The final publication is available at Springer via
http://dx.doi.org/10.1007/978-3-319-34129-3_27

RDF Query Relaxation Strategies Based on Failure Causes

Géraud Fokou, Stéphane Jean, Allel Hadjali, Mickaël Baron

LIAS/ISAE-ENSMA - University of Poitiers
1, Avenue Clement Ader, 86960 Futuroscope Cedex, France
{fokou, jean, hadjali, baron}@ensma.fr

Abstract. Recent advances in Web-information extraction have led to the creation of several large Knowledge Bases (KBs). Querying these KBs often results in empty answers that do not serve the users' needs. Relaxation of the failing queries is one of the cooperative techniques used to retrieve alternative results. Most of the previous work on RDF query relaxation compute a set of relaxed queries and execute them in a similarity-based ranking order. Thus, these approaches relax an RDF query without knowing its *failure causes* (*FCs*). In this paper, we study the idea of identifying these FCs to speed up the query relaxation process. We propose three relaxation strategies based on various information levels about the FCs of the user query and of its relaxed queries as well. A set of experiments conducted on the LUBM benchmark show the impact of our proposal in comparison with a state-of-the-art algorithm.

1 Introduction

Recent projects like DBpedia [1] or Knowledge Vault [2] have created Knowledge Bases (KBs) with millions of facts represented in the RDF format. Despite their large size, KBs face a significant amount of incomplete factual knowledge, which makes query answering over them often unsuccessful. For instance, a recent study on SPARQL endpoints [3] shows that ten percent of the submitted queries between May and July 2010 over DBpedia returned empty answers.

Relaxation of the failing queries is one of the cooperative techniques used to retrieve alternative results in order to serve the users' needs. In the context of RDF, current approaches generate multiple relaxed queries using different techniques such as logical relaxation based on RDFS entailment and RDFS ontologies [4–7], query rewriting rules [8], statistical language models [9] or matching functions [10]. Most of these approaches compute the similarities between the obtained relaxed queries and the user failing query and then proceed to the execution of the relaxed queries in a similarity-based ranking order. A major drawback of the above approaches is the fact they relax the user query without knowing its *Failure Causes* (*FCs*).

In our previous work [11], we have addressed the issue of finding the FCs of an RDF query by computing a set of *Minimal Failing Subqueries* (*MFSs*) and argued that they provide the user with a clear explanation about the reasons of

the empty answer retrieved. In this paper, we investigate the idea of using MFSs to perform the query relaxation process. The main idea is that MFSs can speed up this relaxation process by avoiding executing relaxed queries that still contain one or several FCs. This approach applies both for the user query, as well as for the failing relaxed queries. However, as enumerating the MFSs of a query is an NP-hard problem [12], identifying them could be sometimes disadvantageous since the MFSs computation time may be greater than the execution time of the relaxed queries avoided thanks to them. Thus, we show that there is a tradeoff between not knowing any MFSs and identifying the MFSs of each relaxed query. To do so, we propose three strategies that leverage different levels of information about the MFSs of the user query and of its relaxed queries as well. The main contributions made in this paper are the following.

1. Based on previous work [4, 5], we define the necessary data structures for relaxing the triple patterns of an RDF query and the query itself.
2. We review the state-of-the-art relaxation strategy and propose three new approaches. By doing so, we cover the full spectrum of information about the MFSs as they are respectively not, partially and fully taken into account in these strategies.
3. We provide a set of experiments on several datasets of the LUBM benchmark that were run on top of Jena TDB and Virtuoso. The analysis of the results shows that to guarantee a relaxation process with an acceptable computation time, a balancing between the information pertaining to the MFSs and the relaxed queries is needed.

This paper is organized as follows. In Section 2, we introduce the basic notions used in this paper. The data structures needed for our query relaxation strategies are then defined in Section 3. We detail these strategies in Section 4 and present our experiments to evaluate them in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Preliminaries and Problem Statement

This section formally describes the parts of RDF and SPARQL that are necessary to our proposal using the definitions given in [13]. We also recall an RDF query relaxation model borrowed from [5].

2.1 Notion of Minimal Failing Subquery (MFS)

An *RDF triple* is a triple (subject, predicate, object) $\in (U \cup B) \times U \times (U \cup B \cup L)$ where U is a set of URIs, B is a set of blank nodes and L is a set of literals. We denote by T the union $U \cup B \cup L$. An *RDF database* (or triplestore) stores a set of RDF triples in a triples table or one of its variants.

An *RDF triple pattern* t is a triple (subject, predicate, object) $\in (U \cup V) \times (U \cup V) \times (U \cup V \cup L)$, where V is a set of variables disjoint from the sets U, B and L. We denote by $var(t)$ the set of variables occurring in t. We consider *RDF*

queries defined as a conjunction of triple patterns: $Q = t_1 \wedge \cdots \wedge t_n$. Let D be an RDF database, t a triple pattern and Q an RDF query, the evaluation of t and Q over D are respectively denoted by $[[t]]_D$ and $[[Q]]_D$. This evaluation can be done under different entailment regimes as defined in the SPARQL specification. In this paper, the examples as well as our experiments are based on the RDFS entailment regime.

Given a query $Q = t_1 \wedge \cdots \wedge t_n$, a query $Q' = t_i \wedge \cdots \wedge t_j$ is a *subquery* of Q, $Q' \subseteq Q$, iff $\{t_i, \cdots, t_j\} \subseteq \{t_1, \cdots, t_n\}$. If $\{t_i, \cdots, t_j\} \subset \{t_1, \cdots, t_n\}$, we say that Q' is a *proper subquery* of Q ($Q' \subset Q$). A *Minimal Failing Subquery MFS* of a query Q is defined as follows: $[[MFS]]_D = \emptyset \wedge \nexists\, Q' \subset MFS$ *such that* $[[Q']]_D = \emptyset$. The set of all MFSs of a query Q is denoted by *mfs(Q)*. Examples of MFSs are given in the next section.

2.2 Query relaxation model

Given a triple pattern t, t' is a relaxed triple pattern obtained from t, denoted by $t \prec t'$, if $t' \neq t$ and over every RDF databases D for the given schema, $[[t]]_D \subseteq [[t']]_D$. A *relaxation rule* is a rewrite rule such that its application to a triple pattern results in a relaxed triple pattern. In this paper, we consider the following relaxation rules (*sc* and *sp* are respectively the shorter names for $subClassOf$ and $subPropertyOf$):

- *Class relaxation (R1).* $(s,\ type,\ c_1) \Rightarrow (s,\ type,\ c_2)$ if $(c_1,\ sc,\ c_2)$.
- *Property relaxation (R2).* $(s,\ p_1,\ o) \Rightarrow (s,\ p_2,\ o)$ if $(p_1,\ sp,\ p_2)$.
- *Constant to variable relaxation (R3).* If c is a constant occurring in t, then $t \Rightarrow t'$ where t' is the triple obtained by replacing c by a variable $v \notin var(t)$.

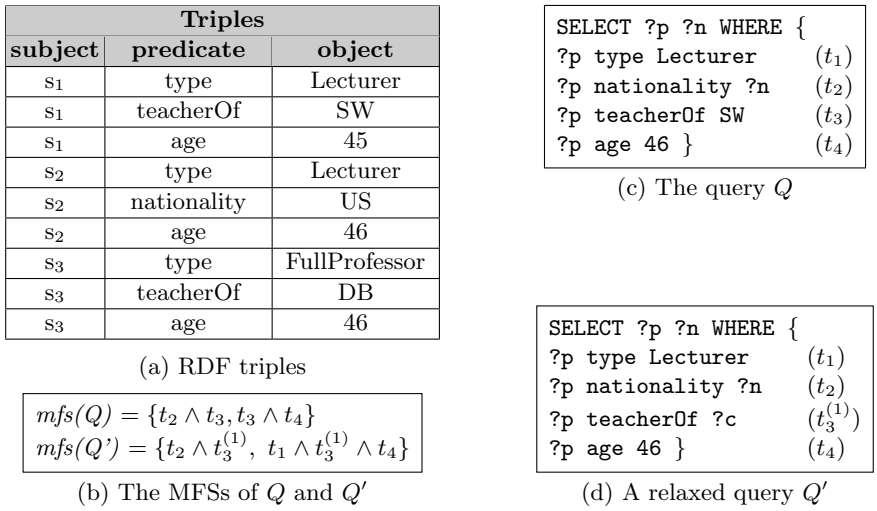
Given a triple pattern $t = (s,\ p,\ o)$ and its relaxed triple pattern $t' = (s',\ p',\ o')$, the similarity between t and t' can be defined as follows [5]:

$$Sim(t, t') = \frac{1}{3} * Sim(s, s') + \frac{1}{3} * Sim(p, p') + \frac{1}{3} * Sim(o, o')$$

We use the following similarity measures for the considered relaxation rules:

- If c is a subclass of c', $Sim(c, c') = \frac{IC(c')}{IC(c)}$ where $IC(c) = -logPr(c)$ and $Pr(c) = \frac{|Instances(c)|}{|Instances|}$ ($Instances(c)$ is the set of instances of c and $Instances$ the set of all instances of the RDF database).
- If p is a subproperty of p', $Sim(p, p') = \frac{IC(p')}{IC(p)}$ where $IC(p) = -logPr(p)$ and $Pr(p) = \frac{|Triples(p)|}{|Triples|}$ ($Triples(p)$ is the set of triples concerning p and $Triples$ the set of all triples of the RDF database).
- If c is a constant and v a variable, $Sim(c, v) = 0$.

Given a user query $Q = t_1 \wedge \cdots \wedge t_n$ and a query $Q' = t'_1 \wedge \cdots \wedge t'_n$, Q' is a relaxed query of Q, denoted by $Q \prec Q'$, if (i) for each triple pattern t_i, $t_i \preceq t'_i$ (either $t_i = t'_i$ or $t_i \prec t'_i$) and (ii) for at least one triple pattern t_j, $t_j \prec t'_j$.



Triples		
subject	**predicate**	**object**
s_1	type	Lecturer
s_1	teacherOf	SW
s_1	age	45
s_2	type	Lecturer
s_2	nationality	US
s_2	age	46
s_3	type	FullProfessor
s_3	teacherOf	DB
s_3	age	46

(a) RDF triples

mfs(Q) $= \{t_2 \wedge t_3, t_3 \wedge t_4\}$
mfs(Q') $= \{t_2 \wedge t_3^{(1)},\ t_1 \wedge t_3^{(1)} \wedge t_4\}$

(b) The MFSs of Q and Q'

```
SELECT ?p ?n WHERE {
?p type Lecturer      (t1)
?p nationality ?n     (t2)
?p teacherOf SW       (t3)
?p age 46 }           (t4)
```

(c) The query Q

```
SELECT ?p ?n WHERE {
?p type Lecturer      (t1)
?p nationality ?n     (t2)
?p teacherOf ?c       (t3(1))
?p age 46 }           (t4)
```

(d) A relaxed query Q'

Fig. 1. Example of a relaxed query of Q

Figure 1 presents an example of a relaxed query Q' of $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$ with their MFSs. In this example $t_3^{(1)}$ is a relaxed triple pattern of t_3.

As for the similarity between a query Q and its relaxed query Q', we use $Sim(Q, Q') = \prod_{i=1}^{n} Sim(t_i, t'_i)$. Let D be an RDF database, if $\mu \in [[Q']]_D$ and $\mu \notin [[Q]]_D$, then μ is an *approximate answer* of Q. The approximate answers are ranked thanks to a score defined by: $Score(\mu, Q) = \{\ max(Sim(Q, Q')) \mid Q \prec Q' \wedge \mu \in [[Q']]_D\ \}$.

Problem Statement. Knowing the set of MFSs of a failing RDF query Q, we are concerned with finding the top-k approximate answers of Q efficiently.

3 Query Relaxation Data Structures

We first define the data structures needed for the proposed relaxation strategies.

3.1 Triple Pattern Relaxation

Let t be a triple pattern. One or several relaxation rules may be applied to t. The same relaxation rules may also be applied several times to the same triple pattern. We denote by $t^{(0)}$ the original triple pattern, by $t^{(i)}$ the i-th best relaxation of t in terms of similarity with t and by $nbRel(t)$ the number of relaxed triple patterns of t. By definition, if $i < j$, then $Sim(t^{(i)}, t) \geq Sim(t^{(j)}, t)$. However, the following relationship does not necessarily hold $t^{(i)} \preceq t^{(j)}$.

Example. Let us assume that *FullProfessor* is a subclass of *Professor*. By applying the previous relaxation rules to the triple pattern (?X, *type, FullProfessor*), we find the following relaxed triple patterns of t: $t^{(1)}$ = (?X, *type,*

Professor) where $Sim(t^{(1)}, t) = 0.9$ and $t^{(2)} = (?X,\ ?Y,\ \textit{FullProfessor})$ where $Sim(t^{(2)}, t) = \frac{2}{3}$. Yet, $t^{(1)} \not\preceq t^{(2)}$.

Let t be a triple pattern and *ApplyRules(t)* be a function that returns all the relaxed triple patterns of t resulting from the application of the three considered relaxation rules. Algorithm 1 computes the relaxed triple patterns of t ordered by similarity. An example of relaxation of the triple pattern ($?X$, *type, FullProfessor*) using our three relaxation rules (*R1*, *R2* and *R3*) is presented in Figure 2.

Algorithm 1: Computation of the relaxation of t ordered by similarity

```
Relax(t)
    input  : A triple pattern t;
    output: the list of relaxed triple patterns of t : t^(0) ··· t^(n);
1   T ← ∅; Res ← ∅; // Res:  resulting list of t^(i) sorted by sim
2   T.enqueue(t); // T: priority queue of t^(i) sorted by sim
3   while T ≠ ∅ do
4       t_i = T.dequeue();
5       Res.enqueue(t_i);
6       foreach triple pattern t_j ∈ ApplyRules(t_i) do
7           if t_j ∉ T then
8               T.enqueue(t_j);
9   return Res;
```

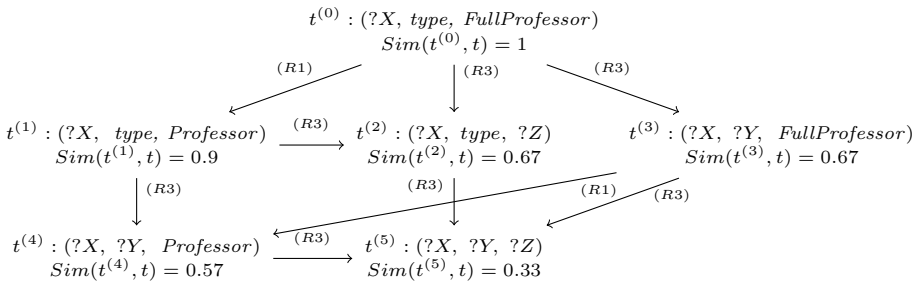


Fig. 2. Relaxation of a Triple Pattern

3.2 Query Relaxation Graph

Let $Q = t_1^{(0)} \wedge \cdots \wedge t_n^{(0)}$ be the original failing RDF query. The set of relaxed queries of Q is $\{\ Q' = t_1^{(i_1)} \wedge \cdots \wedge t_n^{(i_n)} \mid \exists k \in [1, n] : i_k > 0\ \}$. Inspired by [5][1],

[1] The proposed relaxation graph is not equivalent to the one proposed in [5]. Indeed, an edge between two queries Q_1 and Q_2 does not necessarily mean $Q_1 \preceq Q_2$. This property simplifies the computation of the children of a node in the graph.

we organize this set of relaxed queries in a graph structure. The initial query is at the top of this graph and each relaxed query is a node of this graph. An edge from node $Q_i = t_1^{(i_1)} \wedge \cdots \wedge t_n^{(i_n)}$ to $Q_j = t_1^{(j_1)} \wedge \cdots \wedge t_n^{(j_n)}$ exists if and only if (i) for one triple pattern t_l, $i_l = j_l - 1$ and (ii) for each other triple patterns t_k, $i_k = j_k$. Thus, by construction, $Sim(Q, Q_i) \geq Sim(Q, Q_j)$. This graph has different levels according to the lengths of the paths from the root to relaxed queries. At level h we find all relaxed queries $Q' = t_1^{(i_1)} \wedge \cdots \wedge t_n^{(i_n)}$ such as $\sum_{k=1}^{n} i_k = h$. The number of relaxed queries in this relaxation graph is $\prod_{i=1}^{n}(nbRel(t_i) + 1)$.

Figure 3 gives an example of a relaxation graph for our sample query $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$. For simplification, this example assumes that each triple pattern can only be relaxed a single time. We do not give the algorithm to compute this complete query relaxation graph as it is incrementally built in the relaxation strategies proposed in the next section.

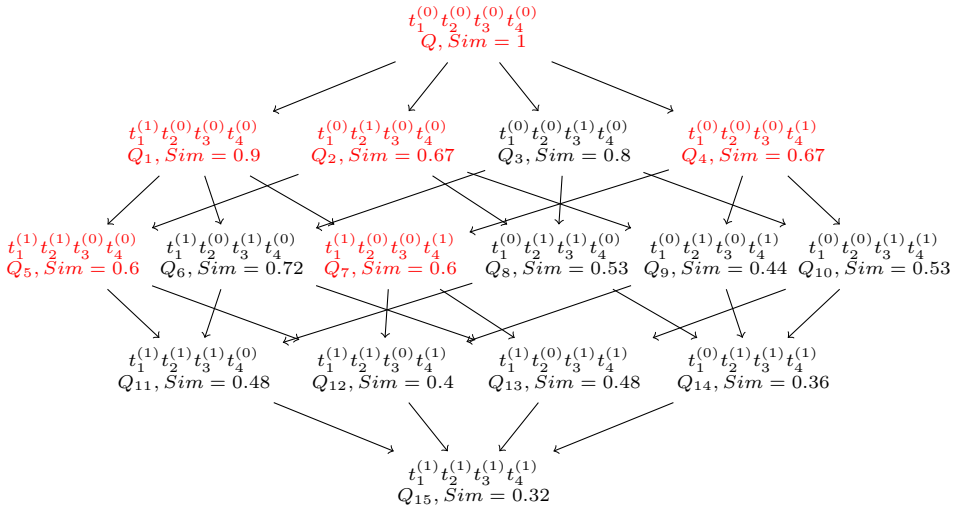


Fig. 3. Query Relaxation Graph

4 Query Relaxation Strategies

In this section, we first review a state-of-the-art strategy for exploring the query relaxation graph introduced in the previous section. Then we propose three MFS-based strategies.

4.1 Best-First Search (BFS)

It can be easily shown that the h-th best relaxed query Q' of Q is at level h or less of the query relaxation graph. Thanks to this property, the top-k approximate answers could be found with an algorithm that executes the relaxed queries of the graph in the ranking order such as the one proposed in [5]. For example, this algorithm explores the query relaxation graph depicted in Figure 3 in the following order : $Q_1, Q_3, Q_6, Q_2, Q_4, Q_5, Q_7, Q_8, Q_{10}, Q_{11}, Q_{13}, Q_9, Q_{12}, Q_{14}, Q_{15}$.

In the best-case scenario, this algorithm will only execute one relaxed query to find the top-k approximate answers. In the worst-case scenario, it has to execute all the queries of the graph. As there is an exponential number of relaxed queries (in terms of query size), this algorithm may require an exponential time.

As the causes of the query's failure are unknown in this algorithm, it may execute queries that cannot have any answers and/or relax triple patterns that do not need to be modified. As it will be seen later, the MFSs provide important clues to avoid these pitfalls.

4.2 MFS-Based Search (MBS)

As stated in the following propositon, the MFSs of the failing query identify some relaxed queries that will necessarily fail.

Proposition 1. *Let Q' be a relaxed query of Q. If Q' does not relax at least one triple pattern of each MFS of Q, then Q' is failing.*

Proof. If there is one MFS of Q, denoted Q^*, such as none of its triple patterns has been relaxed in Q', then $Q^* \subseteq Q'$. A query that contains a failing query, also fails. By definition of an MFS, Q^* is a failing query. Thus Q' is also failing.

Based on Proposition 1, we have devised the Algorithm 2 named *MFS-Based Search* (*MBS*). This algorithm uses a priority queue RQ of relaxed queries ordered by their similarities with Q. Initially the query Q is added to this queue. It explores each query enqueued in RQ and stops when RQ is empty or when the number of expected answers is obtained (line 3). Each query of RQ is explored as follows. If this query is not labelled as failing, it is executed and its answers are added to the result Res (lines 5-6). Then, all the children of this query that have not already been proceeded (labelled as marked) are added to RQ (lines 7-10). If the added child contains an MFS of Q, this query is labelled as failing (lines 12-13). This way, MBS prunes the search space of the query relaxation graph with failing RDF queries identified with MFSs. In this process, Algorithm 1 is used to find the relaxed triple patterns $t^{(i)}$ of each triple pattern t as well as its number of relaxed triple patterns $nbRel(t)$.

If the MFSs of our query $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$ are $t_2 \wedge t_3$ and $t_3 \wedge t_4$, then all the queries in red in Figure 3 (Q_1, Q_2, Q_4, Q_5, Q_7) can be pruned from the relaxation graph thanks to Proposition 1. Thus, MBS executes the queries in the following order : $Q_3, Q_6, Q_8, Q_{10}, Q_{11}, Q_{13}, Q_9, Q_{12}, Q_{14}, Q_{15}$.

Algorithm 2: MFS-Based Query Relaxation

Relax(*Q*, *mfs(Q)*, *D*, *k*)

inputs : A failing query Q ; the set of MFSs of Q : *mfs(Q)*;
an RDF database D ; the number of expected answers k

output: a set of top-k approximate results of Q denoted by Res;

1 $Res \leftarrow \emptyset$; $RQ \leftarrow \emptyset$; `// the relaxed queries ordered by similarities`
2 $RQ.enqueue(Q)$; label Q as failing;
3 **while** $RQ \neq \emptyset \wedge |Res| < k$ **do**
4 $Q' = RQ.dequeue()$;
5 **if** Q' *is not labelled as failing* **then**
6 $Res \leftarrow Res \cup [[Q']]_D$;
7 **foreach** *triple pattern* $t_k^{(i_k)} \in Q'$ *such that* $i_k < nbRel(t_k)$ **do**
8 $Q_c \leftarrow t_1^{(i_1)} \wedge \cdots \wedge t_k^{(i_{k+1})} \wedge \cdots \wedge t_n^{(i_n)}$; `// a child of` Q'
9 **if** Q_c *is not labelled as marked* **then** `// not explored`
10 $RQ.enqueue(Q_c)$;
11 label Q_c as marked;
12 **if** $\exists Q^* \in$ *mfs(Q) such that* $Q^* \subseteq Q_c$ **then**
13 label Q_c as failing;
14 **return** Res;

4.3 Optimized MFS-Based Search (O-MBS)

In the previous approach, we only use the MFSs of the initial query to prune the search space. The idea behind the *Optimized MFS-Based Search* (*O-MBS*) is that the MFSs of the initial query Q give some clues on the MFSs of a relaxed query Q' of Q. Intuitively, a relaxed query Q' of Q fails if and only if at least one MFS of Q has not been repaired in Q' or if there is a failing query in Q' that was not minimal in Q. More formally, let M_Q be an MFS of Q, we denote by $M_Q^{\uparrow Q'}$ the query that corresponds to M_Q in Q'. By extension, we denote by $mfs^{\uparrow Q'}(Q)$ the queries corresponding to the MFSs of Q in Q'. For instance, in the example given in Figure 1, $mfs^{\uparrow Q'}(Q) = \{t_2 \wedge t_3^{(1)},\ t_3^{(1)} \wedge t_4\}$. This example also shows that the following relationship does not necessarily hold: *mfs(Q')* $\subseteq mfs^{\uparrow Q'}(Q)$. However, as we now prove, each MFS of Q' includes a query of $mfs^{\uparrow Q'}(Q)$.

Proposition 2. *For any MFS* $M_{Q'}$ *of* Q' *there is an MFS* M_Q *of* Q *such that* $M_Q^{\uparrow Q'} \subseteq M_{Q'}$.

Proof. Let $M_{Q'}$ be an MFS of Q'. By definition $M_{Q'}$ is failing. As $[[M_{Q'}^{\uparrow Q}]] \subseteq [[M_{Q'}]]$, $M_{Q'}^{\uparrow Q}$ is also failing. So, $M_{Q'}^{\uparrow Q}$ contains an MFS M_Q of Q and thus $M_Q^{\uparrow Q'} \subseteq M_{Q'}$.

Thus, if an MFS has been repaired, there can still be some queries that include this MFS and fail. Identifying these new MFSs is not easy. Indeed, the

number of queries that include the repaired MFS is exponential in terms of the number of query triple patterns. Thus, the O-MBS strategy is only based on the MFSs that are not repaired. It extends the MBS algorithm (Algorithm 2) as follows. For each relaxed query Q' explored in the query relaxation graph, each query $M_{Q'} \in mfs^{\uparrow Q'}(Q)$ is executed. If the query $M_{Q'}$ is failing, $M_{Q'}$ is an MFS of Q' and thus, all queries that contains $M_{Q'}$ can be pruned from the query relaxation graph (thanks to Proposition 1). To optimize this process, the discovered MFSs of each query Q' explored are recorded. They are denoted *dmfs(Q')*. When a query Q_1 is explored, the O-MBS strategy only executes the MFSs in $dmfs(Q_0)$, where Q_0 is the last explored query such that $Q_0 \prec Q_1$. Indeed, it is unnecessary to execute the MFSs that was already repaired previously by Q_0.

Coming back to our example depicted in Figure 3, let us assume that the query Q_3 does not repair the MFS $t_2 \wedge t_3$ (i.e, $t_2 \wedge t_3^{(1)}$ is failing). Then, the queries Q_6, Q_{10} and Q_{13} are pruned from the relaxation graph. If none of the MFSs of the following explored queries are discovered, then O-MBS executes the queries in the following order : $Q_3, Q_8, Q_{11}, Q_9, Q_{12}, Q_{14}, Q_{15}$.

4.4 Full MFS-Based Search (F-MBS)

In the previous strategy, all the MFSs of an explored relaxed query are not necessarily discovered. In this section, we propose an approach to compute this complete set of MFSs. By proposing this approach, we want to investigate if it is worth computing the set of MFSs of each explored node of the query relaxation graph, i.e., if this computation time is acceptable in comparison with the number of relaxed queries that are pruned thanks to the discovered MFSs. This strategy called *Full MFS-Based Search* (*F-MBS*) is based on the two following corollaries that are directly derived from Proposition 2.

Corollary 1. *If all the queries* $M_Q \in mfs^{\uparrow Q'}(Q)$ *are failing, then:* $mfs(Q') = mfs^{\uparrow Q'}(Q)$.

Corollary 2. *Each MFS* $M_{Q'}$ *of* Q' *contains the triple patterns that are shared by the queries of* $mfs^{\uparrow Q'}(Q)$

Thanks to these corollaries, F-MBS extends the O-MBS strategy as follows. For each relaxed query Q', we execute all the MFSs of $mfs^{\uparrow Q'}(Q_0)$, where Q_0 is the last explored query such that $Q_0 \prec Q'$. If all these queries are failing, then $mfs(Q') = mfs^{\uparrow Q'}(Q_0)$ (thanks to Corollary 1). Otherwise, we execute an optimized version of the LBA algorithm [11] to find the MFSs of Q'. As in the previous strategies, the queries that include at least one of the identified MFSs of Q' are pruned from the query relaxation graph.

Because of space limitation, we only describe the main principle of the optimized version of LBA. Let us first describe the main steps of the original version of this algorithm. The LBA algorithm explores the lattice of subqueries of a query Q' built by removing some triple patterns of Q'. It follows a three-steps

procedure: (1) find an MFS of Q', (2) compute the maximal queries that do not include the MFS previously found and (3) apply this process recursively on the failing queries previously computed.

Thanks to the discovered MFSs of Q', *dmfs(Q')*, this algorithm is optimized as follows. Instead of executing the first two steps, it directly computes the maximal queries that do not include the MFSs of *dmfs(Q')*. Moreover, using Corollary 2, the search for the next MFS is simplified as we know that it contains the triple patterns shared by the MFSs of $mfs^{\uparrow Q'}(Q_0)$. In the worst case scenario when none of the MFSs were discovered and no triple pattern is shared by the MFSs of $mfs^{\uparrow Q'}(Q_0)$, LBA is executed in its original version (it may cost exponential time in the worst case). In the best case scenario where only one MFS is missing and most of its triple patterns are included in the discovered MFSs, LBA will only execute one query for each missing triple pattern in this MFS.

Consider again the example depicted in Figure 3 and let us assume that $mfs(Q_3) = \{t_2 \wedge t_3^{(1)},\ t_1 \wedge t_3^{(1)} \wedge t_4\}$. Then, the queries Q_6, Q_{10}, Q_{13} and Q_8 are pruned from the query relaxation graph. If the MFSs of the following explored queries do not help in pruning further the graph, then F-MBS executes the queries in the following order : $Q_3, Q_{11}, Q_9, Q_{12}, Q_{14}, Q_{15}$.

5 Experimental Evaluation

Experimental setup. We have implemented the MBS, O-MBS and F-MBS algorithms in JAVA 1.8 64 bits. These algorithms take as inputs a failing SPARQL query and a number of expected answers k. They return a maximum of k approximate answers of this query. These algorithms are based on the MFSs of the failing query, which are computed with the LBA algorithm [11]. This implementation can be run on top of any triplestore that supports the SPARQL language. In our experiments, they were run on top of Jena TDB (version 3.0.0) and Virtuoso (version 7.2.1). Our implementation is available at http://www.lias-lab.fr/forge/projects/qars.

Our experiments were conducted on a Ubuntu Server 14.04.02 LTS system with Intel XEON CPU E5-2630 v3 @2.4Ghz CPU and 32GB RAM. All times presented are the average of five consecutive runs of the algorithms. Before the actual measured run starts, we run the algorithm once.

Dataset and Queries. As in previous work on RDF query relaxation [5], we used datasets generated with the LUBM benchmark. The used datasets range from LUBM100 (17M triples) to LUBM1K (167M triples). These datasets include both the initial triples generated with the LUBM benchmark and the implicit triples entailed by the RDFS semantics. Statistics on these datasets are precomputed and used later by our algorithms. They are composed of the classes and properties hierarchies, the number of instances by class, the number of triples by property and the total number of instances and triples.

As the workload used in [5] only involves queries with a maximum of 5 triple patterns and 1 MFS, we have modified these 7 queries. The resulting queries

given in Table 1[2] cover the main query patterns (star, chain and composite), range between 1 and 15 triple patterns and include 1 up to 4 MFSs.

Q1 (1 MFS)	SELECT * WHERE { ?X type FullProfessor . ?X title 'Dr' }
Q2 (3 MFSs)	SELECT * WHERE { UndergraduateStudent33 advisor ?Y1 . ?Y1 doctoralDegreeFrom ?Y2 . ?Y2 hasAlumnus ?Y3 . ?Y3 title ?Y4 }
Q3 (4 MFSs)	SELECT * WHERE { ?X type FullProfessor . ?X publicationAuthor ?Y1 . ?X worksFor ?Y2 . ?Y3 advisor ?X . ?X title ?Y4 }
Q4 (3 MFSs)	SELECT * WHERE { ?X type UndergraduateStudent . ?X memberOf ?Y1 . ?X mastersDegreeFrom University822 . ?X emailAddress ?Y2 . ?X advisor FullProfessor0 . ?X takesCourse ?Y3 . ?X name ?Y4 }
Q5 (3 MFSs)	SELECT * WHERE { ?X type FullProfessor . ?X doctoralDegreeFrom ?Y1 . ?X memberOf ?Y2 . ?X headOf ?Y1 . ?X title ?Y3 . ?X officeNumber ?Y4 . ?X researchInterest ?Y5 . ?Y6 advisor ?X . ?Y6 name ?Y7 }
Q6 (4 MFSs)	SELECT * WHERE { ?X type Faculty . ?X doctoralDegreeFrom ?Y1 . ?X memberOf ?Y2 . ?X headOf ?Y3 . ?X title ?Y4 . ?X officeNumber ?Y5 . ?X researchInterest ?Y6 . ?X name 'FullProfessor3' . ?X emailAddress ?Y7 . ?X age ?Y8 . ?X mastersDegreeFrom Department2 . ?X undergraduateDegreeFrom ?Y9 }
Q7 (4 MFSs)	SELECT * WHERE { ?X type Professor . ?X teacherOf Course2 . ?X name ?Y1 . ?X age ?Y2 . ?X emailAddress ?Y3 . ?X mastersDegreeFrom ?Y4 . ?X worksFor ?Y5 . ?Y5 subOrganizationOf ?Y6 . ?Y6 name ?Y7 . ?Y8 advisor ?X . ?Y8 mastersDegreeFrom ?Y4 . ?Y8 memberOf ?Y9 . ?Y8 emailAdress ?Y10 . ?Y8 takesCourse ?Y11 . ?Y8 name ?Y12 }

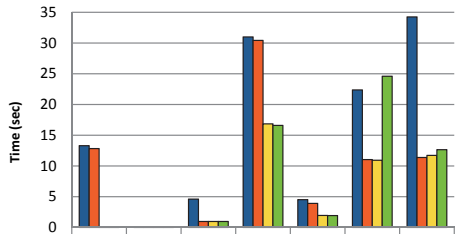
Table 1. Workload queries

Experiment 1. We have first evaluated the scalability properties of MBS, O-MBS and F-MBS in comparison with our own implementation of the BFS algorithm proposed in [5]. This experiment has been run on Jena TDB with the LUBM100 dataset and k (the number of approximate answers) set to 50. Figure 4 and 5 show respectively the execution time and the number of executed queries for each workload query. For the MBS, O-MBS and F-MBS algorithms, these measures include both the computation of the MFSs and the execution of the relaxed queries.

In this experiment, BFS executes more queries than our algorithms. This difference increases with the size of the query. In particular, for queries Q6 and Q7 that have more than 10 triple patterns and 4 MFSs, this algorithm needs to explore a large part of the query relaxation graph to repair the MFSs. This result in more than 1000 executed queries. In the case of Q2, this difference in the number of executed queries does not imply a larger execution time as the relaxed queries have short execution times. But, for other queries, our fastest

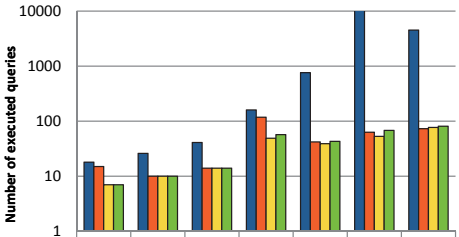
[2] For readability, we shorten URIs and omit namespaces

algorithm O-MBS outperforms BFS by more than a factor of 2 (average query times go from around 18 seconds to around 7 seconds).



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
BFS	13,3	0,06	4,61	30,99	4,52	22,37	34,25
MBS	12,83	0,01	0,97	30,43	3,91	11,04	11,38
O-MBS	0,05	0,01	0,97	16,85	1,95	10,93	11,72
F-MBS	0,06	0,01	0,97	16,60	1,93	24,60	12,65

Fig. 4. Execution time (Jena)



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
BFS	18	26	41	160	761	13809	4520
MBS	15	10	14	118	42	63	73
O-MBS	7	10	14	49	39	53	77
F-MBS	7	10	14	57	43	68	81

Fig. 5. # Executed queries (log scale)

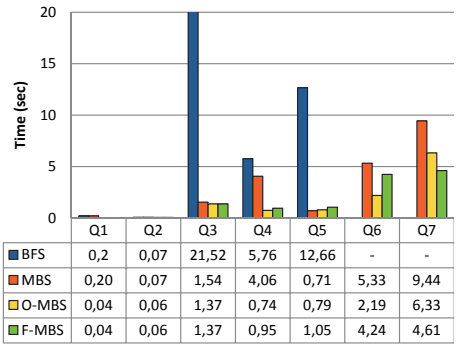
Considering our proposed algorithms, O-MBS is better than the other strategies w.r.t minimizing the computation time as well as the number of executed queries in the majority of cases. Only the MBS algorithm executes less query than O-MBS for Q7 and it does not result in a larger execution time. For other queries O-MBS has better performance than MBS and in particular for Q1, Q4 and Q5. Let us consider Q1 to illustrate how O-MBS reduces the number of executed queries. Q1 has only 1 MFS: $(?X, title, 'Dr')$ as there is not any *title* in the RDF database. MBS and O-MBS start by relaxing this triple pattern, for instance by $(?X, title, ?Y)$. Then the two algorithms differ in their strategies. O-MBS executes this triple pattern and find that it is still an MFS for the relaxed query. As a consequence it will relax again this triple pattern and find approximate answers. Conversely, as MBS only uses the MFSs of the initial query, this algorithm tries to relax the other triple pattern of the initial query, which will result in relaxed queries that fail. Thus, MBS will execute more queries than O-MBS and its execution time will be significantly longer.

Our last algorithm F-MBS always executes an equal or superior number of queries compared to O-MBS. This behavior is explained by the fact that the number of queries executed to find the MFSs of the relaxed queries is greater than the number of queries pruned in the relaxation graph (thanks to MFSs). We illustrate this fact with Q7. O-MBS executes 25 relaxed queries and 52 queries for computing the MFSs while F-MBS only executes 7 relaxed queries but 74 to compute all the MFSs. In some queries such as Q6, this difference impacts negatively the execution time of F-MBS in comparison with O-MBS.

Experiment 2. In the second experiment, we have evaluated the impact of the triplestore on the performance of our algorithms. Figure 6 presents the execution time of the different algorithms run on top of Virtuoso in the same conditions as the previous experiment. Again, we can observe that our algo-

rithms perform better than BFS. For this latter algorithm, the query Q6 and Q7 took more than 1 hour to execute and thus their execution times are not shown. Even if the trends observed on Jena TDB are confirmed on Virtuoso, the execution times of the algorithms on Virtuoso differ significantly from the ones obtained with Jena TDB. For Q1 and Q4 the execution times are better on Virtuso and conversely for other queries. This is in agreement with the findings of several benchmarks (e.g., [14]) that no triplestore is superior for all queries.

Experiment 3. In the last experiment we have evaluated the scalability of our algorithms when the size of the dataset increases. This experiment was run on Jena TDB with k set to 50. Figure 7 presents the execution time of the different algorithms for the query Q5 when the dataset ranges from 17M to 167M triples. In this experiment, the algorithms scale almost linearly with the size of the repository. We obtained the same result for other queries. As the LUBM generates data that have proportionally similar statistics, our queries have the same MFSs on the different repositories and the same relaxed queries are executed in the same order. As the execution times of the queries scale linearly with the size of the datasets, this explain the result of this experiment.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
BFS	0,2	0,07	21,52	5,76	12,66	-	-
MBS	0,20	0,07	1,54	4,06	0,71	5,33	9,44
O-MBS	0,04	0,06	1,37	0,74	0,79	2,19	6,33
F-MBS	0,04	0,06	1,37	0,95	1,05	4,24	4,61

Fig. 6. Execution time (Virtuoso)

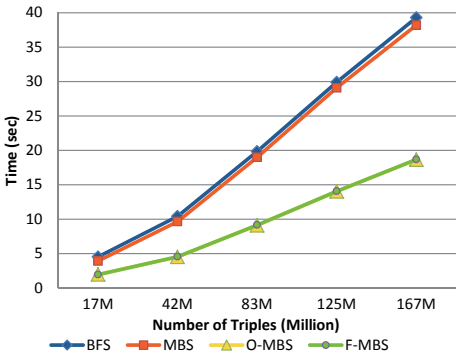


Fig. 7. Execution time vs data size (Jena)

6 Related Work

We provide here a review of the closest approaches related to our proposal both in the context of RDF and relational databases. In the first setting, Hurtado et al. [4] propose a relaxation approach based on the inferences rules of RDFS. This approach leverages the *subClassOf*, *subPropertyOf*, *domain* and *range* relationships of RDFS to relax a SPARQL query. The end-user can choose the triples that must be relaxed in a query by using the *RELAX* clause. Huang et al. [5] use the same relaxation techniques based on RDFS entailment. To ensure quality of alternative answers, they leverage a semantic similarity measure based on concept statistics. Several optimization techniques are also proposed to obtain

the top-k approximate answers efficiently. Elbassuoni et al. [9] show a process for finding similar values of a precise value needed for a query relaxation.

In our previous work [6], we have proposed a set of primitive relaxation operators and have shown how these operators can be integrated in SPARQL in a simple or combined way. Cali et al. [7] have also extended a fragment of this language with query approximation and relaxation operators. As an alternative to query relaxation, *query auto-completion* techniques check the data during query formulation to avoid empty answers (e.g., [15]). It is worth noticing that none of the above approaches has considered the issue related to the causes of an RDF query failure and thus the issue of MFS computation. In our recent work [11], we have addressed this issue but only for providing users with some explanation about this failure.

As for relational databases, many approaches have been proposed for query relaxation (see Bosc et al. [16] for an overview). In particular, Godfrey [12] has defined the algorithmic complexity of the problem of identifying the MFSs of failing relational queries and developed the ISHMAEL algorithm for retrieving them. Jannach [17] studied the concept of MFS in the recommendation system setting. The MFSs computed are used to relax the query at hand by removing some parts of the query. Bosc et al. [16] and Pivert et al. [18] extended Godfrey's approach to the fuzzy queries context. In [16], to speed up the relaxation process, the authors attempt to leverage the MFSs when relaxing the failing query. Unfortunately, this approach does not work in all situations (e.g., when the set of MFSs of the relaxed query is not subsumed by the one of the original query).

As it can be seen, the MFS paradigm has never been used to guide the relaxation process in the context of RDF. To the best of our knowledge, this is the first attempt towards an MFS-driven relaxation of RDF queries.

7 Conclusion and Perspectives

In this paper, we have proposed three strategies to relax an RDF query. Their originalities is that they use different levels of information about the FCs of the initial query and its relaxed queries. In the first strategy, named MBS, only the FCs of the initial query are used to prune from the search space all the relaxed queries that include FCs. The second strategy named O-MBS extends the previous one by searching the FCs that remain in the relaxed query. In this strategy, all the FCs of a relaxed query are not necessarily discovered. Our last strategy named F-MBS fills this gap by using an optimized version of a previous work algorithm to find the FCs of the relaxed queries.

We have run several experiments on the LUBM benchmark with two triple-stores to compare these strategies with a state-of-the-art strategy, which consists in executing the relaxed queries in their ranking order. In these experiments, our best strategy O-MBS outperforms it by more than a factor of 2. O-MBS is a good compromise between MBS, which often does not use enough information about the FCs, and F-MBS which uses too much information about them.

This paper opens many perspectives. As our approach is defined for conjunctive RDF queries, we plan to extend it to support other SPARQL queries. Studying the relevance of our strategies when they are applied on other query relaxation models that use different relaxation operators is another perspective. As our strategies gather a lot of information about the failure of many queries, we intend to design an interactive approach based on our strategies. Finally, we plan to investigate whether the FCs of a query could be used in conjunction with other cooperative techniques that aim at handling the empty-answer problem.

References

1. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. Semantic Web **6**(2) (2015) 167–195
2. Dong, X., Gabrilovich, E., Heitz, G., Horn, W., Lao, N., Murphy, K., Strohmann, T., Sun, S., Zhang, W.: Knowledge Vault: A Web-scale Approach to Probabilistic Knowledge Fusion. In: ACM SIGKDD. (2014) 601–610
3. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.N.: LSQ: the linked SPARQL queries dataset. In: The Semantic Web - ISWC. (2015) 261–269
4. Hurtado, C.A., Poulovassilis, A., Wood, P.T.: Query Relaxation in RDF. Journal on Data Semantics X **10** (2008) 31–61
5. Huang, H., Liu, C., Zhou, X.: Approximating query answering on RDF databases. Journal of World Wide Web **15**(1) (2012) 89–114
6. Fokou, G., Jean, S., Hadjali, A.: Endowing Semantic Query Languages with Advanced Relaxation Capabilities. In: ISMIS'14. (2014) 512–517
7. Calí, A., Frosini, R., Poulovassilis, A., Wood, P.: Flexible Querying for SPARQL. In: ODBASE'14. (2014) 473–490
8. Dolog, P., Stuckenschmidt, H., Wache, H., Diederich, J.: Relaxing RDF queries based on user and domain preferences. IJIIS **33**(3) (2009) 239–260
9. Elbassuoni, S., Ramanath, M., Weikum, G.: Query Relaxation for Entity-Relationship Search. In: ESWC'11. (2011) 62–76
10. Hogan, A., Mellotte, M., Powell, G., Stampouli, D.: Towards Fuzzy Query-Relaxation for RDF. In: ESWC'12. (2012) 687–702
11. Fokou, G., Jean, S., Hadjali, A., Baron, M.: Cooperative techniques for SPARQL query relaxation in RDF databases. In: ESWC'15. (2015) 237–252
12. Godfrey, P.: Minimization in Cooperative Response to Failing Database Queries. International Journal of Cooperative Information Systems **6**(2) (1997) 95–149
13. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transaction on Database Systems **34**(3) (2009) 16:1–16:45
14. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. Semantic Web and Information Systems **5**(2) (2009) 1–24
15. Campinas, S.: Live SPARQL Auto-Completion. In: ISWC'14 (Posters & Demos). (2014) 477–480
16. Bosc, P., Hadjali, A., Pivert, O.: Incremental controlled relaxation of failing flexible queries. JIIS **33**(3) (2009) 261–283
17. Jannach, D.: Fast computation of query relaxations for knowledge-based recommenders. AI Communications **22**(4) (2009) 235–248
18. Pivert, O., Smits, G., Hadjali, A., Jaudoin, H.: Efficient Detection of Minimal Failing Subqueries in a Fuzzy Querying Context. In: ADBIS'11. (2011) 243–256