langle \mu, c \rangle \in [[Q_1 \text{ AND } Q_2]]_G$), depend on the query Q which we assume is fixed. To verify that an evaluation $\langle \mu, 0 \rangle$ is in $[[t]]_G$, with t a triple pattern of query Q, can be done in $|E|$ steps. Therefore, the evaluation can be computed in $O(|E| * |\mu|^{|Q|})$ steps.

When we include the SELECT operator we need to add a further non-deterministic step, that is, generating a new mapping μ' from μ such that $\pi_{\overrightarrow{w}}(\langle \mu', c \rangle) = \langle \mu, c \rangle$. From the proof of Theorem 5 we can see that this can be done in $O(|ND|^{|var(Q)|})$. Since the query is fixed, we conclude that the data complexity is polynomial. □

Proof of Lemma 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 $\mu_1 \cup \mu_2$. □

Proof of Lemma 3

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. □

Proof of Theorem 9.

Proof. For ease of reading, in this proof we will replace the operators APPROX and RELAX with A and R respectively and will denote with $A/R(t)$ that we are applying either APPROX or RELAX to a triple pattern t. 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:

$$\text{CostProj}([[A/R(x,p,y)]]_{G,K}, 0) = \bigcup_{t' \in rew(A/R(x,p,y))_0} [[t']]_{G,K} \tag{5}$$

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 (5) 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:

$$\text{CostProj}([[A(x,p,y)]]_{G,K}, c) = \bigcup_{t' \in rew(A(x,p,y))_c} [[t']]_{G,K} \tag{6}$$

The induction hypothesis is that (6) holds 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 (6) is true when one of, i, j or k is incremented by 1.

Considering the RHS of equation (6), 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 ev-

ery possible edit operation to the original triple pattern, we have that:

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

Considering the LHS of equation (6), again by the semantics of approximation, we have that:

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

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

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

Given Corollary 1, it is sufficient to show that all the following equations hold:

$$[[\langle x,p,y\rangle]]_{G,K} = [[\langle x,p,y\rangle]]_{G,K} \tag{7}$$

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

$$\begin{array}{l}\bigcup_{t' \in rew(A(x,p/\_,y))_{c-\alpha}} [[t']]_{G,K} = \\ \text{CostProj}([[A(x,p/\_,y)]]_{G,K}, c-\alpha)\end{array} \tag{9}$$

$$\begin{array}{l}\bigcup_{t' \in rew(A(x,\epsilon,y))_{c-\beta}} [[t']]_{G,K} = \\ \text{CostProj}([[A(x,\epsilon,y)]]_{G,K}, c-\beta)\end{array} \tag{10}$$

$$\begin{array}{l}\bigcup_{t' \in rew(A(x,\_,y))_{c-\gamma}} [[t']]_{G,K} = \\ \text{CostProj}([[A(x,\_,y)]]_{G,K}, c-\gamma)\end{array} \tag{11}$$

Equation (7) is trivially true. Equations (10) and (11) 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 (8), 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 (8) to:

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

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

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

Therefore we need to show that the following holds:

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

Given Corollary 1 it is sufficient to show that:

$$\begin{array}{c}[[\langle x,\_,z\rangle]]_{G,K} = \\ \text{CostProj}([[A(x,\_,z)]]_{G,K}, c-\alpha)\end{array} \tag{12}$$

$$\bigcup_{t' \in rew(A(z,p,y))_{c-\alpha}} [[t']]_{G,K} = \text{CostProj}([[A(z,p,y)]]_{G,K}, c-\alpha) \quad (13)$$

Equation (12) holds by similar reasoning to equation (11). Equation (13) holds by the induction hypothesis.

Equation (9) can be shown to hold by similar reasoning to equation (8). We conclude that equation (6) holds for every $c \geq 0$.

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

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

The induction hypothesis is that (14) holds 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 Corollary 1 it is sufficient to show that the following hold::

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

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

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

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

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

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

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:

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

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:

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

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

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:

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

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:

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

It is possible to drop the outer CostProj operators on both sides of the above equation. Applying Corollary 1 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:

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

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:

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

By Corollary 1 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:

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

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:

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

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

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

By Corollary 1 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 Corollary 1 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 Corollary 1, 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.

□

Proof of Theorem 10.

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. □