QaRS: A User-Friendly Graphical Tool for Semantic Query Design and Relaxation

Géraud Fokou
LIAS/ISAE-ENSMA
University of Poitiers
Futuroscope Cedex - France
geraud.fokou@ensma.fr

Stéphane Jean
LIAS/ISAE-ENSMA
University of Poitiers
Futuroscope Cedex - France
stephane.jean@ensma.fr

Allel Hadjali
LIAS/ISAE-ENSMA
University of Poitiers
Futuroscope Cedex - France
allel.hadjali@ensma.fr

Mickaël Baron
LIAS/ISAE-ENSMA
University of Poitiers
Futuroscope Cedex - France
mickael.baron@ensma.fr

ABSTRACT

This paper presents a *Query-and-Relax System* (QaRS) designed to facilitate the exploitation of large knowledge bases. QaRS proposes a graphical interface to construct a SPARQL query and use different cooperative answering techniques. The proposed cooperative techniques help users in finding alternative answers when their queries fail or do not return the expected number of answers. The present demonstration includes three main relaxation strategies: (1)- *automatic* where the system automatically relaxes the query based on similarity measures, (2)- *manual* where the user can specify the conditions that can or cannot be relaxed as well as the tolerance values and (3)- *interactive* where QaRS computes the causes of the query failure as a set of Minimal Failing Subqueries (MFSs) and then the user chooses the relaxation operators according to these MFSs.

General Terms

Algorithms, Design, Experimentation

Keywords

SPARQL, Relaxation, Minimal Failing Subquery, Similarity

1. INTRODUCTION

In recent years, several large Knowledge Bases (KBs) have been created such as YAGO [5] or Knowledge Vault [1] which contain millions of entities and facts about them. Such information is usually stored in RDF format and queried with the SPARQL language. Large KBs are difficult to use as (1)- their schema (often called ontology) and its underlying semantics are rarely understood by end users and (2)- RDF can be used to represent data ranging from unstructured to structured data leading to more or less sparse data [2]. A common issue encountered by users is the problem of failing queries, i.e., query results are empty or do not contain the number of expected answers.

As an example, let us consider the ontology inspired by the LUBM Benchmark depicted in Figure 1. If a user wants to find the professors who are assisted by one of her/his *phD* student in an *UnderGraduateCourse*, (s)he may write the query:

```
 SELECT ?X ?Y ?Z
WHERE { ?Z ub:teacherOf ?Y.
    ?Y rdf:type ub:UnderGraduateCourse.
    ?X ub:teachingAssistantOf ?Y.
    ?Z ub:advisorOf ?X.
    ?X rdf:type ub:AssistantProfessor. }
```

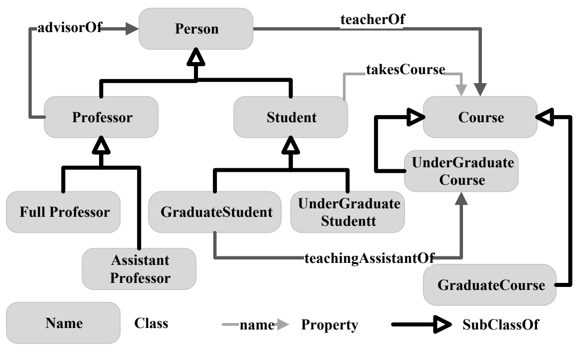


Figure 1: Ontology Example

In this query, the user makes the false assumption that a teaching assistant of a course is an *AssistantProfessor* (instead of a *GraduateStudent*). With a deeper knowledge of the ontology, the user could have known that the *teachingAssistantOf* property has the *GraduateStudent* class as domain and thus his/her query can not return any result.

Moreover, even if the query was written without any misconception, the query could still have failed if it is too restrictive or if the target KB is incomplete. To solve these problems, several works have proposed relaxation techniques for SPARQL queries (e.g., [3, 7]). But none of them proposes a simple and intuitive graphical system to build a SPARQL query and relax it with or without the help of the user. Conversely, several works have proposed graphical systems to query large knowledge bases (e.g., [8, 9]). But they do not support any cooperative query answering technique needed to return alternative answers to failing queries. The QaRS system described in this paper aims at filling this gap. First, the functionalities of this system and its architecture are discussed. Then, the demonstration scenarios that we intend to show the audience are presented.

2. QARS'S FUNCTIONALITIES

The system we propose has three main functionalities, (i) a visual assistant for designing consistent queries, (ii) a visual help for queries relaxation and (iii) an explanation of both queries's failure and relaxation process.

2.1 Graphical query design

The ontology browser panel displays the ontology as a graph (see Figure 2). As an ontology can be large, a search box with auto completion feature is available for finding classes and properties that will be used in the query. When a class or property is selected, the graph is centered around this concept to see all the related concepts.

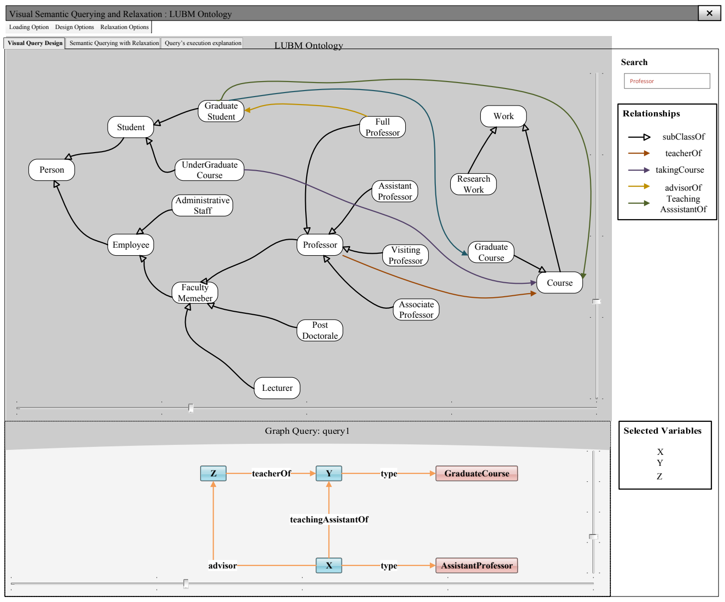


Figure 2: QaRS Query Design Panel

The visual construction of a query is composed of three main steps:
1) dragging and dropping classes and/or properties from the ontology browser into the query panel. Dropping a class C in this panel creates a graph corresponding to the triple ($?v_i$ $rdf{:}type$ C) where v_i is a variable which has not been previously used in the query. In the same way, dropping a property P creates a graph showing the triple ($?v_i$ P $?v_j$).
2) Linking the triple patterns defined in the previous step by identifying the variables that they share. This action is done by dragging a variable and dropping it into an other variable. It indicates that the two variables are the same. In this step, QaRS checks whether the query can return a result by testing its consistency w.r.t. the domain and range of the properties. In our running example, since we know that X is a teaching assistant of the course Y, according to the ontology, one can deduce that X can not be an $AssistantProfessor$. So the last triple of this query leads to an inconsistency. As another example, the query Q:

```
SELECT ?X ?Y WHERE {
  ?X rdf:type ub:GraduateStudent.
  ?X ub:teachingAssistantOf ?Y.
  ?Y rdf:type ub:GraduateCourse. }
```

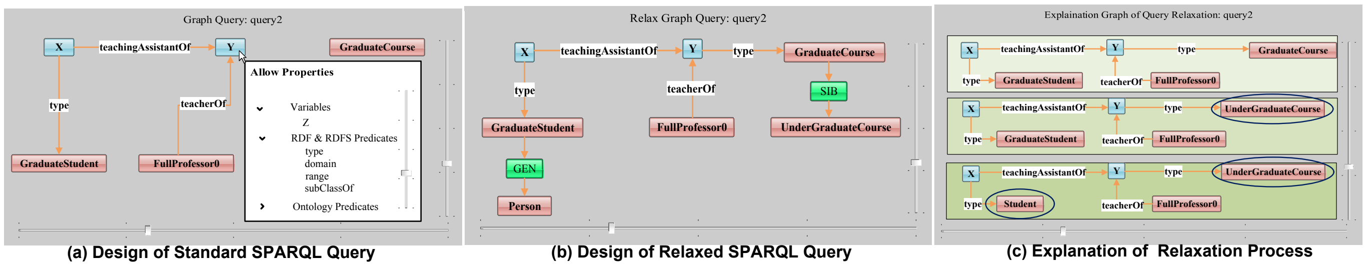
is consistent when it has only the first two triple patterns as depicted in Figure 3-(a) (*there are graduate students who are teaching assistants*). But when the last triple pattern is added, the query becomes inconsistent, i.e. it can not return any answer (*graduate students can not be teaching assistants of a graduate course*). In this case, QaRS alerts the users.

3) adding FILTER, OPTIONAL and/or UNION operators by selecting the components of the query on which they must be applied (a variable, a triple or a set of triples), right clicking on them and choosing the corresponding operators. Each one of these operators is graphically identified in the query by a specific color or component. The SPARQL query corresponding to the graphical query can also be displayed and the user can interact both with the textual or graphical query to edit it. Modifying the graphical query automatically changes the textual query and vis-versa.

2.2 Query relaxation strategies

Automatic relaxation. When executing the query designed in the previous step, the user can specify the minimum number, say k, of expected answers. If the query result does not have k answers, QaRS considers it as a failing query and automatically relaxes it. Basically, this automatic relaxation process consists in computing a set of possible relaxed queries (see further) from the initial query, ordering them according to their similarities with the initial query and executing them following this order until the number of expected results is reached. If the result of a relaxed query is large, the answers are ordered according to their satisfaction degrees w.r.t. the initial query and then the user is provided with the $top-k$ answers.

Manual relaxation. Users may have some constraints on the parts of the query that can be relaxed as well as on the tolerance values that are acceptable. They can manually specify these constraints in the design query panel. Three kind of constraints can be graphically defined by users. (i) Triple patterns that must not be relaxed. The user simply selects a subset of the query graph that must not be relaxed, right clicks on it and selects the corresponding option. (ii) Allowed classes (resp. properties) in the hierarchy of a class (resp. property) that can be used to relax the query. The user selects a class or property, right clicks on it and selects the relaxation option. The hierarchy of the class or property is then displayed and the user can select the classes or properties that can be used in the relaxation process (see Figure 3-(b)). In this step, QaRS checks that the selected classes or properties lead to a consistent relaxed query. (iii) Allowed values to relax filters. In a similar way as above, the user selects a filter that can be relaxed, right clicks on it and



(a) Design of Standard SPARQL Query (b) Design of Relaxed SPARQL Query (c) Explanation of Relaxation Process

Figure 3: User interface of the System

selects the relaxation option. According to the datatype of the filter, a panel allows the user to define the tolerance values. Thanks to methods borrowed from fuzzy logic theory, one can obtain satisfaction degrees of the relaxed value of the filter. Finally, the user specifies the minimum number of expected results and executes the query. If the query fails, QaRS triggers the relaxation process while respecting his/her constraints (see Figure 3-(c)).

Interactive relaxation. In the previous scenario, the user does not know the causes of the failure of his/her query. QaRS can provide him/here with explanation. This explanation consists in displaying a set of Minimal Failing Subqueries (MFSs) [4] of the query. For the query Q (section 2.1), the cause of its failure is the subquery below.

```
 SELECT ?X ?Y
WHERE { ?X ub:teachingAssistantOf ?Y.
                ?Y rdf:type ub:GraduateCourse. }
```

Each MFS (i) is a failing subquery of the initial query and (ii) does not include a failing subquery. Thus, if the MFSs of a query are not relaxed, the initial query will never return non-empty answers. In this scenario, the relaxation process is a two-step procedure: (i) QaRS displays the set of MFSs of the query; (ii) the user can automatically or manually relax each MFS like in the previous scenarios. By default, QaRS proposes to make optional the triple patterns of the MFSs.

2.3 Explanation and customization

Similarity is a key notion in QaRS. It is leveraged by the system, on the one hand, for measuring the similarity between the initial query and the relaxed ones and, on the other hand, for computing the satisfaction degrees of alternative answers returned w.r.t the initial query. The latter point allows to provide user with discriminated set of answers and then (s)he can select the *top-k* answers (where k is the minimum number of expected answers). As for query similarity, it helps users to rank-order the relaxed queries and choose an appropriate set of queries to be executed to obtain k answers. These executed queries can be seen as an explanation for the user to (progressively) reach the desired answers. As different similarity measures can be used to compute the similarity between two classes (or properties) of an ontology, relaxation performed by QaRS can be customized by selecting measures that fit best the user's needs.

3. SYSTEM ARCHITECTURE

The architecture of QaRS is illustrated in Figure 4. It comprises two main parts. The first part includes two components: *Graphical Design Of* SPARQL *Query* (GDSQ) and SPARQL *Query Analyzer* (SQLA). While the second part, which is related to the core of query relaxation, is composed of four modules: *Relaxation Operator Interpretor*, *Automatic Query Relaxation*, MFS *Engine for* SPARQL *Query* and *Ranking Alternative Answers Engine*. The module *Extended* SPARQL *Query Engine* is an extension of standard SPARQL query engine which allows us to launch the relaxation process when the query at hand fails. This module also makes easy the integration of QaRS in any triplestore environment. Now, we provide details about each component of QaRS.

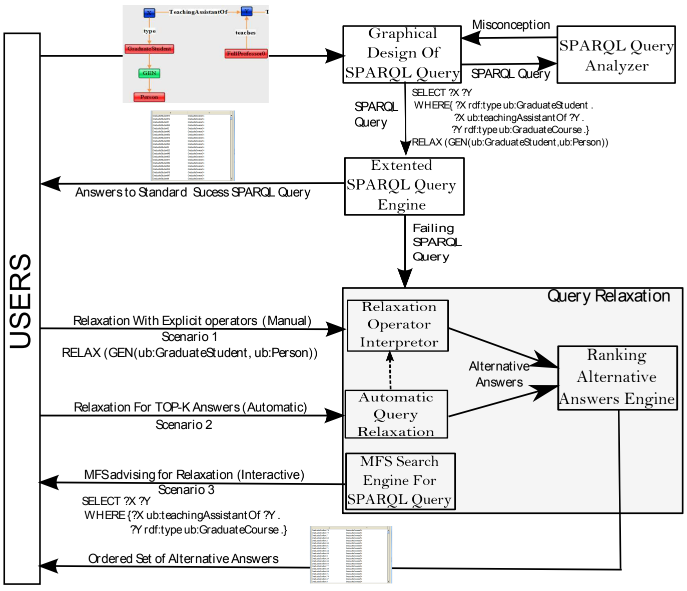

USERS
TeachingAssistantOf
type
teaches
GraduateStudent
FullProfessor0
GEN
Person
Graphical Design Of SPARQL Query
Misconception
SPARQL Query
SPARQL Query Analyzer
SPARQL Query
SELECT ?X ?Y
WHERE{ ?X rdf:type ub:GraduateStudent .
?X ub:teachingAssistantOf ?Y .
?Y rdf:type ub:GraduateCourse .}
RELAX (GEN(ub:GraduateStudent,ub:Person))
Extented SPARQL Query Engine
Answers to Standard Sucess SPARQL Query
Failing SPARQL Query
Query Relaxation
Relaxation Operator Interpretor
Relaxation With Explicit operators (Manual)
Scenario 1
RELAX (GEN(ub:GraduateStudent, ub:Person))
Alternative Answers
Ranking Alternative Answers Engine
Automatic Query Relaxation
Relaxation For TOP-K Answers (Automatic)
Scenario 2
MFS Search Engine For SPARQL Query
MFS advising for Relaxation (Interactive)
Scenario 3
SELECT ?X ?Y
WHERE {?X ub:teachingAssistantOf ?Y .
?Y rdf:type ub:GraduateCourse .}
Ordered Set of Alternative Answers

Figure 4: Architecture of QaRS

GDSQ and SQA modules

GDSQ offers a user-friendly interface to assist users in the design and building of their SPARQL queries in a graphical and intuitive way. As for SQA module, which is an online analyzer, it checks on-the-fly the syntax and the consistency of the designed query. It also proposes auto completion and suggests concepts for designing the query.

Relaxation interpretor

This module interprets each of the three relaxation operators studied in [3] and generates the corresponding set of relaxed SPARQL queries. *GEN* operator takes as input a concept C_i and a super concept C_f. The system generates SPARQL queries with C_i replaced by the classes in the path from C_i

to C_f in the ontology. As for SIB operator, the system generates SPARQL queries where a concept C_0 is replaced by its sibling classes $C_1, C_2, ..., C_m$. In the case of $PRED$ operator, it incrementally relaxes filters involving simple data types (numeric, string, etc). This module is launched when the clause $RELAX$ is used in the designed SPARQL query.

Automatic query relaxation

This module is called when users want to have the *top-k* answers without setting the relaxation operators to use. QaRS generates all the relaxed queries using the three previous operators and their combinations, in the spirit of the approach proposed by [6]. A rank-ordering of these queries is established according to their similarity w.r.t. the failing query. The following similarity measure between classes is used [3]:

$$Sim(C_i, C_f) = \frac{IC(msca(C_i, C_f))}{IC(C_i) + IC(C_f) - IC(msca(C_i, C_f))}$$

Then, the relaxed queries are executed from the most similar to the least similar and the answers obtained are sent to the *Ranking alternative answers engine*.

Ranking alternative answers engine

QaRS provides the user with a set of alternative answers in a discriminated way. Each answer h_i is associated with a satisfaction score computed as follows [3]:

$SatQ(h_i) = \min(Sim(Q', Q), SatQ'(h_i))$

where $Sim(Q, Q')$ stands for the similarity measure between the initial query Q and its relaxed form Q' and $SatQ'(h_i)$ for a satisfaction degree of h_i w.r.t. Q'. This latter degree is obtained thanks to the formula [3]:

$SatQ'(h_i) = min(\max_{t \in type(h_i)} Sim(t, c'), \mu_p(h_i.propRelax))$

where c' is the relaxed class which gives the answer h_i and μ_p is the membership function of the (fuzzy) property *propRelax*.

MFS search engine

The MFS search engine is a module which identifies the causes of query failure. To do so, a set of MFSs of the failing query are computed. MFSs provide user with a clear explanation on the empty answer problem. First, we transform the target SPARQL query into a set of triple patterns to form a conjunctive query. Next, an MFS of the conjunctive query is computed. To find the other MFSs, a set of significant subqueries (SSQs) is calculated. Each SSQ is characterized by three properties: (i) it does not contain the MFSs found; (ii) it is not included in those MFSs and, (iii) it does not include any other SSQ. The above two step-procedure is executed recursively on each element of the set of SSQs. All MFSs produced by this procedure are shown to the user as an explanation about his/her query failure.

4. DEMO SCENARIOS

We run two scenarios on LUBM ontology data. The first scenario aims at relaxing a failing query Q manually. The second one shows the interest of the MFSs as explanation of the query failure and their use for an efficient relaxation. To run the above scenarios, we use Jena triplestore to load the generated LUBM's data.

Scenario 1

The user wants to find all *"the graduate students who are teaching assistants of a graduate course"*. The SPARQL query for this request is given in section 2.1. To obtain non empty answers, the user can ask a generalization (resp. substitution) of *GraduateStudent* (resp. *GraduateCourse*) concept to *Person* (resp. with *UnderGraduateCourse*) concept. This can be done graphically as shown in figure 3-(b). The relaxation operators proposed by QaRS depend of the query's concept to relax and ontology. As it can be seen in the ontology of Figure 1, the relaxed query may result in non empty answers since *GraduateStudent* may be teaching assistants of *UnderGraduateCourse* which is a sibling *Class* of *GraduateCourse*. It is worthing to note that this kind of relaxation does not always guarantee the success of the relaxation process. It is the case for the generalization of *GraduateStudent* to *Person* (where there is none subclass of *Person* with *teachingAssistantOf* as property, except the subclass *GraduateStudent*).

Scenario 2

To avoid the main flaw of the above scenario, we first identify the causes (i.e., MFSs) of query failure, then we apply appropriate relaxations on the triple patterns involved in the MFSs of the query. For our running example of section 2.1, which contains one MFS (see section 2.2), QaRS identifies it and shows this MFS graphically to the user. Then, the system suggests appropriate operators for relaxing the MFS. In our case, GEN or SIB operator will be proposed to relax the triple (?Y *rdf*:*type* *ub*:*GraduateCourse*) included in the MFS. By this way, alternative answers that fit best the user's needs are returned by the system.

5. REFERENCES

[1] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge Vault: A Web-scale Approach to Probabilistic Knowledge Fusion. In *ACM SIGKDD, (KDD '14)*, pages 601–610, 2014.

[2] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets. In *SIGMOD '11*, pages 145–156, 2011.

[3] G. Fokou, S. Jean, and A. Hadjali. Endowing Semantic Query Languages with Advanced Relaxation Capabilities. In *ISMIS'14*, pages 512–517, 2014.

[4] P. Godfrey. Minimization in Cooperative Response to Failing Database Queries. *International Journal of Cooperative Information Systems*, 6(2):95–149, 1997.

[5] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194:28–61, 2013.

[6] H. Huang, C. Liu, and X. Zhou. Approximating Query Answering on RDF Databases. *World Wide Web*, 15(1):89–114, 2012.

[7] C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Query Relaxation in RDF. *Journal on data semantics X*, pages 31–61, 2008.

[8] N. Jayaram, M. Gupta, A. Khan, C. Li, X. Yan, and R. Elmasri. GQBE: Querying knowledge graphs by example entity tuples. In *IEEE ICDE'14*, pages 1250–1253, 2014.

[9] A. Russell and P. R. Smart. NITELIGHT: A Graphical Editor for SPARQL Queries. In *Proceedings of the Poster and Demonstration Session at ISWC'08*, 2008.