Optimisation Techniques for Flexible SPARQL queries

RICCARDO FROSINI, Knowledge Lab, Birkbeck, University of London, UK
ALEXANDRA POULOVASSILIS, Knowledge Lab, Birkbeck, University of London, UK
PETER T. WOOD, Knowledge Lab, Birkbeck, University of London, UK
ANDREA CALÌ, Knowledge Lab, Birkbeck, University of London, UK and Oxford-Man Institute, UK

RDF datasets can be queried using the SPARQL language but are often irregularly structured and incomplete, which may make precise query formulation hard for users. The SPARQL^{AR} language extends SPARQL 1.1 with two operators — APPROX and RELAX — so as to allow flexible querying over property paths. These operators encapsulate different dimensions of query flexibility, namely approximation and generalisation, and they allow users to query complex, heterogeneous knowledge graphs without needing to know precisely how the data is structured. Earlier work has described the syntax, semantics and complexity of SPARQL^{AR}, has demonstrated its practical feasibility, but has also highlighted the need for improving the speed of query evaluation. In the present paper, we focus on the design of two optimisation techniques targeted at speeding up the execution of SPARQL^{AR} queries and on their empirical evaluation on three knowledge graphs: LUBM, DBpedia and YAGO. We show that applying these optimisations can result in substantial improvements in the execution times of longer-running queries (sometimes by one or more orders of magnitude) without incurring significant performance penalties for fast queries.

CCS Concepts: • **Information systems** → *Query languages for non-relational engines*; **Query optimization**; *Resource Description Framework (RDF).*

Additional Key Words and Phrases: SPARQL 1.1, path queries, query approximation, query relaxation

1 INTRODUCTION

SPARQL is the predominant language for querying RDF data, which is the standard model for representing web data and more specifically Linked Open Data. However, datasets in RDF form can be hard to query by users if they do not have full knowledge of the structure of the data. Moreover, RDF datasets are often extracted from webpage content, leading to incomplete and irregular data. We have extended SPARQL 1.1 with two operators — APPROX and RELAX — with the aim of supporting users' flexible querying over the property path queries of SPARQL 1.1, calling this new language SPARQL^{AR}. APPROX and RELAX encapsulate different aspects of query flexibility: finding different answers and finding more general answers, respectively. APPROX automatically applies approximations to a SPARQL 1.1 property path, such as inserting a property, removing a property, or substituting a property by a different one. RELAX automatically generalises a property path by replacing a class by a superclass, a property by a superproperty, or a property by its domain or range type.

Extending SPARQL 1.1 with APPROX and RELAX allows users to query complex and heterogeneous knowledge graphs without the need to know precisely how the data is structured. To illustrate, suppose a user is querying

Authors' addresses: Riccardo Frosini, Knowledge Lab, Birkbeck, University of London, Malet Street, London, WC1E 7HX, UK, frosini.riccardo@gmail.com; Alexandra Poulovassilis, Knowledge Lab, Birkbeck, University of London, Malet Street, London, WC1E 7HX, UK, a.poulovassilis@bbk.ac.uk; Peter T. Wood, Knowledge Lab, Birkbeck, University of London, Malet Street, London, WC1E 7HX, UK, p.wood@bbk.ac.uk; Andrea Calì, Knowledge Lab, Birkbeck, University of London, Malet Street, London, WC1E 7HX, UK and Oxford-Man Institute, Eagle House,Walton Well Road, Oxford, OX2 6ED, UK, a.cali@bbk.ac.uk.

1559-1131/2022/6-ART $15.00
https://doi.org/10.1145/3532855

LUBM to find the titles of works co-authored by teachers and teaching assistants who teach on the same course, and poses the following SPARQL 1.1 query:

```
SELECT ?x ?t WHERE {
   ?x (publicationAuthor/teacherOf) ?c .
   ?x (publicationAuthor/teachingAssistantOf) ?c .
   ?x rdf:type Article . ?x title ?t }
```

This query returns no answers because in LUBM the property `title` is associated with the class `Person`, not the class `Article`. Applying APPROX to the last triple pattern allows `title` to be automatically replaced by all valid properties of articles (such as `name`, `publicationAuthor` and `publicationDate`), with a user-specified *cost* c_s being assigned to this substitution operation:

```
SELECT ?x ?t WHERE {
   ?x (publicationAuthor/teacherOf) ?c .
   ?x (publicationAuthor/teachingAssistantOf) ?c .
   ?x rdf:type Article . APPROX(?x title ?t) }
```

This query now returns the co-authored articles and their properties, with the cost c_s being associated with these answers. Applying also RELAX to the third triple pattern replaces `Article` by its superclass `Publication`, which also has additional subclasses such as `Book` and `Manual`, so that a broader set of co-authored works can be returned to the user than just articles. This relaxation operation, too, has a user-specified cost $c_{superclass}$ assigned to it, so the resulting query answers now have a cost of $c_s + c_{superclass}$ associated with them[1]:

```
SELECT ?x ?t WHERE {
   ?x (publicationAuthor/teacherOf) ?c .
   ?x (publicationAuthor/teachingAssistantOf) ?c .
   RELAX(?x rdf:type Article) . APPROX(?x title ?t) }
```

Several more examples of SPARQL 1.1 queries and their approximation and/or relaxation are given in Section 4.

The above examples illustrate how users can control where the APPROX and RELAX operators can be applied, as well as the costs associated with individual approximation and relaxation operations. Operation costs can be hidden from users if desired, by a system choosing appropriate default costs (e.g., a cost of one for each operation). To aid users in choosing which triple patterns to approximate or relax, a production version of our approach could include a facility that indicates to the user which triple patterns in their initial query are amenable to approximation and which to relaxation — this can be achieved by applying a first step of rewriting to each triple pattern and determining which triple patterns can be approximated/relaxed to produce a non-empty set of answers. Users would then be able to select which triple patterns to approximate/relax in the first instance.

In [28] we described the extension of SPARQL 1.1 with the APPROX and RELAX operations illustrated above and presented in detail the SPARQL^{AR} language, including its syntax, semantics and complexity of query answering. We also described a prototype implementation of SPARQL^{AR}, and conducted a query performance study over the YAGO knowledge graph which pointed to the practical feasibility of the approach but also highlighted the need for developing optimisation techniques for SPARQL^{AR} query evaluation. In the present paper, we focus on the design of two such optimisation techniques and on their empirical evaluation with respect to three knowledge

[1]In general, the overall cost of an approximated and/or relaxed query is the summed cost of the sequence of approximation or relaxation operations that have generated that query.

graphs: LUBM, DBpedia and YAGO. The contribution of the paper is the design of two optimisation techniques targeted at speeding up the execution of approximated and/or relaxed SPARQL queries, and the demonstration that applying these optimisations brings substantial improvements to the execution times of longer-running SPARQL^{AR} queries whilst not incurring a significant performance penalty for fast queries.

We continue the paper in Section 2 with an overview of related work on flexible querying in databases and the semantic web, and also on RDF data summarisation and query containment, which are the two query optimisation approaches that we explore later in the paper. In Section 3 we recall from [28] the syntax, semantics and implementation of SPARQL^{AR}, to the level of detail needed for our purposes here. Our implementation approach — originally described in [28], and improved here with a more effective query caching mechanism — is based on query rewriting. Section 4 presents a performance study of the baseline implementation described in Section 3, using three sets of SPARQL queries over the LUBM, DBpedia and YAGO knowedge graphs, respectively. Section 5 then motivates and describes our two optimisation techniques for SPARQL^{AR}, one based on constructing an RDF graph summary and using it to reduce the number of rewritten queries that need to be evaluated; and one based on detecting query containment relationships which again are used to reduce the amount of query evaluation that is undertaken. In Section 6 we undertake a performance study of these two optimisations, both individually and combined, and compare the optimised query evaluation against the baseline implementation results of Section 4. Section 7 draws our conclusions and directions of further work.

2 RELATED WORK

2.1 Flexible querying

Early work on relaxation for query languages over structured data models, such as SQL or OQL, explored the removal of a selection criterion from the query's WHERE clause or the 'widening' of a selection criterion so as to match a broader range of values [10, 32]. Another common early approach was fuzzy matching of a selection criterion, using a similarity function to determine the degree of matching of each query answer [8, 9, 29, 51]. Bordogna and G. Psaila [8] proposed 'soft' conditions that tolerate incomplete matchings by exploiting fuzzy set theory. Generalisation of queries through type abstraction was also proposed [18] as was the use of 'malleable' schemas comprising overlapping definitions of data structures and attributes [62].

Proposals for flexible querying of semi-structured data have included relaxing queries by removing conditions from XPath expressions [3], 'widening' queries by using knowledge from an ontology, thesaurus or schema [33, 44, 59], and fuzzy XPath query evaluation [2]. For flexible querying of RDF data, SPARQL's OPTIONAL clause [31] provides some flexibility by returning query answers that may fail to match specified triple patterns of the query. More generally, Hurtado et al. [37] proposed a RELAX operator that allows ontology-based relaxation of specified triple patterns; such triple patterns are rewritten into successively more general ones and answers are incrementally returned at increasing 'costs' from the exact form of the query. The RELAX operator that we support here is based on that work and we describe it in more detail in Section 3. Meng et al. [48] relax queries on RDF data based on user preferences, as do Dolog et al. [21].

In contrast to query relaxation, which aims to return additional answers compared to the original query, query approximation returns potentially different answers that may still be of relevance to the user. (We note that our focus in this paper is on *query approximation*, i.e. generating and precisely evaluating variants of the original query, rather than on *approximate query answering* using techniques such as histograms [39], wavelets [17] or sampling [5], which are complementary to our work.) Fink and Olteanu [24] discuss approximation of queries over probabilistic databases through specifying lower- and upper-bound queries for a given query. Buratti and Montesi [11] propose cost-based edit operations that transform one XQuery path expression into another. Grahne and Thomo [30] explored approximate matching of regular path queries (RPQs) over semi-structured data, using a weighted regular transducer to perform transformations on RPQs. Hurtado et al. [38] extended this approach to conjunctive regular path queries (CRPQs). The techniques proposed by Hurtado et al. [37] and Hurtado et al. [38] were combined by Poulovassilis and Wood [56] and Poulovassilis et al. [55] to support concurrently both relaxation and approximation of CRPQs.

Frosini et al. [12, 28] similarly investigate extensions to fragments of SPARQL 1.1 to allow both approximation and relaxation of its property path expressions; complexity bounds for several fragments are derived and it is shown that adding the APPROX and RELAX operators does not increase the theoretical complexity class of the language fragments studied.

Similarity-based techniques have also been proposed for flexibly querying RDF data e.g. similarity measures on literals or resource names [34, 42], structural similarity measures exploiting the graph structure of the data [19, 61], and ontology-driven similarity measures [35, 36, 57]. Other approaches include using knowledge of the semantic relationships between graph nodes for flexible query matching [47], using predicates to express flexible paths between graph nodes [16], and using transformation functions to map nodes/edges appearing in a graph query to matches in the data graph [60]. Elbassuoni et al. [22] propose keyword-based search extensions of SPARQL and IR-style ranking of query answers. De Virgilio et al. [20] propose a meta-level framework for integrating different flexible query processing methods on graph-structured data. Pivert et al. [54] undertake a survey of techniques for flexibly querying RDF data based on user preferences.

Unlike most of these works, our approach centres on query rewriting and we associate a specific rewritten query with each flexible answer returned to the user, thereby directly supporting an explanation of how query answers have arisen. Moreover, users can select which of the full range of approximation/relaxation operations they wish to be applied to which parts of their queries, and they can also set the cost of each approximation/relaxation operation (see Section 3.4).

A complementary direction of research is that of *semantic schema discovery* [41] which can aid users in querying complex, heterogeneous datasets by inferring additional schema structures upon which users can base their query formulations. In contrast, our flexible querying approach implicitly discovers relationships in the data and incorporates them into an automatic query rewriting process.

2.2 RDF summarisation

A considerable amount of work has been done on graph summarisation in general (e.g. see the survey by Liu et al. [45]) and on RDF data in particular (e.g. see the survey by Čebirić et al. [15]). Most of this work is concerned with using the summary as some form of substitute for the data graph, in the sense that the summary contains references to the original graph nodes. This is certainly true when the summary is used as an index, where often some notion of bisimulation is used to construct a so-called quotient graph from the original. A recent paper by Blume et al. [7] unifies this work on structural graph summarisation by presenting a common model in which such summaries can be defined.

Our application of summaries is different in that we are interested only in the sequences of labels appearing along paths in a graph (which we call *path labels*) rather than the nodes comprising the paths. As such, our approach is more similar to those used in text indexing and searching. In text processing, it is often useful to be able to recognise all substrings of a finite string or set of finite strings. Devices that recognise all substrings of a set of strings have been called *factor automata* [49]. The summary automata that we use to optimise SPARQL^{AR} queries are similar to factor automata in that, given a graph G and a predefined length k, they recognise precisely the path labels of length less than or equal to k appearing in G. However, they differ from factor automata in that they also recognise all the remaining (possibly infinite) path labels in G, as well as possibly some of length greater than k that do not appear in G.

Instead of using our summary automata to match path labels, we could have used quotient graphs based on bisimulation, or k-bisimulation [40] since we only handle exact matching up to a pre-defined path length k.

However, the path matching process using our summary automaton is more efficient than when using a quotient graph. Given a path label p of length k, our automaton can be used to check for the existence of p in time $O(k)$, whereas the time using a quotient graph is $O(n \cdot k)$, where n is the number of nodes in the quotient graph. Details of our approach are given in Section 5.1.

2.3 Query containment

The problem of query containment, for a particular query language, is fundamental and hence usually studied intensively. For conjunctive regular path queries (CRPQs), one of the earliest papers on containment was that by Florescu et al. [26]. They provided an EXPSPACE-algorithm to decide containment of CRPQs, but no lower bound for the problem was proven. Calvanese et al. [13] extended CRPQs to include an inverse operator, and showed that query containment is EXPSPACE-complete for this class of queries. More recently, Figuera et al. [23] show that, by considering various sub-classes of CRPQs which are known to occur frequently in practice, the complexity of deciding containment can be reduced.

For SPARQL 1.0, the complexity of query containment is the same as that for the relational algebra, given their equivalence in terms of expressive power [4]. A comprehensive analysis of the complexity of containment for several fragments of SPARQL is carried by by Pichler and Skritek [53]. Kostylev et al. [43] study containment for SPARQL 1.1, including property paths and most of the other features of the language, proving EXPSPACE lower bounds for those fragments where containment is decidable. On the other hand, Mailis et al. [46] consider simpler forms of conjunctive queries on RDF, for which they show that containment can be decided in polynomial time.

We already mentioned the paper by Grahne and Thomo [30] on applying approximation to regular path queries (RPQs). In that paper, they also define the problem of *approximate containment* between queries, showing that it is equivalent to deciding containment between a pair of regular expressions and hence PSPACE-complete. In our containment optimisation, we too check for containment between the regular expressions occurring in triple patterns in SPARQL^{AR}, extending this to a sufficient test for containment between SPARQL^{AR} queries. As a result, our method may miss some opportunities for removing redundant queries, but it remains in PSPACE rather than being in EXPSPACE, as would be the case for a complete algorithm based on CRPQs. Details of our approach are given in Section 5.2.

3 SPARQL^{AR} LANGUAGE AND IMPLEMENTATION

We begin by introducing the syntax and semantics SPARQL^{AR} in Section 3.1, to the level of detail needed for the present paper. We then specify in Section 3.2 the algorithms underpinning SPARQL^{AR} query evaluation. Next we describe in Section 3.3 our sub-query caching mechanism, which aims to improve query execution times by caching and reusing the answers to parts of queries. Finally we briefly describe our prototype SPARQL^{AR} implementation in Section 3.4, again to the level of detail necessary here.

3.1 Theoretical Foundations

To recall the syntax and semantics of SPARQL^{AR}, we first give some fundamental definitions, many taken from previous work (e.g. [52]), modified for our purposes:

Definition 3.1 (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 we omit blank nodes from triples because their use is discouraged for Linked Data since they represent resources identified by IDs that may not be unique in the dataset [6]. In practice, blank nodes are handled as URIs that do not identify a specific resource, both in the semantics of our language and in its implementation (see www.w3.org/TR/2014/REC-rdf11-mt-20140225/#blank-nodes).

Definition 3.2 (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 have modified the definition of an RDF-Graph from [52] to add weights to the edges — all set to 0 — which simplifies the formalisation of the SPARQL^{AR} semantics.

Definition 3.3 (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 respectively encompass the RDFS vocabulary fragment rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain and rdfs:range (known as ρDF [50]).

In an RDF-graph $G = (N, D, E)$, we assume that each node in N represents an instance or a class. 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 $K = (N_K, E_K)$ associated with an RDF-graph $G = (N, D, E)$ we assume that each node in N_K represents a class (a “class node”) or a property (a “property node”). The intersection $N \cap N_K$ is contained in the set of class nodes of N_K. D is contained in the set of property nodes of N_K.

Definition 3.4 (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.

For simplicity, we omit blank nodes from our language as these act as variables and a query containing blank nodes can be expressed in an equivalent form without using blank nodes (see www.w3.org/TR/2013/REC-sparql11-query-20130321/#QSynBlankNodes).

Definition 3.5 (Mapping). A *mapping* μ from ULV to UL is a partial function $\mu : ULV \rightarrow UL$. 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.

We now specify the syntax of SPARQL^{AR} regular expression patterns, query patterns and queries:

Definition 3.6 (Regular expression pattern). A SPARQL^{AR} *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.

The above is compliant with a fragment of the full property path syntax of SPARQL 1.1 (leaving out inverse paths and negated properties), with two extensions necessary for SPARQL^{AR}. Firstly, the symbol _ can appear in a rewritten SPARQL^{AR} query (but not in a user-submitted query). Prior to query evaluation (which we undertake by translating SPARQL^{AR} into SPARQL 1.1), _ is replaced by $!u$ (a negated property), where u is a URI that is known not to exist in the RDF-graph. Secondly, the symbol ϵ too can appear in a rewritten SPARQL^{AR} query (but not in a user-submitted query). Prior to query evaluation ϵ is replaced by $u?$.

Definition 3.7 (Query Pattern). A SPARQL^{AR} *query pattern* Q is defined as follows:

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

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 concrete SPARQL^{AR} syntax, a dot (.) is used for conjunction (as in SPARQL) but, for greater clarity, we use AND in the abstract SPARQL^{AR} syntax.

Definition 3.8 (SPARQL^{AR} Triple Pattern). A SPARQL^{AR} *triple pattern t* is defined as follows:
$t := UV{\times}UV{\times}UVL \mid UV{\times}RegEx(U){\times}UVL \mid \text{RELAX}(UV{\times}RegEx(U){\times}UVL) \mid \text{APPROX}(UV{\times}RegEx(U){\times}UVL)$
We denote by $var(t)$ the set of all variables occurring in a triple pattern t.

Definition 3.9 (Query). A SPARQL^{AR} query has the form SELECT$_{\overrightarrow{w}}$ WHERE Q, with Q a SPARQL^{AR} query pattern and $\overrightarrow{w} \subseteq var(Q)$. We may omit in the abstract syntax the keyword WHERE for simplicity. Given a SPARQL^{AR} query $Q' = \text{SELECT}_{\overrightarrow{w}}Q$, the *head* of Q', $head(Q')$, is $\overrightarrow{w}$ if $\overrightarrow{w} \neq \emptyset$ and is $var(Q)$ otherwise. We may omit in the abstract syntax the SELECT keyword if $\overrightarrow{w} = vars(Q)$.

The semantics of SPARQL^{AR} are an extension of the SPARQL 1.1 semantics to include the costs of applying the APPROX and RELAX operators. These costs determine the ranking of query 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 $\langle \mu, c\rangle$, where c is a non-negative integer indicating the cost of the answers arising from the mapping μ. More formally:

A *mapping* μ from ULV to UL is as defined earlier. Two mappings μ_1 and μ_2 are *compatible* if $\forall x \in var(\mu_1) \cap var(\mu_2)$, $\mu_1(x) = \mu_2(x)$. The *union* of two mappings $\mu = \mu_1 \cup \mu_2$ can be computed only if μ_1 and μ_2 are compatible. The resulting μ is a mapping such that $var(\mu) = var(\mu_1) \cup var(\mu_2)$ and: for each x in $var(\mu_1) \cap var(\mu_2)$, 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 $\mu(x) = \mu_2(x)$.

The result, M, of evaluating a SPARQL^{AR} query comprises a set of pairs of the form $\langle \mu, c\rangle$ where μ is a mapping and c is a non-negative integer indicating the cost of the answers arising from that mapping. We define the *union* and *join* of two sets of SPARQL^{AR} query evaluation results, M_1 and M_2 as follows:

$$\begin{aligned} M_1 \cup M_2 \quad &= \quad \{\langle \mu, c\rangle \mid \langle \mu, c_1\rangle \in M_1 \text{ or } \langle \mu, c_2\rangle \in M_2 \text{ with } c = c_1 \text{ if } \nexists c_2.\langle \mu, c_2\rangle \in M_2, \\ &\quad\quad c = c_2 \text{ if } \nexists c_1.\langle \mu, c_1\rangle \in M_1, \text{ and } c = min(c_1, c_2) \text{ otherwise}\} \\ M_1 \bowtie M_2 \quad &= \quad \{\langle \mu_1 \cup \mu_2, c_1 + c_2\rangle \mid \langle \mu_1, c_1\rangle \in M_1 \text{ and } \langle \mu_2, c_2\rangle \in M_2, \\ &\quad\quad \text{with } \mu_1 \text{ and } \mu_2 \text{ compatible mappings}\} \end{aligned}$$

The *semantics of a triple pattern* $\langle x, p, y\rangle$ *with respect to an RDF-graph G*, denoted $[\![\langle x, p, y\rangle]\!]_G$, is defined recursively as follows, where P, P_1, P_2 are regular expression patterns, x, y, z are in ULV, and $w_1, \ldots, w_n$ are fresh variables:

$$\begin{aligned} [\![\langle x, \epsilon, y\rangle]\!]_G \quad &= \quad \{\langle \mu, 0\rangle \mid var(\mu) = var(\langle x, \epsilon, y\rangle) \wedge \exists c \in N \,.\, \mu(x) = \mu(y) = c\} \\ [\![\langle x, z, y\rangle]\!]_G \quad &= \quad \{\langle \mu, c\rangle \mid var(\mu) = var(\langle x, z, y\rangle) \wedge \langle \mu(\langle x, z, y\rangle), c\rangle \in E\} \\ [\![\langle x, P_1|P_2, y\rangle]\!]_G \quad &= \quad [\![\langle x, P_1, y\rangle]\!]_G \cup [\![\langle x, P_2, y\rangle]\!]_G \\ [\![\langle x, P_1/P_2, y\rangle]\!]_G \quad &= \quad [\![\langle x, P_1, z\rangle]\!]_G \bowtie [\![\langle z, P_2, y\rangle]\!]_G \\ [\![\langle x, P^*, y\rangle]\!]_G \quad &= \quad [\![\langle x, \epsilon, y\rangle]\!]_G \cup [\![\langle x, P, y\rangle]\!]_G \cup \bigcup_{n\geq 1}\{\langle \mu, c\rangle \mid \langle \mu, c\rangle \in [\![\langle x, P, w_1\rangle]\!]_G \\ &\quad\quad \bowtie [\![\langle w_1, P, w_2\rangle]\!]_G \bowtie \cdots \bowtie [\![\langle w_n, P, y\rangle]\!]_G\} \end{aligned}$$

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

Fig. 1. RDFS entailment rules

A mapping μ *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 $\mu \models R_1$;

Finally, the semantics of SPARQL^{AR} queries not including APPROX and RELAX are 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 $\overrightarrow{w}$:

$$\begin{aligned}
[\![Q_1 \text{ AND } Q_2]\!]_G &= [\![Q_1]\!]_G \bowtie [\![Q_2]\!]_G \\
[\![Q_1 \text{ UNION } Q_2]\!]_G &= [\![Q_1]\!]_G \cup [\![Q_2]\!]_G \\
[\![Q \text{ FILTER } R]\!]_G &= \{\langle \mu, c\rangle \in [\![Q]\!]_G \mid \mu \models R\} \\
[\![\text{SELECT}_{\overrightarrow{w}} Q]\!]_G &= \pi_{\overrightarrow{w}}([\![Q]\!]_G)
\end{aligned}$$

3.1.1 Semantics of RELAX. The RELAX operator relies on the ρDF entailment rules shown in Figure 1. In order to guarantee that relaxed queries have unambiguous costs, we require that the ontology K is acyclic and also that its *extended reduction* is used (see [37] for detailed discussion). The extended reduction of K, $extRed(K)$, is computed as follows: (i) compute the closure of K under the rules of Figure 1; (ii) apply the rules of Figure 2 in reverse until no more rules can be applied[2]; (iii) for every pair of triples $\langle a, sp, b\rangle$ and $\langle b, sp, c\rangle$ in K, apply rule 1 of Figure 1 in reverse unless there exists a URI d such that triples $\langle c, sp, d\rangle$ and $\langle a, sp, d\rangle$ are also contained in K[3]; (iv) for every pair of triples $\langle a, sc, b\rangle$ and $\langle b, sc, c\rangle$ in K, apply rule 3 of Figure 1 in reverse unless there exists a URI d such that triples $\langle c, sc, d\rangle$ and $\langle a, sc, d\rangle$ are also contained in K.

For the purposes of SPARQL^{AR} query evaluation, we assume that $K = extRed(K)$. This means that only *direct relaxations* (see below) are applied to queries. These query relaxations correspond to the 'smallest' possible relaxation steps, thereby allowing an unambiguous cost to be assigned to rewritten queries (see [37] for detailed discussion and formal proofs).

Following the terminology of [37], 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. There is a cost c_{rule_i} associated

[2]Applying a rule in reverse means removing a triple that is deducible by the rule, i.e. if there are two triples t and t' that match the antecedent of a rule then remove a triple that can be derived from t and t' using that rule.
[3]The proviso "unless there exists a URL d such that ..." here, and in step (iv), ensures that removal of redundant triples happens from the 'outside-in' for the transitive sc and sp properties, so as to correctly minimise them.

$$\text{(e1)}\ \frac{\langle b, dom, c\rangle\langle a, sp, b\rangle}{\langle a, dom, c\rangle} \quad \text{(e2)}\ \frac{\langle b, range, c\rangle\langle a, sp, b\rangle}{\langle a, range, c\rangle}$$

$$\text{(e3)}\ \frac{\langle a, dom, b\rangle\langle b, sc, c\rangle}{\langle a, dom, c\rangle} \quad \text{(e4)}\ \frac{\langle a, range, b\rangle\langle b, sc, c\rangle}{\langle a, range, c\rangle}$$

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

with the application of such a direct relaxation. For example, suppose the ontology K contains the triples $\langle worksOn, dom, FilmCrew\rangle$, $\langle worksOn, range, Film\rangle$, $\langle starsIn, dom, FilmStar\rangle$, $\langle starsIn, range, Film\rangle$, $\langle starsIn, sp, worksOn\rangle$, $\langle FimStar, sc, FilmCrew\rangle$. Then rule 5 can be used to deduce the triple pattern $\langle x, type, FilmStar\rangle$ from the triple patterns $\langle x, starsIn, Titanic\rangle$ and $\langle starsIn, dom, FilmStar\rangle$; rule 6 can be used to deduce triple pattern $\langle y, type, Film\rangle$ from triple patterns $\langle KateWinslett, starsIn, y\rangle$ and $\langle starsIn, range, Film\rangle$; rule 2 can be used to deduce triple pattern $\langle x, worksOn, y\rangle$ from triple patterns $\langle x, starsIn, y\rangle$ and $\langle starsIn, sp, worksOn\rangle$; and rule 4 can be used to deduce triple pattern $\langle x, type, FilmCrew\rangle$ from triple patterns $\langle x, type, FilmStar\rangle$ and $\langle FimStar, sc, FilmCrew\rangle$.

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 cost of such a sequence of direct relaxations is the sum of the individual costs of the direct relaxations in the sequence. 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 over all sequences of direct relaxations that derive $\langle x', p', y'\rangle$ from $\langle x, p, y\rangle$.

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

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

3.1.2 Semantics of APPROX. For query approximation, we apply edit operations that *delete*, *insert* or *substitute* a URI in a $SPARQL^{AR}$ regular expression pattern, as specified in Algorithm 8 in Appendix A. The application of such an edit operation has a non-negative cost c_d, c_i or c_s, respectively, associated with it. We write $p \rightsquigarrow^* P$ if a sequence of edit operations can be applied to a URI p to derive a regular expression pattern P. The *edit cost* of deriving P from p, denoted $ecost(p, P)$, is is the minimum cost over all sequences of edit operations that derive. P from p.

The semantics of the APPROX operator are as follows, where P, P_1, P_2 are regular expression patterns, x, y are in ULV, p, p' are in U, and $z, z_1, \ldots, z_n$ are fresh variables:

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

3.1.3 Complexity of SPARQL^{AR} query answering. We studied in [28] the data, query and combined complexity of SPARQL^{AR} query answering, extending earlier results from [52, 58] for simple SPARQL queries, from [1] for SPARQL with regular expression patterns to include our new flexible querying constructs, and from [12] to include also UNION in SPARQL^{AR}. We showed that the data complexity of SPARQL^{AR} is in PTIME, while the query and combined complexity are NP-Complete — we refer readers to [28] for details[4].

3.2 SPARQL^{AR} Query Processing

To evaluate a SPARQL^{AR} query Q that may contain occurrences of APPROX or RELAX, we make use of a *query rewriting algorithm* that incrementally creates a set of queries $\{Q_0, Q_1, \ldots\}$ not containing these operators, such that $\bigcup_i [\![Q_i]\!]_{G,K} = [\![Q]\!]_{G,K}$[5]. Our query rewriting algorithm has previously been described in [28]. For completeness we include it here as Algorithm 6 in Appendix A where an overview of its operation is also given. The cost of each query generated by the rewriting algorithm is the summed cost of the sequence of approximation or relaxation operations that have created the query. If the same query is generated more than once, only the one with the lowest cost is retained. The set of queries generated is kept sorted by increasing cost, and for practical reasons we limit the number of queries generated by bounding the cost of queries up to a maximum value c. The soundness, completeness and termination of the rewriting algorithm are proved in [28].

To compute the answers to a SPARQL^{AR} query Q, we use Algorithm 1. This calls the rewriting algorithm (function *rewrite*) to create the list of rewritten queries up to a maximum cost c, applies an evaluation function (*eval*) to each such query (in order of increasing cost of the queries), and assigns the cost of the query to each mapping returned by *eval*. If a particular mapping is created more than once, only the one with the lowest cost is retained. The set of mappings M is maintained in order of increasing cost.

We note that our algorithm for applying APPROX to a regular expression pattern (Algorithm 8 in Appendix A) already includes some simple optimisations. To illustrate, consider a regular expression pattern comprising a single URI l. One step of approximation yields the set of $\langle expression, cost\rangle$ pairs $L_0 = \{\langle \epsilon, c_d\rangle, \langle !l, c_s\rangle, \langle (\_/l), c_i\rangle, \langle (l/\_), c_i\rangle\}$. where c_d is the cost of a deletion, c_s is the cost of a substitution, and c_i is the cost of an insertion. Naïvely applying approximation again would yield 16 pairs. However, the first **if** statement in Algorithm 8 ensures that the pair $\langle \epsilon, c_d\rangle$ is not approximated further (we assume that the cost of substitution is less than that of deletion plus insertion, so there is no point in inserting into the empty label).

[4]We also discussed in [28] how SPARQL^{AR} could be extended to allow OPTIONAL subqueries, with the triple patterns in such subqueries being able to have APPROX or RELAX applied to them. Since the combined complexity of SPARQL with OPTIONAL is PSPACE-complete, the combined complexity of SPARQL^{AR} would increase similarly.

[5]We note that the set of queries resulting from rewriting a SPARQL^{AR} query that includes APPROX/RELAX up to a specified maximum cost are themselves SPARQL 1.1 queries and therefore they could be expressed as a single SPARQL 1.1 query using SPARQL's UNION operator (losing, of course, the cost associated with each subquery).

ALGORITHM 1: Flexible Query Evaluation

```
input  : Query $Q$; approximation/relaxation 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
        $M := M \cup \{\langle \mu, cost \rangle\}$
    end
end
return M;
```

Next, we apply approximation to the remaining three pairs in L_0. Approximating $\langle !l, c_s \rangle$ yields the set $L_1 = \{\langle (\_/!l), (c_s + c_i) \rangle, \langle (!l/\_), (c_s + c_i) \rangle\}$. Only insertions are applied since deleting or substituting a substituted label will only generate queries at costs greater than those already generated. Approximating $\langle (\_/l), c_i \rangle$ yields the set $L_2 = \{\langle ((\_/\_)/l), (2 * c_i) \rangle, \langle \_, (c_i + c_d) \rangle, \langle (\_/!l), (c_i + c_s) \rangle, \langle (\_/(\_/l)), (2 * c_i) \rangle, \langle (\_/(l/\_)), (2 * c_i) \rangle\}$ The first of these pairs is equivalent to the fourth, while the third is the same as the first in L_1, so both pairs can be removed from L_2. Approximating $\langle (l/\_), c_i \rangle$ yields the set $L_3 = \{\langle \_, (c_i + c_d) \rangle, \langle (!l/\_), (c_i + c_s) \rangle, \langle ((\_/l)/\_), (2 * c_i) \rangle, \langle ((l/\_)/\_), (2 * c_i) \rangle, \langle (l/(\_/\_), (2 * c_i) \rangle\}$. The first of these pairs is the same as the second in (the original) L_2, while the second is the same as the second in L_1, so both can be removed. The third pair in L_3 is equivalent to the fifth in L_2, while the fourth and fifth pairs in L_3 are equivalent, so one can be removed. So after two rounds of approximation, we are left with $2 + 3 + 1 = 6$ rather than 16 new pairs.

3.3 Pre-Computation Optimisation

In order to speed up query execution, we have implemented an optimisation technique that caches and reuses the answers to selected subqueries of the queries generated by the rewriting algorithm. For each query Q generated by the rewriting algorithm, we generate a set QS of subqueries of Q using the ConnectedTripleSet function listed in Algorithm 2. Each query (i.e. set of triple patterns) $q \in QS$ contains (i) the exact part of Q, i.e. the triple patterns that are not approximated or relaxed, plus (ii) possibly additional approximated or relaxed triple patterns such that the following property holds: every $q' \subseteq q$ contains at least one triple pattern that shares a variable with a triple pattern from $q - q'$. If a query q has this property, we say that it is *connected*. When evaluating such a query q, it is never the case that for q', q'' with $q = q' \cup q''$ we have that $eval(q', G) \bowtie eval(q'', G) = eval(q', G) \times eval(q'', G)$. In other words, in creating an answer set for q from partial answer sets for pairs of its subqueries, it is never necessary to compute a Cartesian product[6]. We note that Algorithm 2 may return an empty set if it is not possible to generate any connected subqueries of Q satisfying criteria (i) and (ii).

[6]This contrasts with the pre-computation optimisation that we presented in [28] in which queries were split into their exact part and their approximated/relaxed part, irrespective of the join relationships between triple patterns, which led potentially to the breaking of join relationships and the need to compute the Cartesian product of two sets of pre-computed answers. We refer readers to [28] for a detailed description of this problem with that earlier version of our pre-computation optimisation.

ALGORITHM 2: ConnectedTripleSet Function

```
input  : a query $Q$.
output : Set of subqueries of $Q$, $QS$.
$QS := \emptyset$;
$Exact$ := subset of triple patterns of $Q$ that are not labelled with APPROX or RELAX;
foreach set of triple patterns $q$ such that $Exact \subseteq q \subset Q$ and $q$ is connected do
    $QS := QS \cup q$;
end
return $QS$;
```

Example 3.10. Given query $Q = (x_1, p_1, x_2)$ AND $APPROX(x_1, p_2, x_3)$ AND $RELAX(x_4, p_3, x_2)$ AND $RELAX(x_4, p_4, x_5)$, ConnectedTripleSet returns the following set of subqueries:

$$\{\{(x_1, p_1, x_2)\}, \tag{1}$$
$$\{(x_1, p_1, x_2), APPROX(x_1, p_2, x_3)\}, \tag{2}$$
$$\{(x_1, p_1, x_2), RELAX(x_4, p_3, x_2)\}, \tag{3}$$
$$\{(x_1, p_1, x_2), APPROX(x_1, p_2, x_3), RELAX(x_4, p_3, x_2)\}, \tag{4}$$
$$\{(x_1, p_1, x_2), RELAX(x_4, p_3, x_2), RELAX(x_4, p_4, x_5)\}\} \tag{5}$$

We see that each of these subqueries contains the exact part of Q (i.e. just the triple (x_1, p_1, x_2) in this example), plus potentially one or more additional approximated/relaxed triple patterns such that the subquery is connected. We notice that the triple pattern $RELAX(x_4, p_4, x_5)$ appears only together with triple pattern $RELAX(x_4, p_3, x_2)$ as these share the variable x_4 and the variable x_2 is needed to connect to the exact triple pattern (x_1, p_1, x_2).

Algorithm 3 specifies the evaluation of SPARQL^{AR} queries up to a maximum cost c, utilising a query cache and the ConnectedTripleSet function. The algorithm invokes in two places a modified query evaluation function *eval* which exploits now any sub-query answers that have already been computed and stored in the cache, retrieving such answers and combining or extending them to produce the complete answer set. Algorithm 3 begins by invoking the query rewriting algorithm described earlier. The ConnectedTripleSet function is used to generate a set of connected subqueries for each query Q' generated by the rewriting algorithm. Each such subquery Q'' whose results are not already in the cache is evaluated, using the *eval* function, and its results are stored in the cache; if parts of Q'' have already been evaluated and cached, then these answers are reused by *eval* and extended with further partial evaluation results to produce a full answer set for Q''. To avoid memory overflow, there is an upper limit on the size of the cache[7]. The answers of the full query Q' are then computed, using again the *eval* function, and added to the list of answers M.

In this paper, we use Algorithm 3 as the baseline for investigating the effectiveness of the two optimisations that we introduce in Section 5, since it is these two new optimisations that are our focus here. We refer readers to Chapter 6 of [27] for a detailed comparison of query performance with and without sub-query caching[8].

[7]Currently there is no cache replacement policy in our prototype. The cache is initialised when a new user query is submitted, and results caching ceases once the cache is full (omitted from Algorithm 3 for simplicity). It is an area of future work to implement and evaluate more sophisticated policies.

[8]In brief, the findings from that analysis are that the pre-computation optimisation reduces the execution time of queries that have a large number of rewritings due to its caching of partial answers, but that it may increase somewhat the execution time of queries with a low number of rewritings due to the additional subquery evaluation undertaken. The determination of the precise threshold of when to apply this optimisation would be implementation-dependent and in the case of our prototype implementation it is at around 15 rewritten queries.

ALGORITHM 3: Flexible Query Evaluation with Pre-Computation

```
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 $Q$;
$cache := \emptyset$ ;                /* cache is a set of pairs of query/evaluation results */
$M := \emptyset$;
foreach $\langle Q', cost\rangle \in rewrite(Q,c,K)$ do
    foreach $Q'' \in ConnectedTripleSet(Q')$ do
        if $Q''$ is not in cache then
            $cache := cache \cup \langle Q'', eval(Q'', cache, G)\rangle$ ;
        end
    end
    foreach $\langle \mu, 0\rangle \in eval(Q', cache, G)$ do
        $M := M \cup \{\langle \mu, cost\rangle\}$;
    end
end
return $\pi_{\overrightarrow{w}}(M)$;
```

3.4 SPARQL^{AR} Implementation

We have implemented a prototype that supports SPARQL^{AR} query evaluation, which is described in detail in [28] (the source code is available at https://github.com/riccardofrosini/SPARQLAR). Its user interface allows queries to be formulated and submitted, datasets and ontologies to be selected, and query answers to be incrementally returned to the user in order of increasing cost. Users can select which of the full range of approximation/relaxation operations they wish to be applied to which parts of their queries. They can also set the cost of each approximation/relaxation operation. Datasets are stored in Jena[9] using the TDB database[10]. Jena library methods are used to execute SPARQL 1.1 queries over the selected RDF dataset, as requested by the SPARQL^{AR} query evaluator (the *eval* function described above).

In subsequent sections of the paper we focus on the performance and optimisation of SPARQL^{AR} queries using this prototype. In practice, this SPARQL^{AR} query processing system would be part of a broader framework supporting flexible SPARQL query processing for end-users. For example, a keyword-based or natural language query interface could be supported, translating users' queries and their approximation/relaxation options into SPARQL^{AR}. To help users interpret the answers to their queries, in our current prototype users can ask to see which rewritten query has generated a given answer. This facility could be extended to allow users to request full query provenance information showing the sequence of approximation/relaxation steps that led to that query.

4 QUERY PERFORMANCE STUDY

For our empirical evaluation of SPARQL^{AR} query processing — i.e. Algorithm 3 described in the previous section — and of the optimised query processing described in Section 5, we use three datasets: LUBM[11] (Lehigh University

[9]https://jena.apache.org
[10]https://jena.apache.org/documentation/tdb/
[11]https://swat.cse.lehigh.edu/projects/lubm/

Benchmark), YAGO 3.0[12] and DBpedia[13]. The LUBM benchmark constructs datasets that describe universities, departments, professors, publications and students. By specifying the number of universities, the benchmark scales the size of the dataset. We set the number of universities to be 50 which generates an RDF dataset containing approximately 6,700,000 triples. The YAGO dataset integrates data from Wikipedia, Geonames and Wordnet. It contains approximately 120 million triples, corresponding to a size of 10 GB in the Jena TDB format. YAGO also records an estimated accuracy measure for each fact, which we removed for our performance study. The DBpedia dataset contains facts extracted from Wikipedia's *infobox*. It also contains triple patterns that record links between different Wikipedia pages and links to external sources, and redirection URLs from one page to another. It contains approximately 4,230,000 URLs and 62 million triples.

We defined five SPARQL^{AR} queries for each of LUBM, DBpedia and YAGO — which we call "top-level queries" below, to distinguish them from the sets of rewritten queries produced by the rewriting algorithm. Each query contains between one and seven triple patterns, some of which are approximated or relaxed in order to retrieve the answers that the user is seeking or that may more generally be of interest to the user. We explain each query, and how applying approximation or relaxation may be useful, in the following subsections.

The cost of each edit and relaxation operation is set to 1 for our performance study. We also assume that the user has selected the full range of edit and relaxation operations to be applicable by the APPROX and RELAX operators. We successively set the maximum cost of the desired query answers to be 1, 2 and 3. We report on the number of queries generated by the rewriting algorithm for each of these maximum costs, and on the time taken to execute each top-level query up to each maximum cost. The performance study was conducted using a Windows machine with 32GB of RAM and an Intel Core i7 processor. Each top-level query was executed 5 times; the first execution was discarded to account for Jena cache warm-up effects, and the query execution time we used was taken to be the average of the four subsequent executions. By 'query execution time' we mean the total time taken from submitting a top-level query to returning all the answers up to the specified maximum cost. For the purposes of this performance study, the cache relating to the pre-computation optimisation is emptied at the start of each top-level query evaluation.

4.1 LUBM

We first describe each of the five top-level queries run on the LUBM dataset. Although each query includes APPROX and/or RELAX operators, we first describe what the exact form of the query specifies (i.e. without the APPROX/RELAX operators) before motivating the inclusion of the APPROX and RELAX operators and describing the additional results returned.

Query Q_1 asks for the titles of articles written by a teacher and a teaching assistant who teach on the same course:

Q_1:
```
SELECT ?x ?t WHERE {
    ?x (publicationAuthor/teacherOf) ?c .
    ?x (publicationAuthor/teachingAssistantOf) ?c .
    RELAX(?x rdf:type Article) . APPROX(?x title ?t) }
```

The exact form of the query will not return any answers because in LUBM the property `title` has domain `Person`, not `Article`. By approximating the last triple pattern, the rewriting algorithm will replace the property `title` with the expression `!title` (amongst other edit operations) and the resulting query will return all properties of

[12]The version we use is from https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/archive The latest YAGO database can be found at https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/

[13]The data we use is from http://downloads.dbpedia.org/2016-04/. The latest database and information on DBpedia can be found at https://wiki.dbpedia.org/

articles other than `title`. Relaxing the triple pattern `(?x rdf:type Article)` also allows the user to see other types of publication that the authors have co-written, since `Article` has a superclass `Publication` which has subclasses such as `Book` and `Manual`.

Query Q_2 asks for every course that was taken by both `GraduateStudent1`, who has a Masters degree, and `Student25`, who is an alumnus of the same university from which `GraduateStudent1` obtained their Masters degree:

```
Q2:  SELECT ?c WHERE {
        RELAX(GraduateStudent1 (mastersDegreeFrom/hasAlumnus) Student25) .
        GraduateStudent1 takesCourse ?c .
        Student25 takesCourse ?c }
```

Because `GraduateStudent1` did not in fact get a Masters degree from the university of which `Student25` is an alumnus, the exact query returns no answers. By relaxing the first triple pattern, the property `mastersDegreeFrom` will be replaced by its super-property `degreeFrom`. Since `GraduateStudent1` does in fact have other degrees from the university of which `Student25` is an alumnus, the rewritten query will return the courses taken by both of them, among other answers.

Query Q_3 asks for the publications co-authored by `AssociateProfessor3` and a student that she advises:

```
Q3:  SELECT * WHERE {
        ?z publicationAuthor AssociateProfessor3.
        APPROX(?z (publicationAuthor/advisor) AssociateProfessor3) }
```

The exact form of this query does return some answers. Applying APPROX to the second triple pattern allows the property `advisor` to be dropped from the query (among other edits), allowing the user to see more generally all the publications written by `AssociateProfessor3`.

Query Q_4 asks for every undergraduate student `?s` and course `?c`, such that `?s` has email address `"UndergraduateStudent5@Depart`
`?c` is taught by an `AssistantProfessor`, and `?s` takes `?c`:

```
Q4:  SELECT ?s ?c WHERE {
        ?x rdf:type AssistantProfessor . ?x teacherOf ?c .
        ?s takesCourse ?c . RELAX(?s rdf:type UndergraduateStudent) .
       APPROX(?s address "UndergraduateStudent5@Department1.University0.edu") }
```

The property `address` is not present in LUBM, so the exact query will not return any answers. Applying APPROX to the last triple pattern allows `address` to be replaced by `!address`, which will match the property `emailAddress` among others. The resulting query would still not return any answers since the student with the specified email address is in fact a graduate student rather than an undergraduate student. The RELAX operator on the fourth triple pattern allows graduate students with that email address to also be returned.

Query Q_5 asks for every assistant professor who is an author of `Publication0` and works for an organization that has `ResearchGroup3` as a sub-organization:

```
Q5:  SELECT ?p WHERE {
        RELAX(ResearchGroup3 subOrganizationOf* ?x) .
        RELAX(?p rdf:type AssistantProfessor) . ?p worksFor ?x  .
        Publication0 publicationAuthor ?p }
```

The exact form of the query returns no answers. Relaxing the first triple pattern allows its replacement by (`Organization rdf:type⁻/subOrganizationOf* ?x`), using the statement (`subOrganizationOf rdfs:range Organization`) from the LUBM ontology, thus removing the requirement for `ResearchGroup3`; the resulting query returns assistant professors who are authors of `Publication0`. Relaxing the second triple pattern allows its replacement by (`?p rdf:type Professor`), returning authors of `Publication0` who are professors (which includes `FullProfessor`, `AssociateProfessor` and `AssistantProfessor`); further steps of relaxation will replace `Professor` by `Faculty` and then `Employee`.

Table 1 shows the number of queries generated by the rewriting algorithm for each of the queries Q_1 to Q_5 and each of the maximum costs 1, 2 and 3. The number of queries generated depends on the number of relaxed and approximated triple patterns, as well as on the length of the property path in each triple pattern, and may increase exponentially with respect to the maximum cost. In general, the APPROX operator generates a greater number of rewritten queries than the RELAX operator, since the latter is applicable only if the ontology contains specific rules related to the triple pattern being relaxed. We see that query Q_3 generates the largest number of rewritten queries at maximum cost 3 even though it has only one approximated triple pattern. This is because its property path comprises a concatenation of two URIs.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	6	4	8	6	3
2	16	7	28	16	5
3	30	7	64	30	7

Table 1. Number of LUBM queries generated by the rewriting algorithm, for maximum costs 1, 2 and 3.

Table 2 shows the number of answers returned by each top-level query when executed with maximum cost 0, 1, 2 and 3 (cost 0 corresponds to the exact form of the query). All queries except Q_1 return more answers after one step of approximation/relaxation. Query Q_1 returns more answers after two steps of approximation. We also notice that, after the first two steps (i.e. max cost 2), no additional answers are returned for any query. This is due to the highly structured nature of the LUBM dataset and the sparsity of the connections between URIs, which leave less scope for additional connections being discovered through the flexible query processing.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
0	0	0	7	0	0
1	0	3	13	1	1
2	2036	3	13	1	1
3	2036	3	13	1	1

Table 2. Number of answers returned by each LUBM query, for each maximum cost up to 3.

Table 3 shows the execution times in seconds of the five queries with maximum cost 1, 2 and 3. We see that, at all maximum costs, queries Q_3 and Q_4 take much longer to execute than the others. This is due to the APPROX operator introducing the _ symbol into more complex property paths. For example, in Q_3 the conjunct

```
APPROX(?z (publicationAuthor/advisor) AssociateProfessor3) .
```

is rewritten into the following alternative conjuncts (among others) after 1 step of approximation:

```
APPROX(?z (_/publicationAuthor/advisor) AssociateProfessor3)
APPROX(?z (publicationAuthor/_/advisor) AssociateProfessor3)
APPROX(?z (publicationAuthor/advisor/_) AssociateProfessor3)
```

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	0.76	0.83	154.14	125.21	0.92
2	2.64	1.88	221.42	162.39	1.73
3	9.66	2.04	243.22	177.33	2.21

Table 3. Execution times (seconds) for LUBM queries, for maximum costs 1, 2 and 3.

4.2 DBpedia

We first describe each of the five queries run on the DBpedia dataset.

Query Q_1 asks for those books which follow "The Hobbit" in the author's work.

Q_1:
```
SELECT ?y WHERE {
     APPROX(The_Hobbit subsequentWork* ?y). ?y rdf:type  Book }
```

The exact form of the query returns only the URI `The_Hobbit`, but the user would have expected further results in line with the three books comprising "The Lord of the Rings". Applying APPROX to the first triple pattern generates, among others, a query containing the triple pattern

```
The_Hobbit subsequentWork*/!subsequentwork/subsequentWork* ?y
```

which matches every book connected to the URI of the "The Hobbit" resource, including

```
dbr:The_Return_of_the_King
dbr:The_Two_Towers
dbr:The_Fellowship_of_the_Ring
```

Query Q_2 asks for the albums published by the Rolling Stones on the London Records label, along with the songs on each album.

Q_2:
```
SELECT ?x ?y WHERE {
     APPROX(?x albumBy The_Rolling_Stones) . ?x rdf:type Album .
     ?y album ?x . RELAX(?x recordLabel London_Records)
     } group by ?x
```

The exact form of this query returns no answers since the predicate `albumBy` is not present in DBpedia. Applying APPROX to the first triple pattern will replace `albumBy` by `!albumBy` which matches the predicate `artist`, among others, giving the desired results. Applying RELAX to the fourth triple pattern expands the results to encompass albums release by the Rolling Stones with other publishers.

Query Q_3 asks for anyone who died during the "Battle of Poitiers", along with the dates of both the battle and the person's death.

Q_3:
```
SELECT ?k ?d ?kd WHERE {
     APPROX(?k diedIn Battle_of_Poitiers) .
     Battle_of_Poitiers date ?d . ?k deathDate  ?kd }
```

The exact form of the query returns no answers since the predicate `diedIn` is not present in DBpedia. Applying APPROX to the first triple pattern will replace `diedIn` by `!diedIn`. The resulting query will return every person connected to the "Battle of Poitiers" (since the domain of `deathDate` has to be a person), including "Peter I, Duke of Bourbon" who is connected to the battle by the predicate `deathPlace`.

Query Q_4 asks for every Scientist who died in the 1800s during a duel.

```
Q4:  SELECT ?x WHERE {
        ?x subject Duelling_Fatalities . RELAX(?x deathDate "18xx-xx-xx") .
        RELAX(?x rdf:type Scientist) }
```

The exact form of the query returns no answers since the year "18xx-xx-xx" is not formatted correctly. Applying RELAX to the second triple pattern replaces it by `?x rdf:type Person`, and the resulting query will return "Évariste Galois" amongst others. If the user wishes to see all people who died during a duel, applying RELAX to the last triple pattern replaces it by `?x rdf:type owl:Thing`, returning additional answers such as the scientist "Martin Lichtenstein", who is not explicitly classified as a scientist in DBpedia.

Query Q_5 asks for every person born in New York who has a parent who acted in the film "12 Angry Men", along with all the films in which the person acted.

```
Q5:  SELECT ?x ?f  WHERE {
        APPROX(12_Angry_Men_(1957_film) actor ?a) . ?x parent ?a .
        APPROX(?f actor ?x).
        RELAX(?x birthPlace New_York) }
```

The exact form of the query returns no answers since the predicate `actor` is not present in DBpedia. Applying APPROX to the first and third triple patterns will replace `actor` by `!actor`, which matches the predicate `starring` among others. The resource "New York" refers to the state rather than the city, and for this reason the resulting query will return fewer answers than expected since in DBpedia not every person is connected to their state of birth by the predicate `birthPlace`. Applying RELAX to the last triple pattern will cause it to be rewritten as `(?f, rdf:type, Person)`, removing the requirement that the person be born in New York.

Table 4 shows the number of queries generated by the rewriting algorithm for these five queries at cost 1, 2 and 3. Query Q_1 results in 3 rather than 4 rewritten queries at cost 1 because the rewriting algorithm recognises that applying deletion to a Kleene closure of a single URI results in an equivalent regular expression. Query Q_4 results in the fewest rewritten queries overall since it does not contain the APPROX operator. Query Q_5 results in the greatest number of rewritten queries since it has two approximated triple patterns. Query Q_3 results in only 6 new rewritings at cost 2 because some redundant rewritings are removed as explained in the illustration of applying APPROX to a regular expression pattern given at the end of Section 3.2.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	3	5	5	3	10
2	7	11	11	6	47
3	15	19	19	10	149

Table 4. Number of DBpedia queries generated by the rewriting algorithm, for maximum costs 1, 2 and 3.

Table 5 shows the number of answers returned by each top-level query when executed with maximum cost 1, 2 and 3. The numbers of answers for queries Q_1 and Q_2 are constrained by the use of the `rdf:type` property. In query Q_3 the constant URL `Battle_of_Poitiers` limits the number of answers since it has few connections to persons with a death date. Query Q_4 has only one variable ?x, that needs to be connected to the subject

`Duelling_Fatalities`, hence bounding the number of answers. Query Q_5 produces no answers at maximum cost 1 since it contains two occurrences of the `actor` property which does not appear in the data; hence, at least two edit operations are needed to produce a query that returns results.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
0	1	0	0	0	0
1	4	60	1	1	0
2	5	60	4	69	54
3	5	66	8	69	369

Table 5. Number of answers returned by each DBpedia query, for each maximum cost up to 3.

Table 6 shows the execution times of each of the queries with maximum cost 1, 2 and 3. The high execution time of query Q_1 with maximum cost 2 and 3 is due to the presence of the Kleene-closure which, combined with the APPROX operator, generates queries with concatenations of multiple Kleene-closure operations that are time consuming to evaluate. The high execution time of query Q_3 is because the variable ?k binds to many constants in the graph after application of the APPROX operator.

Query Q_5 has high execution time due to the large number of queries the rewriting algorithm generates, combined with the presence of two APPROX operators leading to multiple occurrences of the _ symbol. However, the pre-caching optimisation described earlier limits the increase in execution times for maximum costs 2 and 3.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	0.003	0.05	129.23	0.03	101.84
2	52.23	0.11	244.84	0.31	122.52
3	97.14	1.29	334.23	0.65	152.68

Table 6. Execution times (seconds) for DBpedia queries, for maximum costs 1, 2 and 3.

4.3 YAGO

We first describe each of the five queries run on the YAGO dataset.

Query Q_1 asks for the geographic coordinates of the “Battle of Waterloo”.

```
Q1:  SELECT * WHERE {
       APPROX(Battle_of_Waterloo happenedIn/(hasLongitude|hasLatitude) ?x) }
```

The exact form of this query returns no answers since YAGO does not record the geographic coordinates of the town of Waterloo. Applying APPROX to the query’s triple pattern will, among other edits, result in happenedIn being removed. The resulting query now returns the coordinates of the “Battle of Waterloo”, since these *a*re directly recorded in YAGO.

Query Q_2 asks for the family names of people who acted in the film “Tea with Mussolini”.

```
Q2:  SELECT * WHERE {
        ?x actedIn Tea_with_Mussolini . RELAX(?x hasFamilyName ?z) }
```

The exact form of the query returns only a partial set of answers because, in YAGO, some actors have only a first name (e.g. Cher) and others have their full name recorded using the predicate `rdfs:label`. Applying RELAX to the second triple pattern replaces “hasFamilyName” by “label”, returning the names of actors in the film that are recorded using the property “label” or using its sub-properties such as “hasGivenName”.

Query Q_3 asks for events taking place in Berkshire in 1643.

Q_3:
```
SELECT * WHERE {
    ?x rdf:type Event . ?x happenedOnDate "1643-##-##" .
    APPROX(?x happenedIn "Berkshire") }
```

The exact form of the query returns no answers since the predicate `happenedIn` is not directly connected to the literal “Berkshire”. Applying APPROX to the third triple pattern will generate, among other queries, the following query which will return all the events that occurred in 1643 in Berkshire:

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

Queries Q_4 and Q_5 are two of the most challenging queries that were attempted over YAGO in [28] (they were numbered Q6 and Q7 in that paper)[14]. Q_4 executed in under one second in its exact form but took over one minute with maximum cost set to 2. Q_5 executed in five seconds in its exact form but failed to terminate with maximum cost set to 2[15].

Q_4:
```
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) }
```

[14]Note that in the present paper we use a more recent, and larger, version of YAGO compared to that used in [28] and so the query timings we obtain here cannot be directly compared with the query timings in that paper

[15]The longest-running query in [28] was actually the one numbered Q5 in that paper: 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)}. This too failed to terminate with maximum cost set to 2. Running the exact form of this query with our current prototype, and on the most recent version of YAGO, returns 2,943,311 answers and takes over 17 hours due to the number of isLocatedIn edges present in the graph and the double presence of the Kleene-closure of isLocatedIn. It was not possible to execute within 24 hours even at just maximum cost 1 the approximated/relaxed version of this query, not even when we applied both of the new optimisations described Section 5. However, we would argue that, given the very large number of answers returned by the exact form of this query, this would not in reality be a query for which a user would be likely to request approximation or relaxation.

Q_5:
```
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 }
```

Q_4 returns returns every Chinese actor who played in American films and directed Australian films. The two uses of APPROX allow the relationships between the person and Australia and the USA to be approximated to other connections. The use of RELAX allows the class of the person to be relaxed to superclasses of 'actor'. Q_5 returns the labels of every event ?a and person ?p such that ?p was born in the same place and on the same day that ?a occurred. The one use of APPROX allows the entity ?a to be connected in potentially different ways to the entity ?p. The two uses of RELAX allow the properties 'happenedIn' and 'happenedOnDate' to be relaxed to their super-properties.

Table 7 shows the number of queries generated by the rewriting algorithm for each of the YAGO queries at maximum cost 1, 2 and 3. Query Q_1 results in a high number of queries even though it has only one triple pattern which is approximated. This is because its single triple pattern contains a complex property path, namely, the concatenation of a property with a disjunction of two properties. Query Q_2 generates only 2 queries for maximum costs 1, 2 and 3, since the RELAX operator can only be applied once to the triple pattern. Query Q_3 has only a single predicate in the property path in the approximated triple pattern which reduces the number of possible queries that can be generated. In contrast, queries Q_4 and Q_5 have 3 conjuncts each with a flexible operator which increases the number of queries to be evaluated considerably. Query Q_4 in particular has 2 approximated triple patterns containing complex property paths.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	12	2	5	16	8
2	50	2	11	119	30
3	120	2	19	560	74

Table 7. Number of YAGO queries generated by the rewriting algorithm, for maximum costs 1, 2 and 3.

Table 8 shows the number of answers returned by each top-level query when executed with maximum cost 0, 1, 2 and 3. The number of answers returned by query Q_1 increases exponentially with the maximum cost because the successive application of APPROX yields an exponentially increasing number of nodes that are reachable from the node `Battle_of_Waterloo`. Similarly, Q_5 contains only one constant literal, namely `Event`, which is part of an approximated triple pattern and so allows a large number of answers, while the two approximated triple patterns of Q_4 are able to match many distant nodes in the YAGO database graph. In contrast, the number of answers to query Q_2 does not increase as the maximum cost increases since the number of queries generated by the rewriting algorithm does not increase. For query Q_3, the number of answers is bounded by the presence of constants as well as the `rdf:type` property which constrains the variable `?x` to be of type `Event`.

Table 9 shows the execution times of each of the queries with maximum cost 1, 2 and 3. The execution times of Q_2 and Q_3 are low whereas Q_1, Q_4 and Q_5 take much more time, especially with maximum cost 2 and 3. This is due to the large number of queries generated by the rewriting algorithm and the exponential growth of the answer set.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
0	0	4	0	32	1450
1	1381	263	1	234	54023
2	18584	263	1	25199	73311
3	116082	263	1	33316	455181

Table 8. Number of answers returned by each YAGO query, for each maximum cost up to 3.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5
1	0.02	0.012	0.11	2.44	214.55
2	31.18	0.012	0.66	122.77	602.11
3	264.53	0.012	2.11	648.29	1301.54

Table 9. Execution times (seconds) for YAGO queries, for maximum costs 1, 2 and 3.

5 OPTIMISATION TECHNIQUES

In this section, we describe the two optimisation techniques that we implemented with the aim of improving the performance of our SPARQL^{AR} query processing prototype. The techniques were designed to identify rewritten queries that do not need to be evaluated because they return no answers, or no new answers.

We term the first optimisation technique, described in Section 5.1, the *summary optimisation*. This technique relies on constructing a graph summary from the RDF data being queried. This summary is then used to detect queries that can return no answers because they do not match the graph summary. We also use the summary to replace occurrences of the _ symbol and of expressions of the form $!p$ within rewritten queries by a disjunction of specific properties based on their presence in the summary, which we have empirically determined results in faster query execution.

We term the second optimisation technique, described in Section 5.2, the *containment optimisation* as it is based on query containment. Using this optimisation, a rewritten query q with cost c that can return only answers already returned by a query q' at cost $c' \leq c$ that contains q is discarded.

The improvements in performance provided by each of these optimisations individually and by their combined use are reported in Section 6.

5.1 Summary Optimisation

Our summary structure for an RDF-graph G is defined by a deterministic finite state automaton R_G constructed from the edge labels of paths in G of length less than or equal to a specified maximum length n. We term a sequence of edge labels in a path of G a *path label*. The set of sequences of labels of length up to n that is recognised by R_G corresponds precisely to the set of path labels in G of length up to n. Moreover, every path label in G of length greater than n is also recognised by R_G; however, there may be sequences of of labels of length greater than n recognised by R_G that do not correspond to any path label in G.

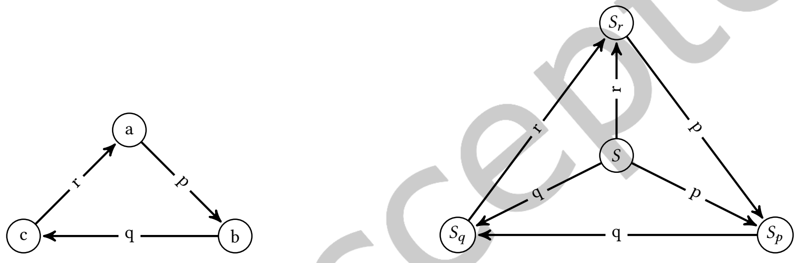
We take the common approach of viewing a graph G also as a non-deterministic finite state automaton A_G in which nodes and labelled edges in G become states and transitions in A_G, respectively, and each state in A_G is both an initial and a final state. Clearly, A_G accepts all path labels in G, i.e. for each path label s in G, $s \in \mathcal{L}(A_G)$. For a summary R_G of G, we prove in Proposition 5.4 below that $\mathcal{L}(G) \subseteq \mathcal{L}(R_G)$. Hence, if a path label appearing in a query is not in $\mathcal{L}(R_G)$, we can be certain that there are no such path labels in G. This allows us to remove such queries from the set of rewritten queries, thus speeding up query evaluation.

Given a graph G, the summary automaton R_G of length $n \geq 2$ for G is defined as follows:

(1) R_G has a state S which is both initial and final.
(2) For each path label $p_1p_2 \ldots p_k \in \mathcal{L}(G)$ with $1 \leq k < n$, R_G has states $S_{p_1}, S_{p_1p_2}, \ldots, S_{p_1 \ldots p_k}$, each of which is final, and transitions (S, p_1, S_{p_1}), $(S_{p_1}, p_2, S_{p_1p_2})$, $\ldots$, $(S_{p_1 \ldots p_{k-1}}, p_k, S_{p_1 \ldots p_k})$.
(3) For each path label $p_1p_2 \ldots p_n \in \mathcal{L}(G)$, R_G has the transition $(S_{p_1 \ldots p_{n-1}}, p_n, S_{p_2 \ldots p_n})$[16].
(4) There are no other states or transitions in R_G.

The states of the automaton R_G keep track of the last k transitions that have been traversed, for all $k < n$. So even if it has traversed more than $n-1$ states, R_G will keep track of only the last $n-1$ states.

Step (3) ensures that all path labels in the graph G are recognised by the automaton. To illustrate, consider the graph G and summary R_G shown in Figure 3 (we do not indicate the initial and final states, since S is always the initial state and all states are final). If we were to construct a summary R_G of length $n = 2$ using only steps (1) and (2) above, R_G would have only the three transitions from S: (S, p, S_p), (S, q, S_q) and (S, r, S_r). Hence, the summary would not recognise the path label pqr in G. Step (3) includes the transitions (S_r, p, S_p), (S_p, q, S_q) and (S_q, r, S_r) in R_G, as shown in Figure 3(b). Now all three path labels arising from the cycle in G are recognised by R_G.



(a) Graph G. (b) Summary R_G.

Fig. 3. Representing graph cycles in the summary.

We note that, in the worst case, the size of a summary of length n for a graph G can be $O(D^n)$, where D is the number of distinct property labels of G. However, in practice, RDF-graphs tend to be sparse, resulting in much smaller summaries.

Example 5.1. To illustrate the construction of our summary, we consider the graph G shown in Figure 4 which describes a portion of a film database. The summary of length $n = 2$ is formed from the following path labels of length 1 and 2 found in G:

```
actedIn, year, hasDirector,
actedIn/year, actedIn/hasDirector, hasDirector/actedIn
```

The resulting automaton R_2 is shown in Figure 5 (we use y, d and a to denote the properties *year*, *hasDirector* and *actedIn*, respectively). It is easy to verify that R_2 recognises every sequence in $\mathcal{L}(G)$; moreover, in this particular case, $\mathcal{L}(R_2) = \mathcal{L}(G)$.

If we now consider a summary of length $n = 3$, then we have the following 4 additional paths of length 3:

```
actedIn/hasDirector/actedIn, actedIn/hasDirector/year,
hasDirector/actedIn/year, hasDirector/actedIn/hasDirector
```

[16]Note that states $S_{p_1 \ldots p_{n-1}}$ and $S_{p_2 \ldots p_n}$ will both have been defined in step 2, with $k = n - 1$.

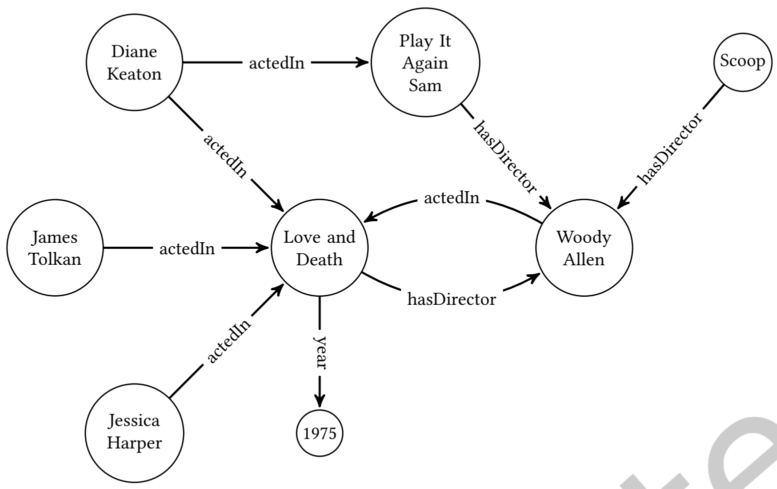


Fig. 4. Graph describing part of a film database.

and the resulting automaton R_3 will be as shown in Figure 6. We note that R_3 is larger than R_2, even though the two are in fact equivalent (i.e $\mathcal{L}(R_3) = \mathcal{L}(R_2)$): states X_{ay} and X_{dy} are equivalent to X_y, while X_{da} is equivalent to X_a and X_{ad} is equivalent to X_d.

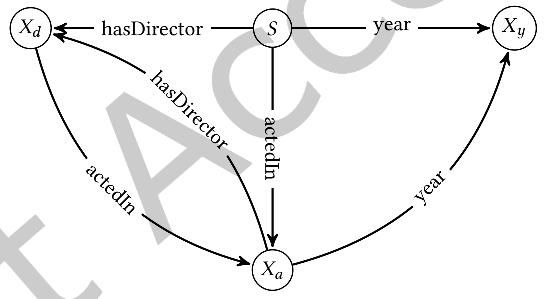


Fig. 5. Summary R_2 of size 2.

Although it happened to be the case that $\mathcal{L}(R_2) = \mathcal{L}(G)$ in the above example, this is of course not always true. For example, consider a graph G having path labels ab and bc, but no path label abc. A summary R_G of length 2 will recognise abc because R_G will have transitions (S, a, S_a), (S_a, b, S_b) and (S_b, c, S_c). We characterise in Proposition 5.4 below precisely the path labels recognised by a summary. First, we show two preliminary results.

Lemma 5.2. *Any summary R_G constructed from a graph G is deterministic.*

Proof. Suppose that R_G is of length n, $n \geq 2$. Every transition labelled a in R_G is either from a state S_u to a state S_{ua}, where u is a path label with $|u| < n - 2$ (step (2)), or from a state S_{bu} to a state S_{ua}, where u is a path label with $|u| = n - 2$ and b is an edge label (step (3)). In each case, the successor state for a transition is determined by the source state and the transition label, so there cannot be two transitions with the same label a leading to two different states S_{ua} and S_{va}, $u \neq v$. □

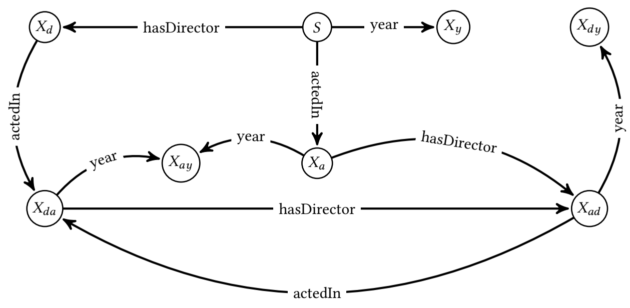


Fig. 6. Summary R_3 of size 3.

The next corollary follows from the above result.

Corollary 5.3. *Any string $s \in \mathcal{L}(R)$ is accepted in state S_s if $|s| \leq n-1$; and in state S_v, where $s = uv$ for label paths u and v with $|v| = n-1$, if $|s| \geq n$.*

We now provide a characterisation of the path labels accepted by a summary. For brevity, we abbreviate "path label" by "path" below, and use "subpath" to mean a contiguous sequence of edge labels taken from a path label.

Proposition 5.4. *Let R_G be a summary of an RDF graph G of length $n \geq 2$. Then for any path s, $s \in \mathcal{L}(R_G)$ if and only if for each subpath t of s, with $|t| \leq n$, $t \in \mathcal{L}(R_G)$.*

Proof. The proof is by induction on the length $|s|$ of s. We take $|s| \leq n$ as the base case since this is obviously true based on the construction of R_G from G.

Assume that the result holds for paths of length up to $n+k$, $k \geq 0$, and consider a path $s = aubvc$ of length $n+k+1$, where a, b and c are edge labels, u and v are path labels, and $|u| = k$. So $|bvc| = n$.

If $s \in \mathcal{L}(R_G)$, then by the inductive hypothesis, all subpaths of $aubv$ of length up to n are in $\mathcal{L}(R_G)$. Hence, all subpaths of ubv of length up to n are also in $\mathcal{L}(R_G)$. So we only need to show that all *suffixes* of $ubvc$ of length up to n are in $\mathcal{L}(R_G)$. But $|bvc| = n$, so the suffixes of length up to n do not include any part of u. We know by the inductive hypothesis that bv and all its subpaths are in $\mathcal{L}(R_G)$. By Corollary 5.3, bv must be accepted in state S_{bv} since $|bv| = n-1$. Since $s \in \mathcal{L}(R_G)$, a transition from S_{bv} to S_{vc} must have been added by step (3) of the construction of R_G. Hence all suffixes of s of length up to n must also be in $\mathcal{L}(R_G)$.

Conversely, suppose that for path s, each subpath t of s, with $|t| \leq n$, is in $\mathcal{L}(R_G)$. By the inductive hypothesis, we know that both $aubv$ and $ubvc$ are in $\mathcal{L}(R_G)$. From Corollary 5.3, we know that $aubv$ is accepted by R_G in state S_{bv} and that $ubvc$ is accepted in state S_{vc}. Furthermore, there must be a transition labelled c from S_{bv} to S_{vc}. Hence, $s = aubvc$ is in $\mathcal{L}(R_G)$. □

Corollary 5.5. *Let G be an RDF-graph and R_G be a summary of G of length $n \geq 2$. Then $\mathcal{L}(G) \subseteq \mathcal{L}(R_G)$.*

Proof. Let $s \in \mathcal{L}(G)$. Since every subpath of s of length at most n is in $\mathcal{L}(R_G)$, we conclude that $s \in \mathcal{L}(R_G)$. □

The following lemma allows us to avoid executing queries that will not return any answers by first testing them against the summary.

Lemma 5.6. *Let R_G be a summary of length n constructed from an RDF-graph G, and Q be a SPARQL^{AR} query without any APPROX, RELAX or UNION operators. If there exists a triple pattern $(x, P, y) \in Q$ such that P is a regular expression pattern and $\mathcal{L}(P) \cap \mathcal{L}(R_G) = \emptyset$, then $[\![Q]\!]_G = \emptyset$.*

Proof. Using Corollary 5.5, we can rewrite $\mathcal{L}(P) \cap \mathcal{L}(R_G) = \emptyset$ as $\mathcal{L}(P) \cap \mathcal{L}(G) = \emptyset$. Hence, $[\![\langle x, P, y\rangle]\!]_G = \emptyset$. For any query evaluation result M, we have that $M \bowtie \emptyset = \emptyset$, and therefore $[\![Q]\!]_G = \emptyset$. □

We can also exploit our summary to replace the symbol _ with a disjunction of specific edge labels. Queries containing _ are expensive to evaluate, since _ matches every edge label of a graph G. We similarly replace occurrences of expressions of the form $!p$ by a disjunction of specific edge labels (other than p). For example, consider the SPARQL^{AR} query $Q = \text{APPROX}(x, p_2/p_3, y)$ over an RDF-graph G. For a maximum cost of 1, the rewriting algorithm will generate queries of the form (x, R, y), where R is one of the following:

(1) p_2/p_3
(2) $\_/p_2/p_3$
(3) $p_2/\_/p_3$
(4) $p_2/p_3/\_$
(5) $!p_2/p_3$
(6) $p_2/!p_3$
(7) ϵ/p_3
(8) p_2/ϵ

Suppose that the full set of path labels of length 1 to 3 in G is $\{p_1, p_2, p_3, p_1p_3, p_2p_2, p_2p_3, p_2p_2p_2, p_2p_2p_3\}$. Then the summary automaton R_G with $n = 3$ extracted from G consists of the following transitions:

$$(S, p_1, S_1), (S, p_2, S_2), (S, p_3, S_3), (S_1, p_3, S_{1,3}),$$
$$(S_2, p_2, S_{2,2}), (S_2, p_3, S_{2,3}), (S_{2,2}, p_2, S_{2,2}), (S_{2,2}, p_3, S_{2,3}).$$

We can replace the _ and $!p_i$ symbols within expressions (2) to (6) above as follows:

- in (2) and (3) by p_2 to give $(p_2/p_2/p_3)$ in both cases, since $p_2p_2p_3$ is the only path of length 3 ending in p_3;
- in (5) by p_1 since p_1p_3 and p_2p_3 are the only paths of length 2 ending in p_3 but p_2p_3 cannot be matched to $!p_2/p_3$;
- in (6) by p_2 since p_2p_2 and p_2p_3 are the only paths of length 2 starting with p_2 but p_2p_3 cannot be matched to $p_2/!p_3$;
- in (4) we can detect that $(x, p_2/p_3/\_, y)$ returns no answers since there is no path in which p_3 is followed by another URI;

Given a query Q, Algorithm 4 uses the summary optimisation to determine which rewritten queries can be discarded because they will not return any answers. Algorithm 4 rewrites each query Q', generated from Q by the rewriting function *rew*, by replacing each property path P appearing in a triple pattern in Q' with the corresponding property path of the automaton constructed from the intersection of the summary graph R_G and A_P, the automaton recognising $\mathcal{L}(P)$. If any intersection is empty, then $Q'(G)$ will be empty as well and so does not need to be executed.

For some combinations of property path P and summary graph R_G, the intersection $P' := A_P \cap R_G$ can become very large. For example, consider DBpedia query Q_3 after two steps of approximation:

```
SELECT DISTINCT ?x ?y
WHERE {
  ?y album ?x .
  ?x rdf:type Album .
  ?x _/!albumBy <The_Rolling_Stones> .
  ?x recordLabel <London_Records> }
```

The third triple pattern has a regular expression that, when intersected with R_G, generates a graph with 79915 nodes. Trying to convert this graph back to a regular expression results in Jena returning a “stack overflow” error. To overcome this, we have empirically determined a heuristic for deciding when *not* to apply the summary optimisation. This is reflected in Algorithm 4, where the property path P is replaced by P' only if P' has length less than $MaxLen$, which we set at 50,000 following empirical investigation.

To apply the summary optimisation, we replace the *rewrite* function called in Algorithm 3 (i.e. Algorithm 6) by Algorithm 4 instead.

ALGORITHM 4: Rewriting queries using the summary optimisation

```
input  : Query $Q$; max approx/relax cost $c$; Graph $G$; Ontology $K$; length $n$ for summary.
output : List of ⟨query, cost⟩ pairs sorted by increasing cost.
$R_G$ := summary automaton of $G$ for paths up to length $n$;
$Qs := \emptyset$;
foreach $\langle Q', cost\rangle \in rew(Q,c,K)$ do
    toExecute := true;
    foreach triple pattern $(x, P, y) \in Q'$ do
        $A_P$ := an automaton that recognises $\mathcal{L}(P)$;
        $P'$ := a regular expression equivalent to $A_P \cap R_G$;
        if $\mathcal{L}(P') = \emptyset$ then toExecute := false;
        if $|P'| < MaxLen$ then $Q'$ := replace $(x, P, y)$ with $(x, P', y)$ in $Q'$;
    end
    if toExecute then $Qs := Qs \cup \{\langle Q', cost\rangle\}$ ;
end
return $Qs$;
```

5.2 Query containment

During the evaluation of SPARQL^{AR} queries, opportunities for applying optimisations based on query containment can arise in at least two ways. Firstly, a rewritten query Q may contain a triple pattern which is subsumed by another triple pattern in Q, as illustrated in the next example.

Example 5.7. Consider the LUBM query Q_4 from Section 4.1:

```
SELECT * WHERE {
    ?z publicationAuthor AssociateProfessor3 .
    APPROX(?z publicationAuthor/advisor AssociateProfessor3) }
```

Among other edits, the rewriting algorithm removes the property `advisor` from the second triple pattern, resulting in query Q_4':

```
SELECT * WHERE {
    ?z publicationAuthor AssociateProfessor3 .
    ?z publicationAuthor AssociateProfessor3 }
```

Clearly, one of the two triple patterns in Q_4' is redundant and can be removed prior to evaluation. Furthermore, any other rewriting of Q_4 with cost at least that of Q_4' will be contained in Q_4'.

The second opportunity arises when the application of an edit operation to a property path produces an expression that is contained in the set of property paths already produced (this corresponds to the case of containment of regular path queries under approximate semantics [30]).

Example 5.8. Consider the YAGO query Q_1 from Section 4.3:

```
SELECT * WHERE {
    APPROX(Battle_of_Waterloo happenedIn/(hasLongitude|hasLatitude) ?x) }
```

When rewriting the regular expression `happenedIn/(hasLongitude|hasLatitude)`, the expressions (1) `happenedIn/_/(hasLongitu`
(2) `happenedIn/_/hasLongitude` and (3) `happenedIn/_/hasLatitude` are generated, each at the cost of one insertion. Clearly, expressions (2) and (3) are each contained in expression (1) and can therefore be dropped from the set of queries to be evaluated and rewritten further.

When considering query containment for SPARQL^{AR} we also need to take into account the costs of the mappings. We assume, as is common practice, that the two queries being checked for containment have the same variables in their heads, noting that this will always in fact be the case in our context.

Definition 5.9. Given SPARQL^{AR} queries Q and Q', Q is *contained in* Q' (or Q' *contains* Q), denoted $Q \subseteq Q'$, if for each graph G and each pair $\langle \mu, k \rangle \in [\![Q]\!]_G$, there exists a pair $\langle \mu, k' \rangle \in [\![Q']\!]_G$ such that $k \geq k'$.

Note that the cost of the contained query must be at least that of the containing query (cf. [30]). For our purposes, we also define containment between SPARQL^{AR} triple patterns (once again assumed to refer to the same variables).

Definition 5.10. Given SPARQL^{AR} triple patterns t and t', t is *contained in* t' (or t' *contains* t), denoted $t \subseteq t'$, if for each graph G and each pair $\langle \mu, k \rangle \in [\![t]\!]_G$, there exists a pair $\langle \mu, k' \rangle \in [\![t']\!]_G$ such that $k \geq k'$.

Given a regular expression P, recall that we denote by $\mathcal{L}(P)$ the language denoted by P. The following proposition, stating that containment between two triple patterns $\langle x, P, y \rangle$ and $\langle x, P', y \rangle$ is equivalent to language containment between their regular expression patterns P and P', follows from the analogous result in [30]:

Proposition 5.11. *Given triple patterns $\langle x, P, y \rangle$ and $\langle x, P', y \rangle$, where P and P' are regular expression patterns, $\langle x, P, y \rangle \subseteq \langle x, P', y \rangle$ if and only if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$.*

A triple pattern in our context is equivalent to a regular path query (RPQ). Hence, testing containment between triple patterns is equivalent to testing containment between RPQs, a problem known to be PSPACE-complete [14]. Given regular expression patterns P and P', we can test $\mathcal{L}(P) \subseteq \mathcal{L}(P')$ by constructing deterministic finite automata M_P and $M_{P'}$ for P and P', respectively, and then using standard techniques to determine if $\mathcal{L}(M_P) \subseteq \mathcal{L}(M_{P'})$. The size of M_P may be exponential in the size of P, but since the decision problem is PSPACE-complete, no substantially better method is known. We use this method in our query containment algorithm (Algorithm 5) which we describe below. We note that if we were to allow inverse properties in our regular expression patterns, we would have to use two-way automata to decide containment of triple patterns (or 2RPQs); the complexity of checking containment of 2RPQs remains PSPACE-complete [14].

Algorithm 5 takes as input a query Q, and constructs Qs, the set of all queries generated from Q by the rewriting algorithm up to cost c. Then a query Q_1 is removed from Qs if there is another rewritten query Q_2 that contains Q_1 (as determined by the *contains* function), a process repeated until Qs no longer changes. The *minimise* function removes from a query any triple pattern which contains another, different triple pattern; it also removes triple patterns of the form `(?x type rdfs:Resource)` unless doing so would leave `?x` unbound, since everything is an `rdfs:Resource`.

The *contains* function takes as input queries Q_1 and Q_2. It checks that, for every triple pattern $\langle x, P, y \rangle$ in Q_1, there is a triple pattern $\langle x, P', y \rangle$ in Q_2 such that $\mathcal{L}(P') \subseteq \mathcal{L}(P)$.

We note that Algorithm 5 is sound but not complete, that is, it does not identify *all* possible containment relationships between queries since it undertakes only pairwise comparisons between individual triple patterns.

Similarly to Algorithm 4, Algorithm 5 returns a set of queries to be evaluated, hence we are able to apply the query containment optimisation by replacing the call to the *rewrite* function in Algorithm 3 by a call to Algorithm 5 instead.

ALGORITHM 5: Rewriting queries using the containment optimisation

```
input  : Query $Q$; approx/relax max cost $c$; Graph $G$; Ontology $K$.
output : List $Qs$ of query/cost pairs sorted by cost.
$Qs := \emptyset$;
foreach $Q' \in rew(minimise(Q),c,K)$ do $Qs := Qs \cup \{\langle \text{minimise}(Q'), cost' \rangle\}$;
while $Qs$ changes do
    if exist $\langle Q_1, cost_1 \rangle \in Qs$, $\langle Q_2, cost_2 \rangle \in Qs$ such that $Q_1 \neq Q_2$, $contains(Q_2, Q_1)$ and $cost_2 \leq cost_1$ then
        $Qs := Qs - \{\langle Q_1, cost_1 \rangle\}$;
    end
end
return $Qs$;
```

Function minimise(Q)

```
input  : Query $Q$.
output : Minimised $Q$.
foreach triple pattern $t \in Q$ do
    if $t$ is of the form $\langle x, type, rdfs{:}Resource \rangle$ then remove $t$ from $Q$ unless that would leave $x$ unbound;
    if there exists $t' \in Q$ such that $t' \neq t$ and $t' \subseteq t$ then remove $t$ from $Q$ ;
end
return $Q$;
```

Function contains(Q_1, Q_2)

```
input  : Queries $Q_1$ and $Q_2$.
output : True if $Q_1$ contains $Q_2$; false otherwise.
foreach triple pattern $\langle x, P, y \rangle \in Q_1$ do
    if $\nexists \langle x, P', y \rangle \in Q_2$ such that $M_{P'} \subseteq M_P$ then return False ;
end
return True;
```

6 QUERY PERFORMANCE WITH OPTIMISATIONS

In this section, we evaluate the effectiveness of the optimisations described in the previous section in terms of reducing both the numbers of queries to be evaluated and the time needed to execute them. We group the results by dataset so that comparisons between the methods can be seen side by side. In each case, we compare the numbers of queries or execution times of queries in four settings: (i) with no optimisation (other than subquery caching); (ii) with the containment optimisation; (iii) with the summary optimisation; and (iv) with the summary and containment optimisations combined. We choose this order of presentation because, in most cases, it demonstrates increasing improvement in performance. In (iv), the summary optimisation is applied first, followed by the containment optimisation, because this generally provides more opportunities for applying the containment optimisation and hence improving query execution times (we have empirically verified this by testing also the alternative ordering of the two optimisations). The query execution times reported in the following tables include the time for optimisation, where applicable.

6.1 LUBM

The summary of size 2 for LUBM has 68 transitions (and consumes 5 kilobytes), while the summary of size 3 has 122 transitions (9 kilobytes). For the experiments described here, we use the summary of size 3.

6.1.1 Number of rewritten queries generated. The numbers of rewritten queries generated under the four settings described above are shown in Table 10.

Max Cost	Optimisation	Q_1	Q_2	Q_3	Q_4	Q_5
1	none	6	4	8	6	3
1	containment	6	4	2	6	3
1	summary	3	1	5	3	3
1	combined	2	1	2	2	2
2	none	16	7	28	16	5
2	containment	16	7	2	16	5
2	summary	10	4	19	10	5
2	combined	7	2	2	7	3
3	none	30	7	64	30	7
3	containment	30	7	2	30	7
3	summary	19	3	53	19	7
3	combined	14	2	2	14	4

Table 10. Number of LUBM queries generated, with and without various optimisations.

The containment optimisation by itself only affects the number of queries generated for query Q_3. As we saw in Example 5.7, it detects that all queries resulting from one of the rewritten queries at cost 1 will be contained in it, meaning that only 2 queries are retained overall.

The summary optimisation tends to do better than the containment optimisation in reducing the number of queries generated. For example, it turns out that the property `hasAlumus` does not occur in the data, so 3 of the queries at maximum cost 1 for Q_2 are discarded, leaving only the query in which the first triple pattern is `(GraduateStudent1 (mastersDegreeFrom/type) University)`. On average, about 30% of queries are discarded by the summary optimisation.

When combining the summary optimisation with query containment, Table 10 shows in most cases a further reduction in the number of queries generated compared to the two separate optimisations. For example, the numbers of queries generated for queries Q_1 and Q_4 at maximum cost 3 reduce to 14 (from 19 and 30 for the summary and containment optimisations, respectively). The case of Q_5, where only the combination of the two optimisations is able to reduce the number of queries, is worthy of explanation.

The summary optimisation rewrites the first (relaxed) triple pattern of Q_5, namely `(ResearchGroup3 subOrganizationOf* ?x)` to

```
(ResearchGroup3 (ϵ | subOrganizationOf | (subOrganizationOf/subOrganizationOf)) ?x)
```

because there are at most two consecutive `subOrganizationOf` property labels in the data. Applying one step of relaxation produces

```
(Organization (^type | (^type/subOrganizationOf) |
              (^type/subOrganizationOf/subOrganizationOf)) ?x)
```

since the RDF ontology specifies that the domain of `subOrganizationOf` is of type `Organization`. Query containment now detects that the property path `^type` subsumes both `^type/subOrganizationOf` and `^type/subOrganizationOf/subOrga`
Furthermore, the triple pattern

```
(Organization (^type) ?x)
```

is subsumed by the third triple pattern in Q_5, namely, `(?p ub:worksFor ?x)`. These successful containment tests are only possible after the summary optimisation has replaced the Kleene closure by a disjunction of paths.

We see that, overall, by using the combined optimisations over 70% of queries are discarded.

6.1.2 Query execution times. Table 11 shows the execution times for each of the original queries (maximum cost 0), along with the times for each of the four settings using maximum costs of 1, 2 and 3, respectively.

Max Cost	Optimisation	Q_1	Q_2	Q_3	Q_4	Q_5
0	none	0.02	0.14	0.84	0.93	0.23
1	none	0.76	0.83	154.14	125.21	0.92
1	containment	0.79	0.87	2.27	125.7	0.93
1	summary	0.52	0.49	5.32	23.21	0.94
1	combined	0.59	0.47	2.29	9.85	0.70
2	none	2.64	1.88	221.42	162.39	1.73
2	containment	2.65	1.92	3.43	162.64	1.75
2	summary	2.35	1.75	9.24	97.37	1.76
2	combined	2.12	1.56	2.36	90.23	1.16
3	none	9.66	2.04	243.22	177.33	2.21
3	containment	9.67	2.06	4.42	178.62	2.23
3	summary	8.37	1.92	12.43	107.31	2.24
3	combined	7.26	1.52	2.68	103.12	1.40

Table 11. Execution times (seconds) for LUBM queries, using various optimisations.

The only reduction in execution time using query containment is for query Q_3, since that is the only query for which the containment optimisation reduces the number of queries generated, although the reduction in execution time is quite dramatic (from over 243 seconds to under 3 seconds at maximum cost 3).

Of course query execution time depends not only on the numbers of queries generated, as can be seen in queries Q_1 and Q_4, where the same numbers of queries are generated but execution times differ significantly. For Q_1 (as well as Q_2 and Q_5), query execution times are relatively short. At maximum cost 3, Q_4 takes almost 3 minutes to execute without any optimisation. The combined optimisation reduces this by almost 42%. The reductions for Q_1, Q_2 and Q_5 are somewhat less, at almost 25% for Q_1 and Q_2, and 36% for Q_5.

6.1.3 Overall observations. Overall, we see that the containment optimisation by itself brings limited benefit in reducing the number of rewritten queries generated. The summary optimisation does better than the containment optimisation in reducing the number of queries generated. Combining the two optimisations brings in most cases a further reduction in the number of queries generated compared to the two separate optimisations. A similar pattern is observed in the query execution times: there are limited improvements with query containment alone, more substantial improvements with the summary optimisation, and further improvement using both optimisations.

6.2 DBpedia

The summary of size 2 for DBpedia has 71,514 transitions (4.3 megabytes), while that of size 3 contains 1,109,836 transitions (67.3 megabytes). Attempting to rewrite the DBpedia queries using the summary of size 3 would produce many instances where the summary optimisation cannot be applied due to the *MaxLen* check in Algorithm 4. Therefore, for the experiments described here, we use the summary of size 2.

6.2.1 Number of rewritten queries generated. The numbers of rewritten queries generated, with and without the various optimisations, are shown in Table 12.

Max Cost	Optimisation	Q_1	Q_2	Q_3	Q_4	Q_5
1	none	3	5	5	3	10
1	containment	3	5	5	3	10
1	summary	3	1	1	1	0
1	combined	3	1	1	1	0
2	none	7	11	11	6	47
2	containment	7	11	11	6	46
2	summary	7	4	4	2	3
2	combined	7	4	4	2	3
3	none	15	19	19	10	149
3	containment	15	19	19	10	146
3	summary	15	8	8	4	18
3	combined	15	8	8	4	18

Table 12. Number of DBpedia queries generated, with and without various optimisations.

The query containment optimisation is only able to remove a few queries for Q_5 at maximum costs 2 and 3.

The summary optimisation is more effective than the containment optimisation in removing queries. The most striking case is for query Q_5 at maximum cost 3 when the number of queries is reduced from 149 to 18 (i.e., 88% of queries have been removed). For query Q_1, on the other hand, the summary optimisation was not able to discard any queries.

The number of queries generated by combining the two optimisations, in this case, is the same as the number generated by the summary optimisation.

6.2.2 Query execution times. Table 13 shows the execution times for each of the four settings using maximum costs of 1, 2 and 3[17].

In Table 13 we can see the overhead associated with the optimisation algorithms for queries that execute quickly, e.g., queries Q_1, Q_2 and Q_4 at maximum cost 1, where execution times of the queries without any optimisation are all under 53ms. The containment optimisation reduces the execution times slightly for Q_5 at maximum costs

[17]Even with the use of the summary of size 2, rather than size 3, there were a number of instances of query triple patterns where the summary optimisation was not applied, due to the check in Algorithm 4 that a triple pattern's regular expression P should only be replaced by the intersection, P', of the summary graph and the automaton recognising $\mathcal{L}(P)$ if P' has length less than *MaxLen*. Q_1: at maximum cost 2, the optimisation was not applied to four of the queries, and at max cost 3 it was not applied to twelve of the queries. Q_2: at max cost 2, the query's APPROX triple pattern was not replaced in two queries; at max cost 3 it was not replaced in six queries. Q_3: at max cost 2, the query's APPROX triple pattern was not replaced in two queries; at max cost 3 it was not replaced in six queries. Q_5: at max cost 2, the query's first APPROX triple pattern was not replaced in one query and its second APPROX triple pattern in two other queries; at max cost 3, the query's first APPROX triple pattern was not replaced in eleven queries and its second APPROX triple pattern in sixteen other queries.

Max Cost	Optimisation	Q_1	Q_2	Q_3	Q_4	Q_5
0	none	0.001	0.001	0.001	0.001	0.002
1	none	0.003	0.05	129.23	0.03	101.84
1	containment	0.13	0.19	129.25	0.15	102.00
1	summary	0.53	0.62	1.09	0.07	0.44
1	combined	0.69	0.82	1.13	0.19	0.49
2	none	52.23	0.11	244.84	0.31	122.52
2	containment	52.13	0.26	245.53	0.46	118.63
2	summary	22.18	0.87	2.15	0.23	1.82
2	combined	22.23	0.94	2.52	0.36	1.88
3	none	97.14	1.29	334.23	0.65	152.68
3	containment	97.20	1.32	335.74	0.69	145.11
3	summary	43.83	2.08	10.21	0.43	3.70
3	combined	43.93	2.17	11.20	0.56	3.86

Table 13. Execution times (seconds) for DBpedia queries, using various optimisations.

2 and 3 because there are fewer queries to execute. The summary optimisation is much more effective at reducing the execution times for queries Q_3 and Q_5, each by two orders of magnitude.

It is interesting to note that the summary optimisation also reduces the execution times for Q_1 at maximum costs 2 and 3, even though no fewer queries are produced. This is because Jena can make better use of indexes when the summary optimisation replaces occurrences of _ by the URIs of specific predicates.

6.2.3 Overall observations. Overall, we see again that the containment optimisation brings limited benefit in reducing the number of rewritten queries generated and that the summary optimisation is more effective. The number of queries generated by combining the two optimisations is, in this case, the same as the number generated by the summary optimisation alone. For queries that execute quickly, i.e. in under two seconds, we see that applying the optimisations does not incur a significant performance penalty — less than one second in all cases. For slower queries we see modest or no improvements in execution times with query containment alone, substantive improvements with the summary optimisation, and no significant further improvement when combining both optimisations.

6.3 YAGO

Compared to DBpedia, the summaries for the YAGO dataset are considerably smaller. For size 2, the summary has 1,320 transitions (88.2 kilobytes), while for size 3 it has 10,528 transition (708 kilobytes). For the experiments described here, we use the summary of size 3.

6.3.1 Number of rewritten queries. The numbers of rewritten queries generated, with and without the various optimisations, are shown in Table 14.

We see that it was not possible to eliminate any rewritten queries for query Q_2 since it contains only the RELAX operator. One or two queries were eliminated by one or other of the optimisations for Q_3 at each maximum cost, but the optimisations had the greatest success with Q_1. Here, the containment optimisation was able to discard a significant number of queries at maximum costs 2 and 3, with the summary optimisation doing slightly better in each case. Combining both optimisations resulted in further reductions in the numbers of queries. For example, the containment and summary optimisations each discarded two different queries at maximum cost 1, which meant that the combined optimisation was able to discard four queries. We notice that the summary optimisation

Max Cost	Optimisation	Q_1	Q_2	Q_3	Q_4	Q_5
1	none	12	2	5	16	8
1	containment	10	2	5	16	8
1	summary	10	2	4	16	5
1	combined	8	2	4	16	5
2	none	50	2	11	119	30
2	containment	40	2	11	119	30
2	summary	37	2	10	117	14
2	combined	27	2	10	117	14
3	none	120	2	19	560	74
3	containment	98	2	19	560	74
3	summary	85	2	18	542	28
3	combined	63	2	18	542	28

Table 14. Number of YAGO queries generated, with and without various optimisations.

also reduces the number of queries considerably for query Q_5, especially at maximum costs 2 and 3. On the other hand, only a small number of queries could be eliminated for queries Q_3 and Q_4.

6.3.2 Query execution times. Table 15 shows the execution times for each of the four settings using maximum costs of 1, 2 and 3[18].

Max Cost	Optimisation	Q_1	Q_2	Q_3	Q_4	Q_5
0	none	0.002	0.001	0.001	0.006	3.93
1	none	0.02	0.012	0.11	2.44	214.55
1	containment	0.03	0.013	0.12	2.56	214.78
1	summary	0.09	0.014	0.07	2.49	13.29
1	combined	0.10	0.015	0.09	2.78	13.66
2	none	31.18	0.012	0.66	122.77	602.11
2	containment	25.36	0.013	0.53	129.21	605.71
2	summary	22.13	0.014	0.58	22.87	19.32
2	combined	9.10	0.015	0.59	25.83	22.74
3	none	264.53	0.012	2.11	648.29	1301.54
3	containment	224.66	0.013	1.86	701.19	1307.41
3	summary	192.38	0.014	1.18	118.91	121.02
3	combined	183.14	0.015	1.19	154.10	129.51

Table 15. Execution times (seconds) for YAGO queries, using various optimisations.

As for the DBpedia queries, we see that the overhead of optimisation is noticeable for queries that execute quickly, such as queries Q_1, Q_2, Q_3 and Q_4 at maximum cost 1. We also see that the overhead of the containment optimisation by itself can be significant for slower queries such as Q_4 and Q_5 at maximum costs 2 and 3. In these

[18]There were some instances of query triple patterns where the summary optimisation was not applied due to the *MaxLen* check in Algorithm 4. Q_1: at max cost 3 the summary optimisation was not applied to twelve of the queries. Q_4: at max cost 3, the query's first triple pattern was not replaced in six queries and its second triple pattern in another six queries.

cases, no rewritten queries are discarded by the containment optimisation and there are many queries to check (e.g., 560 for Q_4 at maximum cost 3).

For query Q_1 only at maximum costs 2 and 3 do the benefits of optimisation become apparent. At maximum cost 2, the combined optimisation reduces the execution time to under a third of the time without optimisation. For query Q_3 at maximum cost 3, the summary and combined optimisations are able to almost halve the execution time. The biggest improvements can be seen for queries Q_4 and Q_5, where the summary optimisation reduces the time considerably for Q_4 at maximum costs 2 and 3, and for Q_5 at all maximum costs. This reduction was achieved by the summary optimisation simplifying the triple patterns in Q_4 and Q_5, resulting in much faster execution times, as well as significantly fewer queries needing to be evaluated for Q_5.

6.3.3 Overall observations. Overall, we see again that the containment optimisation brings limited benefit in reducing the number of rewritten queries generated and that the summary optimisation is more effective. Combining the two optimisations brings for one query a further reduction in the number of queries generated compared to the two separate optimisations. Again we see that for queries that execute quickly, the performance overhead of applying the optimisations is not significant. For slower queries, the summary optimisation is able to give substantial reductions in execution times; the containment optimisation gives modest gains in a few instances, but can also incur a significant performance penalty (e.g. for query Q_4 with maximum cost 3); and combining the two optimisations may or may not improve the times obtained with the summary optimisation alone, dependent on the behaviour of the containment optimisation.

7 CONCLUSIONS

In this paper we have designed and evaluated empirically two optimisation techniques for speeding up the evaluation of SPARQL^{AR} queries. SPARQL^{AR} allows users to query RDF data in a flexible way by evaluating multiple alternative queries which might provide useful answers not returned by the user's original query. However, these additional queries can incur a significant performance overhead, necessitating the deployment of optimisation techniques such as those studied in this paper.

Our approach to flexibly querying RDF data using SPARQL 1.1. centres on *query rewriting*, whereby a set of queries is incrementally generated and evaluated at an increasing 'cost' from the submitted query. This allows a specific rewritten query and specific cost to be associated with each query answer returned to the user, allowing the user to see how each answer has arisen. In our framework, users can select which of the full range of approximation and relaxation operations they wish to be applied to a query, and can set the cost of each such operation, giving users control of the range of edits they wish to be applied to a given query and the priority of application of these edits. Alternatively, users can choose to use the system or application defaults, e.g. applying all operations at a cost of 1.

The conclusions to be drawn from our empirical study are that:

- applying the summary optimisation brings substantial improvements in execution times for longer-running SPARQL^{AR} queries, in some cases by one or more orders of magnitude;
- applying the containment optimisation also brings a substantial improvement for some longer-running queries;
- combining the optimisations can bring further improvements for some queries;
- the summary optimisation does not incur a significant performance penalty, and therefore it should always be applied (subject to the heuristic relating to individual triple patterns identified in Section 5.1);
- the containment optimisation should also generally be applied (after applying the summary optimisation) because in most cases it, too, does not incur a significant performance penalty.

With respect to the last point, the containment optimisation can incur a significant performance penalty (over 5 seconds) when there is a large number of queries to be checked for containment and when few or no queries are discarded after its application. This situation is exemplified by DBpedia query Q_5 at maximum cost 3, YAGO query Q_4 at maximum cost 2 and 3, and YAGO query Q_5 at maximum cost 3. Since it is not possible *a priori* to know how many queries will be discarded by the containment optimisation, our recommendation is therefore that *the containment optimisation should not be applied if the number of queries to be checked is more than an upper limit m*, to be empirically determined for a given implementation environment. For our own prototype implementation, *m* is around 70 queries. However, it can be observed from the performance results presented in Section 6 that there is one case — YAGO query Q_1 — where using this heuristic to not apply the containment optimisation after applying the summary optimisation would incur a performance penalty — of approximately 5% (a run-time of 192s achieved by the summary optimisation alone compared with 183s with the combined optimisation). The relative simplicity of YAGO query Q_1 (just one triple pattern) compared to DBpedia query Q_5 (four triple patterns), and YAGO queries Q_4 and Q_5 (six and seven query patterns, respectively) leads us therefore to a modified version of the heuristic for application of the containment optimisation: *the containment optimisation should not be applied if the number of queries to be checked is more than an upper limit m and the original user query contains more than n triple patterns*. Again, the number *n* would need to be empirically determined for a given implementation, and in our case the results of Section 6 indicate that it lies somewhere between 1 and 3.

We have applied our two optimisations to the full set of queries from [28], listing the results in Appendix B, which provide further corroboration of the above conclusions concerning these two optimisations. Directions of future work include:

(1) further empirical evaluation of our baseline and optimised implementation, with additional queries and additional datasets;
(2) extending our APPROX and RELAX operators to apply to the full property path syntax of SPARQL 1.1;
(3) refining the semantics of APPROX and RELAX to encompass also lexical or semantic similarity measures over literals and resources, so as to allow finer-grained ranking of answers;
(4) combining the APPROX and RELAX operators into one integrated "FLEX" operator, that simultaneously applies approximation and relaxation operations to a property path;
(5) application of rewriting-based query approximation to other graph query languages.

In respect of (4), supporting a FLEX operator would allow greater ease of query formulation for users, since they would not need to be aware of the ontology structure and to identify which conjuncts of their query may be amenable to relaxation and which to approximation. Support of a FLEX operator for CRPQs was explored in [55] but has so far not been investigated in the context of SPARQL. In respect of (5), initial work on rewriting-based query approximation for Cypher is reported in [25].

REFERENCES

[1] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. 2009. Extending SPARQL with regular expression patterns (for querying RDF). *Web Semant.* 7, 2 (April 2009), 57–73. https://doi.org/10.1016/j.websem.2009.02.002
[2] J.M. Almendros-Jimenez, A. Luna, and G. Moreno. 2014. Fuzzy XPath queries in XQuery. In *Proc. OTM 2014.* 457–472.
[3] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. 2004. FleXPath: Flexible structure and full-text querying for XML. In *Proc. ACM SIGMOD 2004.* 83–94.
[4] Renzo Angles and Claudio Gutierrez. 2008. The Expressive Power of SPARQL. In *Proceedings of the 7th International Semantic Web Conference* (Karlsruhe, Germany) *(ISWC '08)*. Springer-Verlag, Berlin, Heidelberg, 114–129. https://doi.org/10.1007/978-3-540-88564-1_8
[5] B. Babcock, S. Chaudhuri, and G. Das. 2003. Dynamic sample selection for approximate query processing. In *Proc. ACM SIGMOD 2003.* 539–550.
[6] Christian Bizer, Richard Cyganiak, and Tom Heath. 2007. How to publish Linked Data on the Web. Web page. http://www4.wiwiss.fu-berlin.de/bizer/pub/LinkedDataTutorial/ Revised 2008. Accessed 22/02/2010.
[7] Till Blume, David Richerby, and Ansgar Scherp. 2021. FLUID: A common model for semantic structural graph summaries based on equivalence relations. *Theoretical Computer Science* 854 (2021), 136–158.
[8] G. Bordogna and G. Psaila. 2008. Customizable flexible querying in classical relational databases. In *Handbook of Research on Fuzzy Information Processing in Databases.* IGI Global, 191–217.

[9] P. Bosc, A. Hadjali, and O. Pivert. 2009. Incremental controlled relaxation of failing flexible queries. *Journal of Intelligent Information Systems* 33, 3 (2009), 261–283.

[10] P. Bosc and O. Pivert. 1992. Some approaches for relational databases flexible querying. *Journal of Intelligent Information Systems* 1, 3 (1992), 323–354.

[11] G. Buratti and D. Montesi. 2008. Ranking for approximated XQuery Full-Text queries. In *Proc. BNCOD 2008*. 165–176.

[12] Andrea Calì, Riccardo Frosini, Alexandra Poulovassilis, and Peter T. Wood. 2014. Flexible Querying for SPARQL. In *On the Move to Meaningful Internet Systems: OTM 2014 Conference Proceedings - Confederated International Conferences: CoopIS, and ODBASE*. 473–490. https://doi.org/10.1007/978-3-662-45563-0_28

[13] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2000. Containment of Conjunctive Regular Path Queries with Inverse. In *Proc. of the 7th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2000)*. 176–185.

[14] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2003. Reasoning on Regular Path Queries. *SIGMOD Rec.* 32, 4 (December 2003), 83–92. https://doi.org/10.1145/959060.959076

[15] Šejla Čebirić, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. 2019. Summarizing semantic graphs: a survey. *The VLDB Journal* 28, 3 (01 June 2019), 295–327.

[16] J. P. Cedeno and K. S. Candan. 2011. R2DF framework for ranked path queries over weighted RDF graphs. In *Proc. WIMS 2011*. 1–12.

[17] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. 2001. Approximate query processing using wavelets. *The VLDB Journal* 10, 2-3 (2001), 199–223.

[18] W. W. Chu, H. Yang, K. Chiang, M. Minock, G. Chow, and C. Larson. 1996. CoBase: A Scalable and Extensible Cooperative Information System. *Journal of Intelligent Information Systems* 6, 2/3 (1996), 223–259.

[19] Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. 2013. A similarity measure for approximate querying over RDF data. In *Proc. EDBT/ICDT 2013 Workshops*. 205–213.

[20] Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. 2015. A Unified Framework for Flexible Query answering over Heterogeneous Data Sources. In *Proc. FQAS 2015*. 283–294.

[21] P. Dolog, H. Stuckenschmidt, H. Wache, and J. Diederich. 2009. Relaxing RDF queries based on user and domain preferences. *J. Intelligent Information Systems* 33, 3 (2009), 239–260.

[22] S. Elbassuoni, M. Ramanath, and G. Weikum. 2011. Query relaxation for entity-relationship search. In *Proc. ESWC 2011 (Part 2)*. 62–76.

[23] Diego Figueira, Adwait Godbole, S. Krishna, Wim Martens, Matthias Niewerth, and Tina Trautner. 2020. Containment of Simple Conjunctive Regular Path Queries. In *Proc. 17th International Conference on Principles of Knowledge Representation and Reasoning*. 371–380.

[24] R. Fink and D. Olteanu. 2011. On the optimal approximation of queries using tractable propositional languages. In *Proc. ICDT 2011*. 174–185.

[25] George Fletcher, Alexandra Poulovassilis, Petra Selmer, and Peter T. Wood. 2019. Approximate Querying for the Property Graph Language Cypher. In *Proc. 2019 IEEE International Conference on Big Data*. IEEE, 617–622.

[26] Daniela Florescu, Alon Levy, and Dan Suciu. 1998. Query Containment for Conjunctive Queries with Regular Expressions. In *Proc. Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Washington, USA) *(PODS '98)*. ACM, New York, NY, USA, 139–148. https://doi.org/10.1145/275487.275503

[27] Riccardo Frosini. 2017. Flexible Query Processing of SPARQL Queries. PhD Thesis.

[28] Riccardo Frosini, Andrea Calì, Alexandra Poulovassilis, and Peter T. Wood. 2017. Flexible query processing for SPARQL. *Semantic Web* 8, 4 (2017), 533–563.

[29] J. Galindo, J.M. Medina, O. Pons, and C. Cubero. 1998. A Server for Fuzzy SQL Queries. In *Proc. FQAS 1998*. 164–174.

[30] Gösta Grahne and Alex Thomo. 2006. Regular path queries under approximate semantics. *Ann. Math. Artif. Intell.* 46, 1-2 (2006), 165–190.

[31] S. Harris and A. Seaborne. 2013. SPARQL 1.1 Query Language. W3C Recommendation.

[32] J. Heer, M. Agrawala, and M. Willett. 2008. Generalized selection via interactive query relaxation. In *Proc. CHI 2008*. 959–968.

[33] J. Hill, J. Torson, B. Guo, and Z. Chen. 2010. Toward ontology-guided knowledge-driven XML query relaxation. In *Proc. 2nd Int. Conf. on Computational Intelligence, Modelling and Simulation (CIMSiM) 2010*. 448–453.

[34] A. Hogan, M. Mellotte, G. Powell, and D. Stampouli. 2012. Towards fuzzy query relaxation for RDF. In *Proc. ISWC 2012*. 687–702.

[35] H. Huang and C. Liu. 2010. Query relaxation for star queries on RDF. In *Proc. WISE 2010*. 376–389.

[36] H. Huang, C. Liu, and X. Zhou. 2008. Computing relaxed answers on RDF databases. In *Proc. WISE 2008*. 163–175.

[37] Carlos A. Hurtado, Alexandra Poulovassilis, and Peter T. Wood. 2008. Query relaxation in RDF. *Journal on Data Semantics* X (2008), 31–61.

[38] Carlos A. Hurtado, Alexandra Poulovassilis, and Peter T. Wood. 2009. Ranking Approximate Answers to Semantic Web Queries. In *Proc. ESWC 2009*. 263–277.

[39] Y. Ioannidis and V. Poosala. 1999. Histogram-based approximation of set-valued query-answers. In *Proc. VLDB 1999*. 174–185.

[40] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. 2002. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proc. 18th International Conference on Data Engineering*. 129–140.

[41] Kenza Kellou-Menouer, Nikolaos Kardoulakis, Georgia Troullinou, Zoubida Kedad, Dimitris Plexousakis, and Haridimos Kondylakis. 2021. A survey on semantic schema discovery. *The VLDB Journal* (2021), 1–36.
[42] Christoph Kiefer, Abraham Bernstein, and Markus Stocker. 2007. The Fundamentals of iSPARQL: A Virtual Triple Approach for Similarity-Based Semantic Web Tasks. In *Proc. ISWC 2007.* 295–309.
[43] Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoč. 2015. SPARQL with Property Paths. In *Proc. 14th Int. Semantic Web Conf. (Lecture Notes in Computer Science, Vol. 9366)*, Marcelo Arenas et al. (Eds.). Springer, 3–18.
[44] C. Liu, J. Li, J.X. Yu, and R. Zhou. 2010. Adaptive Relaxation for Querying Heterogeneous XML Data Sources. *Information Systems* 35, 6 (2010), 688–707.
[45] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. 2018. Graph Summarization Methods and Applications: A Survey. *ACM Comput. Surv.* 51, 3, Article 62 (June 2018), 34 pages. https://doi.org/10.1145/3186727
[46] Theofilos Mailis, Yannis Kotidis, Vaggelis Nikolopoulos, Evgeny Kharlamov, Ian Horrocks, and Yannis Ioannidis. 2019. An Efficient Index for RDF Query Containment. In *Proc. 2019 International Conference on Management of Data.* Association for Computing Machinery, New York, NY, USA, 1499–1516.
[47] F. Mandreoli, R. Martoglia, G. Villani, and W. Penzo. 2009. Flexible Query Answering on Graph-Modeled Data. In *Proc. EDBT 2009.* 216–227.
[48] X. Meng, Z. M. Ma, and L. Yan. 2008. Providing flexible queries over web databases. In *Knowledge-Based Intelligent Information and Engineering Systems.* 601–606.
[49] Mehryar Mohri, Pedro Moreno, and Eugene Weinstein. 2007. Factor Automata of Automata and Applications. In *Implementation and Application of Automata*, Jan Holub and Jan Žďárek (Eds.). Springer, Berlin, Heidelberg, 168–179.
[50] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. 2007. Minimal Deductive Systems for RDF. In *Proceedings of the 4th European Conference on The Semantic Web: Research and Applications* (Innsbruck, Austria) *(ESWC '07)*. Springer-Verlag, Berlin, Heidelberg, 53–67. https://doi.org/10.1007/978-3-540-72667-8_6
[51] S. Na and S. Park. 1996. A Process of Fuzzy Query on new Fuzzy Object Oriented Data Model. In *Proc. DEXA 1996.* 500–509.
[52] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3, Article 16 (Sept. 2009), 45 pages. https://doi.org/10.1145/1567274.1567278
[53] Reinhard Pichler and Sebastian Skritek. 2014. Containment and Equivalence of Well-Designed SPARQL. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems.* 39–50.
[54] Olivier Pivert, Olfa Slama, and Virginie Thion. 2016. SPARQL extensions with preferences: a survey. In *Proc. 31st Symposium on Applied Computing.* 1015–1020.
[55] Alexandra Poulovassilis, Petra Selmer, and Peter T. Wood. 2016. Approximation and Relaxation of Semantic Web Path Queries. *Journal of Web Semantics* 40 (2016), 1–21.
[56] Alexandra Poulovassilis and Peter T. Wood. 2010. Combining Approximation and Relaxation in Semantic Web Path Queries. In *Proc. ISWC 2010.* 631–646.
[57] B. R. K. Reddy and P. S. Kumar. 2010. Efficient approximate SPARQL querying of web of linked data. In *Proc. URSW 2010.* 37–48.
[58] Michael Schmidt. 2009. *Foundations of SPARQL Query Optimization.* Ph. D. Dissertation. Albert-Ludwigs-Universitat Freiburg. http://www.informatik.uni-freiburg.de/~mschmidt/docs/diss_final01122010.pdf
[59] M. Theobald, R. Schenkel, and G. Weikum. 2005. An efficient and versatile query engine for TopX search. In *Proc. VLDB 2005.* 625–636.
[60] S. Yang, Y. Wu, and X. Sun, H. Yan. 2014. Schemaless and Structureless Graph Querying. *Proc. VLDB Endowment* 7, 7 (2014), 565–576.
[61] Weiguo Zheng, Lei Zou, Wei Peng, Xifeng Yan, Shaoxu Song, and Dongyan Zhao. 2016. Semantic SPARQL similarity search over RDF knowledge graphs. *Proc. VLDB 2016* 9, 11 (2016), 840–851.
[62] X. Zhou, J. Gaugaz, W.-T. Balke, and W. Nejdl. 2007. Query relaxation using malleable schemas. In *Proc. ACM SIGMOD 2007.* 545–556.

A REWRITING ALGORITHM

The query rewriting algorithm (Algorithm 6) takes as input the SPARQL^{AR} query to be evaluated, Q_{AR}; the maximum approximation/relaxation cost of the rewritten queries, c; and the ontology K. It starts by creating the query Q_0, which returns the exact answers to the user's query Q, i.e. ignoring any occurrences of the APPROX and RELAX operators in Q. To keep track of which triple patterns need to be relaxed or approximated, we label such triple patterns within Q_0 and within subsequently generated queries with A for approximation and R for relaxation.

In Algorithm 6, the function *toCQS* ("to conjunctive query set") takes as input the query Q_0 and returns a set of pairs $\langle Q_i, 0\rangle$ such that $\bigcup_i [\![Q_i]\!]_G = [\![Q_0]\!]_G$ and no Q_i contains the UNION operator. The function *toCQS* exploits the following equalities:

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

The set of queries returned by $toCQS(Q_0)$ is assigned to the variable *oldGeneration* . For each query Q in *oldGeneration*, each triple pattern $\langle x, P, y\rangle$ in Q labelled with A (R), and each URI p appearing in P, we apply one step of approximation (relaxation) to p, and assign the cost of applying that approximation (relaxation) to the resulting query. The *applyApprox* and *applyRelax* functions invoked by Algorithm 6 are shown as Algorithms 7 and 9, respectively. We note that Algorithm 9 is defined more efficiently than the version listed in [28], although their effects are the same. The *addTo* function invoked by Algorithm 6 takes two arguments: a collection C of query/cost pairs and a single query/cost pair $\langle Q, c\rangle$. It adds $\langle Q, c\rangle$ to C if C does not already contain Q. If C already contains a pair $\langle Q, c'\rangle$ such that $c' > c$, then $\langle Q, c'\rangle$ is replaced by $\langle Q, c\rangle$ in C i.e. the smallest cost associated with Q' is retained. We continue to generate queries iteratively up to the maximum query cost c.

ALGORITHM 6: Rewriting algorithm

```
input  : Query $Q_{AR}$; approx/relax max cost $c$; Ontology $K$.
output : List of ⟨query, cost⟩ pairs, sorted by increasing cost.
$Q_0$ := remove the APPROX and RELAX operators from $Q_{AR}$, and label the approximated/relaxed triple patterns by $A/R$;
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
                    addTo(newGeneration, $\langle Q', cost + cost'\rangle$);
                    addTo(queries, $\langle Q', cost + cost'\rangle$);
                    /* The elements of newGeneration and queries are ordered by increasing cost. */
                end
            end
        end
    end
    oldGeneration := newGeneration;
end
return queries;
```

The *applyApprox* (Algorithm 7) and *applyRelax* (Algorithm 9) functions invoke the functions *approxRegex* (Algorithm 8) and *relaxTriplePattern* (Algorithm 10), respectively. Algorithm 8 applies one step of approximation to a regular expression P. If P is a URL p, then deletion is represented by ϵ, substitution by $!p$ (since p has already

ALGORITHM 7: applyApprox

```
input   : Query $Q$; triple pattern $\langle x, P, y\rangle_A$.
output  : Set $S$ of $\langle$query, cost$\rangle$ 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 $Q$;
    $S := S \cup \{\langle Q', cost\rangle\}$;
end
return S;
```

ALGORITHM 8: approxRegex

```
input   : Regular expression $P$.
output  : Set $T$ of $\langle$regular expression, cost$\rangle$ pairs.
$T := \emptyset$;
if $P = \epsilon$ then return $T$ ;
else if $P =$ _ then $T := T \cup \{\langle \_/\_, c_i\rangle\}$ ;
else if $P = !p$ then $T := T \cup \{\langle \_/!p, c_i\rangle, \langle !p/\_, c_i\rangle\}$ ;
else if $P = p$ where $p$ is a URI then $T := T \cup \{\langle \epsilon, c_d\rangle, \langle !p, c_s\rangle, \langle \_/p, c_i\rangle, \langle p/\_, c_i\rangle\}$ ;
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 $T := T \cup \{\langle P_1/P', cost\rangle\}$ ;
end
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 $T := T \cup \{\langle P', cost\rangle\}$ ;
end
else if $P = P_1^*$ then
    foreach $\langle P', cost\rangle \in approxRegex(P_1)$ do $T := T \cup \{\langle (P_1^*)/P'/(P_1^*), cost\rangle\}$ ;
end
return T;
```

ALGORITHM 9: applyRelax

```
input   : Query $Q$; triple pattern $\langle x, P, y\rangle_R$ of $Q$; Ontology $K$.
output  : Set $S$ of $\langle$query, cost$\rangle$ 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 $Q$;
    $S := S \cup \{\langle Q', cost\rangle\}$;
end
return S;
```

appeared, at lower cost), and insertion by _. None of these three terms can be used in the original SPARQL^{AR} query submitted by the user. Hence if they appear in the regular expression being approximated, it must be as the result of an edit operation applied earlier. This means that it is not necessary to apply any further operations

ALGORITHM 10: relaxTriplePattern

```
input  : Triple pattern $\langle x, P, y\rangle$; Ontology $K$.
output : Set $T$ of $\langle$triple pattern, cost$\rangle$ pairs.
$T := \emptyset$;
if $P = p$ where $p$ is a URI then
    foreach $p'$ such that $\exists(p, sp, p') \in E_K$ do $T := T \cup \{\langle\langle x, p', y\rangle, c_{rule_2}\rangle\}$ ;
    foreach $b$ such that $\exists(a, sc, b) \in E_K$ and $p = type$ and $y = a$ do $T := T \cup \{\langle\langle x, type, b\rangle, c_{rule_4}\rangle\}$ ;
    foreach $b$ such that $\exists(a, sc, b) \in E_K$ and $p = type^-$ and $x = a$ do $T := T \cup \{\langle\langle b, type^-, y\rangle, c_{rule_4}\rangle\}$ ;
    foreach $a$ such that $\exists(p, dom, a) \in E_K$ and $y$ is a URI or a Literal do $T := T \cup \{\langle\langle x, type, a\rangle, c_{rule_5}\rangle\}$ ;
    foreach $a$ such that $\exists(p, range, a) \in E_K$ and $x$ is a URI do $T := T \cup \{\langle\langle a, type^-, y\rangle, c_{rule_6}\rangle\}$ ;
end
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 $T := T \cup \{\langle\langle x, P_1/P', y'\rangle, cost\rangle\}$ ;
end
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 $T := T \cup \{\langle\langle x', P', y'\rangle, cost\rangle\}$ ;
end
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 $T := T \cup \{\langle\langle x, P_1^*/P', y'\rangle, cost\rangle\}$ ;
end
return T;
```

to ϵ, and only insertions need be applied to to _ and $!p$, as indicated by the second, third and fourth lines in the algorithm. For detailed explanation of Algorithm 10, please see [28].

B TIMINGS FOR THE TEN YAGO QUERIES OF [28]

Each of the ten queries Q_i from [28] has i triple patterns. The first four queries each have RELAX applied to only a single triple pattern. Hence not many rewritten queries are generated, opportunities for optimisation are limited, and their execution times are fast.

Queries Q_5, Q_6 and Q_{10} each have RELAX applied to one triple pattern and APPROX applied to two. Running the exact form of query Q_5 returns 2,943,311 answers and takes over 17 hours. In Q_5 both triple patterns with APPROX applied to them also use Kleene closure in their property paths. As a result, the set of rewritten queries at each maximum cost from 1 to 3 are not able to complete execution within 24 hours. On the other hand, we see that the summary optimisation in particular is able to reduce the execution times considerably for queries Q_6 to Q_{10} at all maximum costs.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their insightful comments which helped to improve the paper.

Max Cost	Optimisation	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}
1	none	2	3	3	2	13	16	8	6	6	10
1	containment	2	3	3	2	11	16	8	6	4	10
1	summary	2	2	2	1	13	16	5	4	3	8
1	combined	2	2	2	1	11	16	5	4	3	8
2	none	2	5	5	2	85	119	30	16	29	46
2	containment	2	5	5	2	60	119	30	16	12	46
2	summary	2	4	4	1	85	117	14	10	6	35
2	combined	2	4	4	1	60	117	14	10	6	35
3	none	2	5	5	2	389	560	74	30	96	138
3	containment	2	5	5	2	232	560	74	30	23	138
3	summary	2	4	4	1	389	542	28	18	8	108
3	combined	2	4	4	1	232	542	28	18	8	108

Table 16. Number of queries generated, with and without various optimisations.

Max Cost	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}
0	13618	1180	930	8405	2943311	32	1450	11557	0	0
1	13619	260062	116521	8405	N/A	234	54023	12917	0	0
2	13619	260062	116521	8405	N/A	25199	73311	13092	0	0
3	13619	260062	116521	8405	N/A	33316	455181	13092	0	0

Table 17. Number of answers returned by the queries.

Max Cost	Optimisation	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}
1	none	0.031	0.033	0.034	0.010	N/A	2.44	214.55	1428.1	1913.0	2215.0
1	containment	0.031	0.033	0.034	0.010	N/A	2.56	214.78	1428.4	1038.0	2215.0
1	summary	0.031	0.026	0.030	0.007	N/A	2.49	13.29	191.3	90.2	67.5
1	combined	0.031	0.026	0.030	0.007	N/A	2.78	13.66	191.5	90.4	68.0
2	none	0.031	0.041	0.041	0.010	N/A	122.77	602.11	2105.0	4110.0	3603.0
2	containment	0.031	0.041	0.041	0.010	N/A	129.21	605.71	2106.0	2549.0	3605.0
2	summary	0.031	0.039	0.040	0.007	N/A	22.87	19.32	281.1	104.4	89.0
2	combined	0.031	0.039	0.040	0.007	N/A	26.83	22.74	281.6	104.9	91.2
3	none	0.031	0.041	0.041	0.010	N/A	648.29	1301.54	3267.0	7772.0	5630.0
3	containment	0.031	0.041	0.041	0.010	N/A	701.19	1307.41	3270.0	3105.0	5632.0
3	summary	0.031	0.039	0.040	0.007	N/A	118.91	121.02	453.4	122.5	106.6
3	combined	0.031	0.039	0.040	0.007	N/A	154.10	129.51	455.5	123.3	107.9

Table 18. Execution times (seconds) for the queries, using various optimisations.