Cooperative Techniques for SPARQL Query Relaxation in RDF Databases

Géraud Fokou, Stéphane Jean, Allel Hadjali(✉), and Mickael Baron

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

Abstract. This paper addresses the problem of failing RDF queries. Query relaxation is one of the cooperative techniques that allows providing users with alternative answers instead of an empty result. While previous works on query relaxation over RDF data have focused on defining new relaxation operators, we investigate in this paper techniques to find the parts of an RDF query that are responsible of its failure. Finding such subqueries, named *Minimal Failing Subqueries* (MFSs), is of great interest to efficiently perform the relaxation process. We propose two algorithmic approaches for computing MFSs. The first approach (LBA) intelligently leverages the subquery lattice of the initial RDF query while the second approach (MBA) is based on a particular matrix that improves the performance of LBA. Our approaches also compute a particular kind of relaxed RDF queries, called *Maximal Succeeding Subqueries* (XSSs). XSSs are subqueries with a maximal number of triple patterns of the initial query. To validate our approaches, a set of thorough experiments is conducted on the LUBM benchmark and a comparative study with other approaches is done.

1 Introduction

With the extensive adoption of RDF, specialized databases called RDF databases (or triple-store) have been developed to manage large amounts of RDF data (e.g., Jena [1]). RDF databases are based on a generic representation (a triples table or one of its variants) that can manage a set of diverse RDF data, ranging from structured data to unstructured data. This flexibility makes it difficult for users to correctly formulate RDF queries that return the desired answers. This is why user RDF queries often return an empty result.

Query relaxation is one of the cooperative techniques that allows providing users with alternative answers instead of an empty result. Several works have been proposed to relax queries in the RDF context [2–8]. They mainly focus either on introducing new relaxation operators or on the efficient processing of *top-k* RDF queries. Usually, only some parts of a failing RDF query are responsible of its failure. Finding such subqueries, named *Minimal Failing Subqueries* (MFSs), provides the user with an explanation of the empty result returned and a guide to relax his/her query.

F. Gandon et al. (Eds.): ESWC 2015, LNCS 9088, pp. 237–252, 2015.
DOI: 10.1007/978-3-319-18818-8_15

To the best of our knowledge, no work exists in the literature that addresses the issue of computing MFSs of failing RDF queries. Inspired by some previous works in relational databases [9] and recommendation systems [10], we propose in this paper two algorithmic approaches for searching MFSs of failing RDF queries. The first one is a smart exploration of the subquery lattice of the failing query, while the second one relies on a particular matrix obtained by executing each triple pattern involved in the query. These algorithms also compute a particular kind of relaxed queries, called *Maximal Succeeding Subqueries* (XSSs), that return non-empty answers. Each XSS provides a simple way to relax a query by removing or making optional the set of triple patterns that are not in an XSS. Our contributions are summarized as follows.

1. We propose an adapted and extended variant of Godfrey's approach [9], called LBA, for computing both the MFSs and XSSs of a failing RDF query. Both properties and algorithmic aspects of LBA are investigated.
2. Inspired by the work done in [10], we devise a second approach, called MBA, which only requires n queries over the target RDF database, where n is the number of query triple patterns. The skyline of the matrix on which this approach is based, directly provides the XSSs of a query. This matrix can also improve the performance of LBA.
3. We study the efficiency and effectiveness of the above approaches through a set of experiments conducted on two datasets of the LUBM benchmark. We also compare our propositions with existing similar approaches on the basis of the experimental results obtained.

The paper is structured as follows. Section 2 introduces some basic notions and formalizes the problem we consider. Sections 3 and 4 present our approaches LBA and MBA to find the MFSs and XSSs of a failing RDF query. We present our experimental evaluation in Sect. 5 and conclude in Sect. 6.

2 Preliminaries and Problem Statement

This section formally describes the parts of RDF and SPARQL that are necessary to this paper. We use the notations and definitions given in [11].

Data Model. 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* stores a set of RDF triples in a triples table or one of its variants.

Query. 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: $Q = t_1 \wedge \cdots \wedge t_n$. The number of triple patterns of a query Q is denoted by $|Q|$.

Query Evaluation. A *mapping* μ from V to T is a partial function $\mu : V \rightarrow T$. For a triple pattern t, we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are *compatible* when for all $x \in dom(\mu_1) \cap dom(\mu_2)$, it is the case that $\mu_1(x) = \mu_2(x)$ i.e., when $\mu_1 \cup \mu_2$ is also a mapping. Let Ω_1 and Ω_2 be sets of mappings, we define the *join* of Ω_1 and Ω_2 as: $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \textit{ are compatible mappings}\}$. Let D be an RDF database, t a triple pattern. The evaluation of the triple pattern t over D denoted by $[[t]]_D$ is defined by: $[[t]]_D = \{\mu \mid dom(\mu) = var(t) \wedge \mu(t) \in D\}$. Let Q be a query, the evaluation of Q over D is defined by: $[[Q]]_D = [[t_1]]_D \bowtie \cdots \bowtie [[t_n]]_D$. This evaluation can be done under different entailment regimes as defined in the SPARQL specification. In this paper, the examples as well as our implementation are based on the simple entailment regime. However, the proposed algorithms can be used with any entailment regime.

MFS and XSS. 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 $\{i, \cdots, j\} \subseteq \{1, \cdots, n\}$. If $\{i, \cdots, j\} \subset \{1, \cdots, n\}$, we say that Q' is a *proper subquery* of Q ($Q' \subset Q$). If a subquery Q' of Q fails, then the query Q fails.

A *Minimal Failing Subquery MFS* of a query Q is defined as follows: $[[MFS]]_D = \emptyset \ \wedge \nexists\, Q' \subset MFS \textit{ such that } [[Q']]_D = \emptyset$. The set of all MFSs of a query Q is denoted by $mfs(Q)$. Each MFS is a minimal part of the query that fails.

A *Maximal Succeeding Subquery XSS* of a query Q is defined as follows: $[[XSS]]_D \neq \emptyset \ \wedge \nexists\, Q' \textit{ such that } XSS \subset Q' \wedge [[Q']]_D \neq \emptyset$. The set of all XSSs of a query Q is denoted by $xss(Q)$. Each XSS is a maximal (in terms of triple patterns) non-failing subquery viewed as a relaxed query.

Problem Statement. We are concerned with computing the MFSs and XSSs of a failing RDF query over an RDF database efficiently.

3 Lattice-Based Approach (LBA)

LBA is an algorithm to compute simultaneously both the sets $mfs(Q)$ and $xss(Q)$ of a failing RDF query Q. It is a three steps procedure: (1) find an MFS of Q, (2) compute the potential XSSs, i.e., the maximal queries that do not include the MFS previously found and (3) execute potential XSSs; if they return results, they are XSSs, else this process will be applied recursively on failing potential XSSs.

Finding an MFS. This step is performed with the *a_mel_fast* algorithm proposed in [9]. This algorithm is based on the following proposition (proved in [9]). Let $Q = t_1 \wedge ... \wedge t_n$ be a failing query and $Q_i = Q - t_i$ a proper subquery of Q. If $[[Q]]_D = \emptyset$ and $[[Q_i]]_D \neq \emptyset$ then any MFS of Q contains t_i.

This property is leveraged in the Algorithm 1 to find an MFS in n steps (i.e., its complexity is then $\mathcal{O}(n)$). The algorithm removes a triple pattern t_i from Q resulting in the proper subquery Q'. If $[[Q']]_D$ is not empty, t_i is part of any MFS (thanks to the previous proposition) and it is added to the result Q^*. Else, Q' has an MFS that does not contain t_i. Then, the algorithm iterates over another triple pattern of Q to find an MFS in $Q' \wedge Q^*$. This process stops when all the triple patterns of Q have been processed.

Algorithm 1. Find an MFS of a failing SPARQL query Q

```
FindAnMFS($Q$, $D$)
    inputs : A failing query $Q = t_1 \wedge ... \wedge t_n$; an RDF database $D$
    output: An MFS of $Q$ denoted by $Q^*$
    $Q^* \leftarrow \emptyset$;
    $Q' \leftarrow Q$;
    foreach triple pattern $t_i \in Q$ do
        $Q' \leftarrow Q' - t_i$;
        if $[[Q' \wedge Q^*]]_D \neq \emptyset$ then
            $Q^* \leftarrow Q^* \wedge t_i$;
    return $Q^*$;
```

Figure 1 shows an execution of the Algorithm 1 to compute an MFS of the following query $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$:

```
Select ?X ?Y Where {
    ?Y ub:subOrganizationOf <http://www.University8.edu> .  (t1)
    ?X ub:researchInterest "Research28" .                  (t2)
    ?X rdf:type ub:Lecturer .                              (t3)
    ?X ub:worksFor ?Y }                                    (t4)
```

The algorithm removes the triple pattern t_1 from Q, resulting in the subquery Q'. As this subquery returns an empty result, the algorithm iterates over the triple pattern t_2 to find an MFS in $t_2 \wedge t_3 \wedge t_4$. The subquery $t_3 \wedge t_4$ is successful, hence t_2 is part of the MFS Q^*. The same result is obtained for t_3, which is added to Q^*. For t_4, the subquery $t_2 \wedge t_3$ returns an empty result and thus t_4 does not belong to Q^*. As all the triple patterns of Q have been processed, the algorithm stops and returns the MFS $Q^* = t_2 \wedge t_3$.

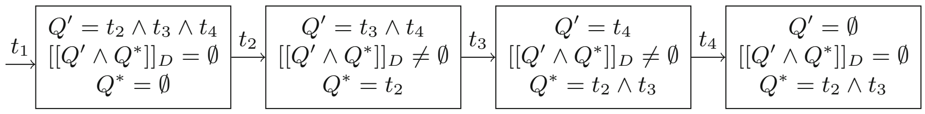
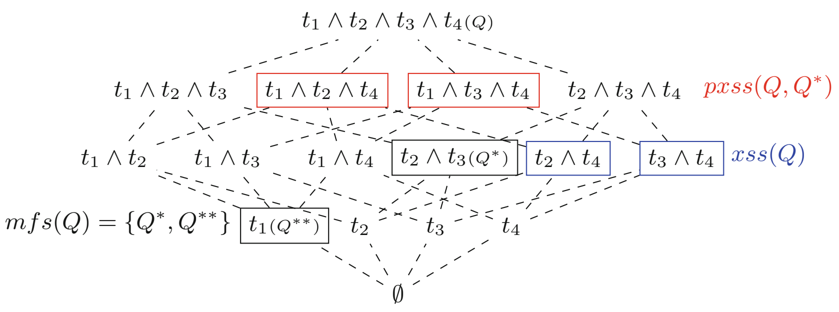

t_1
$Q' = t_2 \wedge t_3 \wedge t_4$
$[[Q' \wedge Q^*]]_D = \emptyset$
$Q^* = \emptyset$
t_2
$Q' = t_3 \wedge t_4$
$[[Q' \wedge Q^*]]_D \neq \emptyset$
$Q^* = t_2$
t_3
$Q' = t_4$
$[[Q' \wedge Q^*]]_D \neq \emptyset$
$Q^* = t_2 \wedge t_3$
t_4
$Q' = \emptyset$
$[[Q' \wedge Q^*]]_D = \emptyset$
$Q^* = t_2 \wedge t_3$

Fig. 1. An execution of Algorithm 1 to find an MFS of Q

Computing Potential XSSs. By definition, all queries that include the MFS Q^*, found in the previous step, return an empty set of answers. Thus, they can be neither MFS nor XSS of Q and they are pruned from the search space. The exploration of the subquery lattice continues with the largest subqueries of Q that do not include Q^*. If these subqueries are successful, they are XSSs of Q. Thus, we call them *potential* XSSs and we denote this set of queries by $pxss(Q, Q^*)$. This set can be computed as follows:

$$pxss(Q, Q^*) = \begin{cases} \emptyset, & \text{if } |Q| = 1. \\ \{Q - t_i \mid t_i \in Q^*\}, & \text{otherwise.} \end{cases}$$



$t_1 \wedge t_2 \wedge t_3 \wedge t_{4(Q)}$
$t_1 \wedge t_2 \wedge t_3$
$t_1 \wedge t_2 \wedge t_4$
$t_1 \wedge t_3 \wedge t_4$
$t_2 \wedge t_3 \wedge t_4$
$pxss(Q, Q^*)$
$t_1 \wedge t_2$
$t_1 \wedge t_3$
$t_1 \wedge t_4$
$t_2 \wedge t_{3(Q^*)}$
$t_2 \wedge t_4$
$t_3 \wedge t_4$
$xss(Q)$
$mfs(Q) = \{Q^*, Q^{**}\}$
$t_{1(Q^{**})}$
t_2
t_3
t_4
$\emptyset$

Fig. 2. The lattice of subqueries of Q with its MFSs and XSSs

Indeed, for each triple pattern t_i of Q^*, a subquery of the form $Q_m \leftarrow Q - t_i$ does not include Q^* and, in addition, it is maximal due to its large size, i.e., $|Q_m| = |Q| - 1$. Following the previous definition, $pxss(Q, Q^*)$ is computed with a simple algorithm running in linear time ($\mathcal{O}(n^*)$ where $n^* = |Q^*|$).

Figure 2 illustrates $pxss(Q, Q^*)$ of our running example ($Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$ and $Q^* = t_2 \wedge t_3$) on the lattice of subqueries. The maximal subqueries of Q that do not contain $t_2 \wedge t_3$ are $t_1 \wedge t_2 \wedge t_4$ and $t_1 \wedge t_3 \wedge t_4$.

Algorithm 2. Find the MFSs and XSSs of a query Q

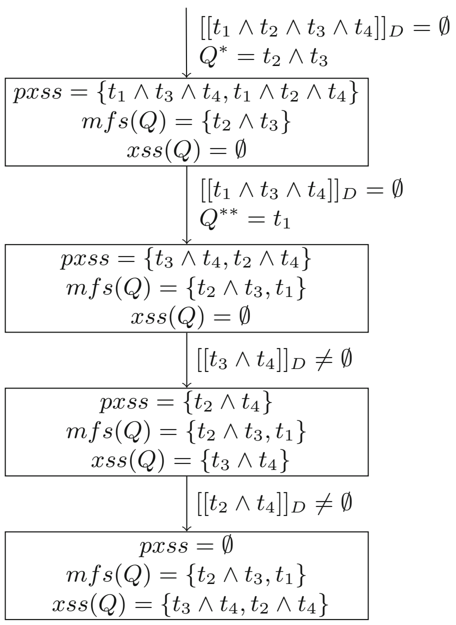
```
LBA(Q, D)
    inputs  : A failing query $Q = t_1 \wedge ... \wedge t_n$; an RDF database $D$
    outputs: The MFSs and XSSs of $Q$
    $Q^* \leftarrow FindAnMFS(Q, D)$;
    $pxss \leftarrow pxss(Q, Q^*)$;
    $mfs(Q) \leftarrow \{Q^*\}$; $xss(Q) \leftarrow \emptyset$;
    while $pxss \neq \emptyset$ do
        $Q' \leftarrow pxss.element()$; /* choose an element of $Q'$ */
        if $[[Q']]_D \neq \emptyset$ then /* $Q'$ is an XSS */
            $xss(Q) \leftarrow xss(Q) \cup \{Q'\}$;
            $pxss \leftarrow pxss - \{Q'\}$;
        else /* $Q'$ contains an MFS */
            $Q^{**} \leftarrow FindAnMFS(Q', D)$;
            $mfs(Q) \leftarrow mfs(Q) \cup \{Q^{**}\}$;
            foreach $Q'' \in pxss$ such that $Q^{**} \subseteq Q''$ do
                $pxss \leftarrow pxss - \{Q''\}$;
                $pxss \leftarrow pxss \cup \{Q_j \in pxss(Q'', Q^{**}) \mid \nexists Q_k \in pxss : Q_j \subseteq Q_k\}$;
    return $\{mfs(Q), xss(Q)\}$;
```

Finding all XSSs and MFSs (Algorithm 2). If Q has only a single MFS Q^* (which includes the case where Q is itself an MFS), then $xss(Q) = pxss(Q, Q^*)$.

Proof. Assume that $\exists Q' \in pxss(Q, Q^*)$ such that $[[Q']]_D = \emptyset$. Since Q has a single MFS, Q^* is a subset of Q'. Contradiction with the definition of $pxss(Q, Q^*)$.

We now consider the general case, i.e., when Q has several MFSs. For each query $Q' \in pxss(Q, Q^*)$, if $[[Q']]_D \neq \emptyset$ then Q' is an effective XSS of Q, i.e., $Q' \in xss(Q)$. Otherwise, Q' has (at least) an MFS, which is also an MFS of Q, different from Q^*. This MFS can be identified with the *FindAnMFS* algorithm (see Algorithm 1) and thus the complete process can be recursively applied on each failing query of $pxss(Q, Q^*)$. However, as different queries of $pxss(Q, Q^*)$ may contain the same MFS, this process may identify the same MFS several times and thus be inefficient. Algorithm 2 improves this approach by incrementally computing potential XSSs that do not contain the set of identified MFSs. When a second MFS Q^{**} is identified, this algorithm iterates over the previously found potential XSSs $pxss$ that contain Q^{**}. To avoid finding again this MFS, the algorithm replaces them by their largest subqueries that do not contain Q^{**} (i.e., their own potential XSSs) and are not included in any query of $pxss$ (otherwise they are not the largest potential XSSs of Q).

Figure 3 shows an execution of Algorithm 2 to compute the MFSs and XSSs of our running example: $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$, $Q^* = t_2 \wedge t_3$ and $pxss(Q, Q^*) = \{t_1 \wedge t_2 \wedge t_4,\ t_1 \wedge t_3 \wedge t_4\}$. The algorithm executes the query $t_1 \wedge t_3 \wedge t_4$. As an empty set of answers is obtained, the Algorithm 1 is applied on this query to find a second MFS $Q^{**} = t_1$. The two potential XSSs contain this MFS and thus



$[[t_1 \wedge t_2 \wedge t_3 \wedge t_4]]_D = \emptyset$
$Q^* = t_2 \wedge t_3$
$pxss = \{t_1 \wedge t_3 \wedge t_4, t_1 \wedge t_2 \wedge t_4\}$
$mfs(Q) = \{t_2 \wedge t_3\}$
$xss(Q) = \emptyset$
$[[t_1 \wedge t_3 \wedge t_4]]_D = \emptyset$
$Q^{**} = t_1$
$pxss = \{t_3 \wedge t_4, t_2 \wedge t_4\}$
$mfs(Q) = \{t_2 \wedge t_3, t_1\}$
$xss(Q) = \emptyset$
$[[t_3 \wedge t_4]]_D \neq \emptyset$
$pxss = \{t_2 \wedge t_4\}$
$mfs(Q) = \{t_2 \wedge t_3, t_1\}$
$xss(Q) = \{t_3 \wedge t_4\}$
$[[t_2 \wedge t_4]]_D \neq \emptyset$
$pxss = \emptyset$
$mfs(Q) = \{t_2 \wedge t_3, t_1\}$
$xss(Q) = \{t_3 \wedge t_4, t_2 \wedge t_4\}$

Fig. 3. An execution of Algorithm 2 to find the MFSs and XSSs of Q

they are replaced with their largest subqueries that do not contain Q^{**}, i.e., $t_3 \wedge t_4$ and $t_2 \wedge t_4$. By executing these two queries, the algorithm finds that these potential XSSs are effectively XSSs. The algorithm stops and returns these two XSSs and the MFSs previously found (see Fig. 2).

4 Matrix-Based Approach (MBA)

In the approach proposed in the previous section, the theoretical search space exponentially increases with the number of triple patterns of the original query. Jannach [10] has proposed a solution to avoid this problem in the context of recommender systems. This approach is based on a matrix, called the *relaxed matrix*, computed in a preprocessing step with n queries where n is the number of query atoms. This matrix gives, for each potential solution of a query, the set of query atoms satisfied by this solution. The XSSs of the query can then be obtained from this matrix without the need for further database queries.

In this section, we adapt this approach to RDF databases to compute both the XSSs and MFSs of a query. Compared to [10], the main difficulty is to compute the set of potential solutions of a query. Indeed, in the context of recommender systems, these solutions are already known as they are the set of products described in the product catalog. This is not the case in the context of RDF databases.

The Relaxed Matrix of a Query. We first informally define the notion of relaxed matrix through an example. Figure 4(c) presents the relaxed matrix of the query Q given in Fig. 4(b) when it is executed on the RDF dataset presented in Fig. 4(a). Each row of the matrix is a mapping (as defined in Sect. 2) that

Triples		
s	p	o
p1	type	Prof
p1	advises	s4
p2	type	Prof
p2	advises	s1
s1	age	27
p3	type	AssistantProf
p3	advises	s2
s2	age	25
s3	age	25

(a) RDF triples

t1 = (?p type Prof) ^
t2 = (?p advises ?s) ^
t3 = (?s age 25)

(b) The query Q

	?p	?s	t1	t2	t3
	p1	NULL	1	0	0
	p2	NULL	1	0	0
*	p1	s4	1	1	0
*	p2	s1	1	1	0
*	p3	s2	0	1	1
*	p1	s2	1	0	1
*	p1	s3	1	0	1
*	p2	s2	1	0	1
*	p2	s3	1	0	1
	NULL	s2	0	0	1
	NULL	s3	0	0	1

(c) The relaxed matrix of Q

xss(Q) = { t1^t3, t2^t3, t1^t2 }
mfs(Q) = { t1^t2^t3 }

(d) XSSs and MFSs of Q

Fig. 4. Matrix-based approach

satisfies at least one triple pattern. For example, the first row corresponds to the mapping $\mu : ?p \rightarrow p_1$. A mapping μ has the value 1 in the column t_i, if μ satisfies t_i. Thus the matrix entry that lies in the first row and the t_1 column is set to 1 as p_1 is a professor in the considered `RDF` dataset.

As we have seen in Sect. 2, the evaluation of a query consists in finding the mappings that satisfy *all its triple patterns* using join operations. The relaxed matrix contains the mappings that satisfy *at least one triple pattern*. Intuitively, one can think of using the `OPTIONAL` operator of `SPARQL` to compute these mappings. However, the semantics of this operator is based on the outer join operation [11], which eliminates from its operands the mappings that satisfy the inner join operation [12]. In our case, we need to keep these mappings as they may be compatible with the mappings of another triple pattern. For example, the operation $[[t_1]]_D ⟗ [[t_2]]_D$ eliminates the mapping $\mu : ?p \rightarrow p_1$ from the relaxed matrix in the example presented in Fig. 4. This mapping is needed to find other mappings such as $\mu : ?p \rightarrow p_1 \; ?s \rightarrow s_2$. As a consequence, we have defined an *extended join operation*, which is defined as follows.

Formal Definition of the Relaxed Matrix. Let Ω_1 and Ω_2 be sets of mappings, the *extended join* of Ω_1 and Ω_2 is defined by: $\Omega_1 ⟗^* \Omega_2 = \Omega_1 \cup (\Omega_1 \bowtie \Omega_2) \cup \Omega_2$. Let Q be a query, the *relaxed evaluation* of Q over D is defined by: $[[Q]]_D^R = [[t_1]]_D ⟗^* \cdots ⟗^* [[t_n]]_D$. We define the relaxed matrix M of a query Q over an `RDF` database D as a two-dimensional table with $|Q|$ columns (one for each triple pattern of the query) and $|[[Q]]_D^R|$ rows (one for each mapping of $[[Q]]_D^R$). For a mapping $\mu \in [[Q]]_D^R$ and a triple pattern $t_i \in Q$, $M[\mu][t_i] = 1 \Leftrightarrow \mu(t_i) \in D, else\ M[\mu][t_i] = 0$.

Computing the Relaxed Matrix. Thus, to obtain the relaxed matrix, we need first to evaluate each triple pattern t_i over D to obtain $[[t_i]]_D$. Then, we compute the extended joins of all the $[[t_i]]_D$ while keeping track of the matched triple patterns to get the matrix values. The Algorithm 3 follows this approach using a nested loop algorithm. This algorithm only requires n queries where n is the number of triple patterns. Yet, our experiments conducted on the `LUBM` benchmark (see Sect. 5) show that this algorithm can still take a notable amount of time as the size of the matrix can be large for queries over large datasets involving triple patterns that are not selective. Moreover, proper subqueries of the initial query can lead to Cartesian products (the triple patterns do not share any variable), which imply an expensive computation cost as well as a matrix of a large size (see Sect. 5 for details). As a first step to improve this approach, we have specialized this approach for star-shaped queries (i.e., a set of triple patterns with a shared join variable in the subject position) as they are often found in the query logs of real datasets [13].

Optimized Computation for Star-Shaped Queries. The computation of star-shaped queries is simpler than in the general case. First, subqueries of a star-shaped query cannot be Cartesian products. Second, a single variable is used to join all the triple patterns. Thanks to this latter property we can use full outer joins to compute the relaxed matrix as depicted in the Algorithm 4. This algorithm executes one query for each triple pattern. For each result μ of such a

Algorithm 3. Computation of the relaxed matrix of a query Q

```
ComputeMatrix($Q$, $D$)
    inputs : A failing query $Q = t_1 \wedge ... \wedge t_n$; an RDF database $D$
    output: The relaxed matrix $M$
    $M \leftarrow \emptyset$;
    foreach triple pattern $t_i \in Q$ do
        foreach $\mu \in [[t_i]]_D$ do
            $isInserted \leftarrow false$;
            foreach $\mu' \in M$ do
                if $\mu$ and $\mu'$ are compatible then
                    if $(\mu' \cup \mu) \notin M$ then
                        $M \leftarrow M \cup \{\mu' \cup \mu\}$;
                        $M[\mu' \cup \mu][t_k] \leftarrow M[\mu'][t_k]$ for $k \in 1 \cdots n \wedge k \neq i$;
                    $M[\mu' \cup \mu][t_i] \leftarrow 1$;
                    if $(\mu \cup \mu') = \mu$ then
                        $isInserted \leftarrow true$;
            if not $isInserted$ then
                $M \leftarrow M \cup \{\mu\}$;
                $M[\mu][t_k] \leftarrow 1$ if $k = i$, else 0; $(k \in 1 \cdots n)$
    return $M$;
```

subquery, the value of the join variable (i.e., the restriction of the function μ to $\{x\}$ denoted by $\mu_{|\{x\}}$) is added to the matrix, if it is not already in it, and the value of this row is set to 1 for the corresponding triple pattern.

The Algorithm 4, called *NQ*, can be used for any RDF database (implemented on a relational database management system (RDBMS) or not). If we consider an RDF database implemented as a triples table $t(s, p, o)$ in an RDBMS, we can use a single SQL query to compute the relaxed matrix. This query is roughly the translation of the $[[t_1]]_D ⟗ \cdots ⟗ [[t_n]]_D$ expression. Inspired by the work of Cyganiak conducted on the translation of SPARQL queries into SQL [14], we use SQL outer join operators to compute this expression and the *coalesce* function[1] to manage unbound values. In addition, we use the *case* operator to test if a triple pattern is matched and thus to get the matrix values (1 if it is matched, else 0). For example, the SQL query used to compute the relaxed matrix of the query $t_1 \wedge t_2$ (Fig. 4) is:

```
select coalesce(t1.s , t2.s),
       case when t1.s is null then 0 else 1 end as t1,
       case when t2.s is null then 0 else 1 end as t2
from (select distinct s from t where p=‘type’ and o=‘professor’) t1
full join (select distinct s from t where p=’advises’) t2 on t1.s = t2.s
```

This approach, called *1Q*, has two advantages: (1) a single query is used to compute the relaxed matrix, (2) the RDBMS chooses the adequate join algorithm.

[1] The *coalesce* function returns the first non-null expression in the list of parameters.

Algorithm 4. Computation of the matrix for star-shaped queries (NQ)

```
ComputeMatrixStarQueryNQ($Q$, $D$)
    inputs : A failing star-shaped query $Q = t_1 \wedge ... \wedge t_n$ with $x$ as join variable;
             An RDF database $D$
    output: The relaxed matrix $M$
    $M \leftarrow \emptyset$;
    foreach triple pattern $t_i \in Q$ do
        foreach $\mu \in [[t_i]]_D$ do
            if $\mu_{|\{x\}} \notin M$ then
                $M \leftarrow M \cup \{\mu_{|\{x\}}\}$;
                $M[\mu_{|\{x\}}][t_k] \leftarrow 0$ for $k \in 1 \cdots n \wedge k \neq i$;
            $M[\mu_{|\{x\}}][t_i] \leftarrow 1$;
    return $M$;
```

Computing the XSSs from the Relaxed Matrix. Abusing notation, we denote by $xss(\mu)$ the proper subquery of Q that can be executed to retrieve μ. It can be directly obtained from the relaxed matrix: $xss(\mu) = \{t_i \mid M[\mu][t_i] = 1\}$. Finding the XSSs of a query Q can be done in two steps:

1. Computing the skyline SKY of the relaxed matrix: $SKY(M) = \{\mu \in [[Q]]_D^R \mid \nexists \mu' \in [[Q]]_D^R \textit{ such that } \mu \prec \mu'\}$ where $\mu \prec \mu'$ if (i) on every triple pattern t_i, $M[\mu][t_i] \leq M[\mu'][t_i]$ and (ii) on at least one triple pattern t_j, $M[\mu][t_j] < M[\mu'][t_j]$. This step can be done by using one of the numerous algorithms defined to efficiently compute the skyline of a table (see [15] for a survey). In Fig. 4(c), all the rows composing the skyline of the relaxed matrix are marked with $*$.
2. Retrieving the distinct proper subqueries of Q that can be executed to retrieve an element of the skyline: $xss(Q) = \{xss(\mu) \mid \mu \in SKY(M)\}$. Each such proper subquery is an XSS. The XSSs of our example are given in Fig. 4(d) and appear in bold in the relaxed matrix.

Using the Relaxed Matrix as an Index for the LBA Approach. In the LBA algorithm, subqueries are executed on the RDF database to find whether they return an empty set of answers or not. Instead of executing a subquery, one can compute the intersection of the matrix columns corresponding to the subquery triple patterns. If the resulting column is empty, the subquery returns an empty set of answers and conversely. Thus, the MBA approach can be seen as an index to improve the performance of the LBA approach. This approach still requires exploring a search space that exponentially increases with the number of triple patterns, but this search space does not require the execution of any database query.

5 Experimental Evaluation

Experimental Setup. We have implemented the proposed algorithms in JAVA 1.7 64 bits on top of Jena TDB. Our implementation is available at http://www.lias-lab.fr/forge/projects/qars. These algorithms take as input a failing SPARQL query and return the set of MFSs and XSSs of this query. We run these algorithms on a Windows 7 Pro system with Intel Core i7 CPU and 8 GB RAM. All times presented in this paper are the average of five runs of the algorithms. The results of algorithms are not shown for queries when they consumed too many resources i.e., when they took more than one hour to execute or when the memory used exceeded the size of the JVM (set to 4 GB in our experiments).

Dataset and Queries. Due to the lack of an RDF query relaxation benchmark, Huang et al. [6] have designed 7 queries based on the LUBM benchmark. These queries cover the main query patterns (star, chain and composite) but they only have between 2 and 5 triple patterns. Yet the study proposed by Arias and al. [13] has shown that real-world SPARQL queries executed on the DBPedia and SWDF datasets range from 1 to 15 triple patterns. As a consequence, we have modified the 7 queries proposed in [6] to reflect this diversity. The modified versions of these queries[2] have respectively 1, 5, 7, 9, 11, 13 and 15 triple patterns. Q1, Q2 are chain, Q3, Q5, Q7 are star and Q4, Q6 are composite query patterns. We used two generated datasets to evaluate the performances of our algorithms on these queries: LUBM20 (3 M triples) and LUBM100 (13 M triples).

Relaxed Matrix Size and Computation Time. The MBA approach relies on the relaxed matrix. To define the data structure of this matrix, we have leveraged the similarity between this matrix and bitmap indexes used in RDBMS. Thus, the matrix is defined as a set of compressed bitmaps, one for each column. We have used the Roaring bitmap library version 0.4.8 for this purpose [16]. As Table 1 shows, this data structure ensures that the matrix size remains small even if the number of matrix rows is large (less than 2 MB for 2 M rows). Table 1 only includes results for star-shaped queries as other queries required too many resources due to Cartesian products.

For the computation of the MBA relaxed matrix, we have tested the two algorithms *1Q* and *NQ* described in Sect. 4. As the *1Q* approach requires an RDF

Table 1. Relaxed matrix properties

	LUBM20			LUBM100		
	Q3	Q5	Q7	Q3	Q5	Q7
Computation time with NQ (in sec)	8.6	8.6	8.6	42.6	43.4	44.6
Computation time with 1Q (in sec)	6.1	6.3	6.8	30.4	34.6	38.5
Size (in KB)	293	400	335	1385	1912	1590
Number of rows (in K)	430	430	430	2149	2149	2149

[2] Available at http://www.lias-lab.fr/publications/16873/Report_MFS_XSS.pdf.

database implemented on top of an RDBMS, we have used the Oracle 12c RDBMS to implement the triples table and test this algorithm. As Table 1 shows, the *1Q* algorithm is about 25 % faster than *NQ*. Even with this optimization, which is only possible for specific RDF databases, the computation time of the matrix is important: around 6 s on LUBM20 and 35 s on LUBM100. Despite this important computation time, the MBA approach can still be interesting as the matrix can be precomputed for usual failing queries identified with query logs. Moreover, the next experiment shows that MBA is faster than other algorithms for large queries even if the matrix is computed at runtime.

XSS and MFS Computation Time. We compare the performance of the following algorithms for computing the XSSs and MFSs of the benchmark queries.

- LBA: the algorithm described in Sect. 3.
- MBA+M: this algorithm first computes the relaxed matrix using Algorithm 4 for star-shaped queries and Algorithm 3 for other queries. Then, it computes XSSs and MFSs of the query with the LBA algorithm that uses the relaxed matrix instead of executing queries.
- MBA-M: same as MBA+M but without the computation of the relaxed matrix.
- DFS: a depth-first search algorithm of the subquery lattice that we modified to prune the search space when no more MFSs and XSSs can be found.
- ISHMAEL: the algorithm proposed in [9] that we have tailored to return both the XSSs and MFSs of a query.

Figure 5 shows the performance of each algorithm displayed in logarithmic scale for readability. An algorithm that evaluates most of the subqueries such as DFS can be used for queries with only a few triple patterns (Q1 and Q2). For larger queries, the number of subqueries exponentially increases and thus the performance of DFS quickly decreases.

In this case, the smart exploration of the search space provided by the LBA and ISHMAEL algorithms is more efficient. Their response times are between

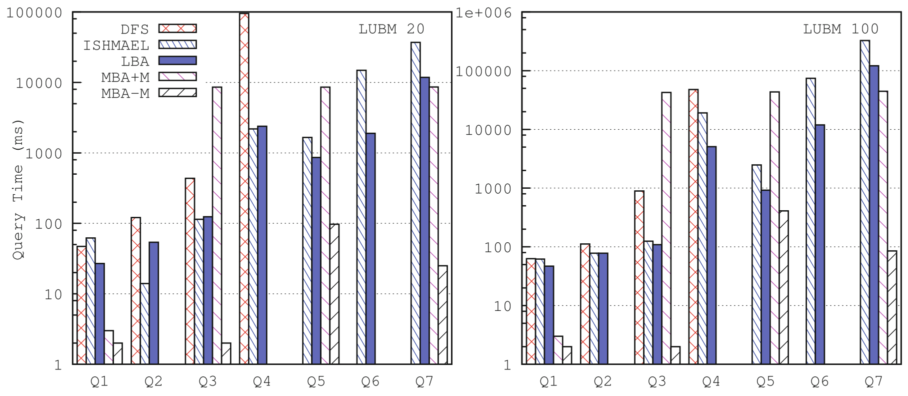


Fig. 5. Performance of the algorithms on LUBM20 and LUBM100

1 and 10 seconds for queries that do not have more than 11 triple patterns (Q1-Q5). The performance of LBA and ISHMAEL are close for queries Q1-Q5 and LBA outperforms ISHMAEL on Q6 and Q7 (recall that the results are presented in logarithmic scale). We have identified that the performance difference is due to the simplified computation of the potential XSSs and to the order in which these potential XSSs are evaluated. Indeed, according to this order, the caching performed by Jena TDB can be more or less efficient. For example, we have found some cases where the same query can be executed with a response time differing by a factor of 2 according to the caching usage. Thus, a perspective is to find the best ordering of the potential XSSs to maximize the cache usage.

Finally, the MBA approach can only be used for star-shaped queries. $\texttt{MBA}-\texttt{M}$ provides response times of some milliseconds even for Q7, which has 15 triple patterns. This is due to the fact that this approach just needs to compute the intersection of bitmaps using bitwise operations instead of executing subqueries. However, this approach makes a strong assumption: the matrix must be precomputed i.e., the query must have been identified as a usual failing query (e.g., using query logs). If the matrix is computed at runtime ($\texttt{MBA}+\texttt{M}$), this computation time is important (see Table 1) and thus $\texttt{MBA}+\texttt{M}$ is only interesting for queries with a large number of triple patterns or with only selective triple patterns (they can be identified using database statistics). As a consequence, the MBA approach is complementary with an approach such as LBA: it should be used when LBA does not scale anymore.

Performance as the Number of Triple Patterns Scales. The previous experiments show that the number of triple patterns plays an important role in the performance of the proposed algorithms. In order to explore this further, we have decomposed Q7 in 15 subqueries ranging from 1 to 15 triple patterns. The first subquery only includes the first triple pattern of Q7, the second subquery includes the first two triple patterns and so on. The result of this experiment is shown in Fig. 6. This experiment confirms our previous observation. DFS does not scale when a query exceeds 5 triple patterns. LBA and ISHMAEL can be used with a response time between 1 and 10 s for queries with less than 13 triple

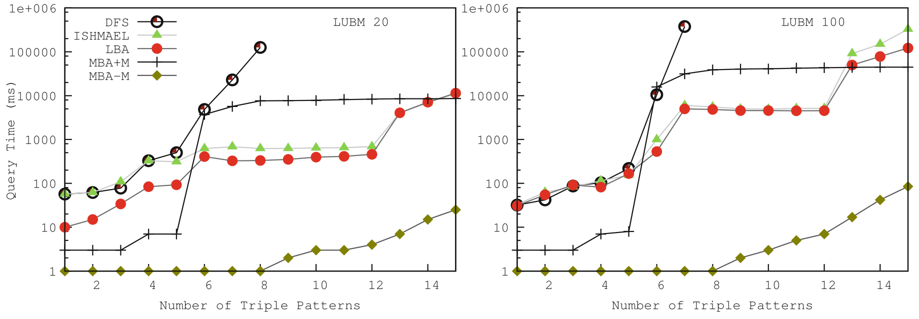


Fig. 6. Query 7 performance as the number of its triple patterns increases

patterns. The $\texttt{MBA} - \texttt{M}$ scales well for star queries even with a large number of triple patterns. The $\texttt{MBA} + \texttt{M}$ is only interesting when the query has more than 13 triple patterns as the cost of computing the matrix is important.

6 Related Work

We review here the closest works related to our proposal done both in the context of RDF and relational databases. In the first setting, Hurtado et al. [5] proposed some rules and operators for relaxing RDF queries. Adding to these rules, Huang et al. [6] specified a method for relaxing SPARQL queries using a semantic similarity measure based on statistics. In our previous work [7], 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. [8] have also extended a fragment of this language with query approximation and relaxation operators. As an alternative to query relaxation, there have been works on *query auto-completion* (e.g., [17]), which check the data during query formulation to avoid empty answers. But, none of the previous works has considered the issue related to the causes of RDF query failure and then the issue of MFS computation.

As for relational databases, many works have been proposed for query relaxation (see Bosc et al. [18] for an overview). In particular, Godfrey [9] has defined the algorithmic complexity of the problem of identifying the MFSs of failing relational query and developed the ISHMAEL algorithm for retrieving them. The LBA approach is inspired by this algorithm. Compared with ISHMAEL, LBA computes both the MFSs and the potential XSSs in one time. Moreover, LBA proposes a simplified computation of the potential XSSs. Bosc et al. [18] and Pivert et al. [19] extended Godfrey's approach to the fuzzy query context. Jannach [10] studied the concept of MFS in the recommendation system setting. The MBA approach is inspired by this approach. Contrary to [10], the computation of the matrix rows is not straightforward in the context of RDF queries. Moreover, in [10], the matrix is only used to retrieve the XSSs of the query while, in our work, we used and stored this matrix as a bitmap index to improve the performance of LBA.

7 Conclusion and Discussion

In this paper we have proposed two approaches to efficiently compute the MFSs and XSSs of an RDF query. The first approach, called LBA, is a smart exploration of the subquery lattice of the failing query that leverages the properties of MFS and XSS. The second approach, called MBA, is based on the precomputation of a matrix, which records, for each potential solution of the query, the set of triple patterns that it satisfies. The XSSs of a query can be found without any database access by computing the skyline of this matrix. Interestingly, this matrix looks like a bitmap index and can also improve the performance of the LBA algorithm. We have done a complete implementation of our propositions and evaluated their performances on two datasets generated with the LUBM benchmark. While a straightforward algorithm does not scale for queries with more than 5 triple patterns, the LBA approach scales up to approximatively 11 triple patterns in our experiments. The MBA approach is only interesting for star-shaped queries. If the matrix is precomputed, which assumes that the query has been identified as a usual failing query, XSSs and MFSs can be found in some milliseconds even for queries with many triple patterns (a maximum of 15 in our experiments). If the matrix is computed at runtime, this approach can still be interesting for large queries as the cost of computing the matrix becomes acceptable in comparison with the optimization of LBA it permits. Optimizing the MBA approach for other kinds of RDF query is part of our future work. We also plan to define query relaxation strategies based on the MFSs and XSSs of a failing RDF query.

References

1. Wilkinson, K.: Jena property table implementation. In: SSWS (2006)
2. Dolog, P., Stuckenschmidt, H., Wache, H., Diederich, J.: Relaxing RDF queries based on user and domain preferences. IJIIS **33**(3), 239–260 (2009)
3. Elbassuoni, S., Ramanath, M., Weikum, G.: Query relaxation for entity-relationship search. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part II. LNCS, vol. 6644, pp. 62–76. Springer, Heidelberg (2011)
4. Hogan, A., Mellotte, M., Powell, G., Stampouli, D.: Towards fuzzy query-relaxation for RDF. In: Simperl, E., Cimiano, P., Polleres, A., Corcho, O., Presutti, V. (eds.) ESWC 2012. LNCS, vol. 7295, pp. 687–702. Springer, Heidelberg (2012)
5. Hurtado, C.A., Poulovassilis, A., Wood, P.T.: Query relaxation in RDF. In: Spaccapietra, S. (ed.) Journal on Data Semantics X. LNCS, vol. 4900, pp. 31–61. Springer, Heidelberg (2008)
6. Huang, H., Liu, C., Zhou, X.: Approximating query answering on RDF databases. J. World Wide Web **15**(1), 89–114 (2012)
7. Fokou, G., Jean, S., Hadjali, A.: Endowing semantic query languages with advanced relaxation capabilities. In: Andreasen, T., Christiansen, H., Cubero, J.-C., Raś, Z.W. (eds.) ISMIS 2014. LNCS, vol. 8502, pp. 512–517. Springer, Heidelberg (2014)
8. Calí, A., Frosini, R., Poulovassilis, A., Wood, P.T.: Flexible querying for SPARQL. In: Meersman, R., Panetto, H., Dillon, T., Missikoff, M., Liu, L., Pastor, O., Cuzzocrea, A., Sellis, T. (eds.) OTM 2014. LNCS, vol. 8841, pp. 473–490. Springer, Heidelberg (2014)
9. Godfrey, P.: Minimization in cooperative response to failing database queries. Int. J. Coop. Inf. Syst. **6**(2), 95–149 (1997)
10. Jannach, D.: Fast computation of query relaxations for knowledge-based recommenders. AI Commun. **22**(4), 235–248 (2009)
11. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3), 16:1–16:45 (2009)
12. Galindo-Legaria, C.A.: Algebraic optimization of outerjoin queries. Ph.D thesis, Harvard University, Technical report TR-12-92 (1992)
13. Arias, M., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P.: An empirical study of real-world SPARQL queries. In: USEWOD (2011)
14. Cyganiak, R.: A relational algebra for sparql. HP-Labs, HPL-2005-170 (2005)
15. Hose, K., Vlachou, A.: A survey of skyline processing in highly distributed environments. VLDB J. **21**(3), 359–384 (2012)

16. Chambi, S., Lemire, D., Kaser, O., Godin, R.: Better bitmap performance with roaring bitmaps (2014). arXiv preprint arXiv:1402.6407
17. Campinas, S.: Live SPARQL auto-completion. In: ISWC 2014 (Posters & Demos), pp. 477–480 (2014)
18. Bosc, P., Hadjali, A., Pivert, O.: Incremental controlled relaxation of failing flexible queries. JIIS **33**(3), 261–283 (2009)
19. Pivert, O., Smits, G., Hadjali, A., Jaudoin, H.: Efficient detection of minimal failing subqueries in a fuzzy querying context. In: Eder, J., Bielikova, M., Tjoa, A.M. (eds.) ADBIS 2011. LNCS, vol. 6909, pp. 243–256. Springer, Heidelberg (2011)