The final publication is available at Springer via
http://dx.doi.org/10.1007/s10115-016-0941-0

Handling Failing RDF Queries: From Diagnosis to Relaxation

Géraud Fokou, Stéphane Jean, Allel Hadjali, Mickael Baron

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

Received: Sep 10, 2015; Revised: Jan 22, 2016; Accepted: Mar 06, 2016

Abstract. Recent years have witnessed the development of large Knowledge Bases (KBs). Due to the lack of information about the content and schema semantics of KBs, users are often not able to correctly formulate KB queries that return the intended result. In this paper, we consider the problem of failing RDF queries, i.e. queries that return an empty set of answers. Query relaxation is one cooperative technique proposed to solve this problem. In the context of RDF data, several works proposed query relaxation operators and ranking models for relaxed queries. But none of them tried to find the causes of an RDF query failure given by *Minimal Failing Subqueries* (MFSs) as well as successful queries that have a maximal number of triple patterns named *Maximal Succeeding Subqueries* (XSSs). Inspired by previous work in the context of relational databases and recommender systems, we propose two complementary approaches to fill this gap. The *Lattice-Based Approach* (LBA) leverages the theoretical properties of MFSs and XSSs to efficiently explore the subquery lattice of the failing query. The *Matrix-Based Approach* (MBA) computes a matrix that records alternative answers to the failing query with the triple patterns they satisfy. The skyline of this matrix directly gives the XSSs of the failing query. This matrix can also be used as an index to improve the performance of LBA. The practical interest of these two approaches are shown via a set of experiments conducted on the LUBM benchmark and a comparative study with baseline and related work algorithms.

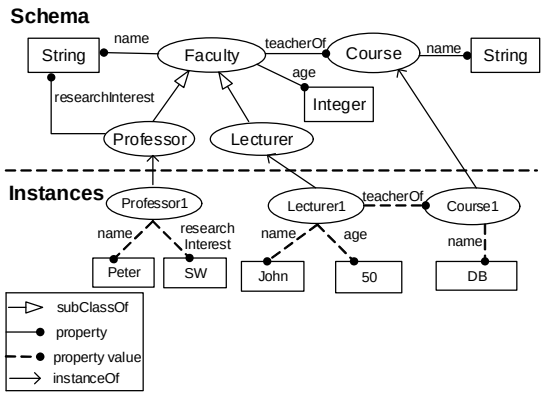
Keywords: Query Relaxation; Knowledge Base; RDF Database; Semantic Web

1 Introduction

A *Knowledge Base* (KB) is a collection of entities and facts about them. In recent years, numerous projects in both industry and academia have been conducted to build large-scale KBs. Well-known examples of KBs include YAGO2 [1] and DBPEDIA [2], resulting from academics projects as well as the KBs designed in commercial projects such as those by Google [3] or Walmart [4]. These KBs are represented in RDF [5] as a set of triples *(subject, predicate, object)* and queried with the SPARQL language [6].

For most users, querying KBs is not an easy task. This difficulty is due to the following three main reasons. First, KBs are often built from a set of heterogeneous data sources and thus the underlying structure of a KB is rarely known by end users. Second, KB schemas (or

ontologies) are defined with a language such as RDFS [7] or OWL [8] whose semantics and underlying assumptions may not be understood by end users. Last, despite their large size (e.g., the FreeBase KB [9] has 1.9B triples and Knowledge Vault from Google has 1.6B triples), KBs are not complete. For example, 60% of the educational institution of the FreeBase KB have no known telephone number, and 95% have no known number of faculty[1]. Hence, when a query does not return any answer, users may not know if the query is too selective or if the intended result is not defined in the KB. Users may try to change the optional parts of the SPARQL query as well as the query conditions. But this is a tedious, frustrating and time-consuming process. This paper addresses this problem of failing RDF queries (i.e., queries that return an empty result) by identifying, on the one hand, the causes of this failure and, on the other hand, some successful relaxed queries.



TRIPLES			
Subject	**Predicate**	**Object**	
Faculty	rdf:type	rdfs:Class	Schema
Professor	rdf:type	rdfs:Class	
Professor	rdfs:subClassOf	Faculty	
name	rdf:type	rdf:Property	
name	rdfs:range	xsd:String	
…	…	…	
Professor1	rdf:type	Professor	Instances
Professor1	name	Peter	
Professor1	researchInterest	SW	
Lecturer1	rdf:type	Lecturer	
…	…	…	

Fig. 1. Running example

As a running example, we consider the KB inspired by the LUBM Benchmark [10] depicted in Figure 1. If a user wants to retrieve the age of lecturers of a database course who are interested in Semantic Web research, she/he may issue the following query[2] $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$:

```
SELECT ?p ?a WHERE {
?p age ?a .                    (t1)
?p type Lecturer .             (t2)
?p researchInterest "SW" .     (t3)
?p teacherOf "DB" }            (t4)
```

This query will fail because the domain of the *researchInterest* property is *Professor* and not *Lecturer*. With a deeper knowledge of the KB schema, the user may be able to correct this query replacing *Lecturer* by *Professor*. But, even in this case, the query may fail again. This will be the case if the age of professors are not known in the KB or if there is no faculty

[1] Numbers current as of February 2015.
[2] For readability, we use names instead of URIs to identify the query elements.

member who both teaches a database course and is interested in Semantic Web research. This query could also fail for many other reasons.

Query relaxation is one of the cooperative techniques used to return alternative answers instead of an empty result. Several approaches proposed to relax queries in the RDF context [11–20]. They introduced relaxation operators based on RDFS semantics (e.g., generalizing triple patterns using class and property hierarchies) [11–17], similarity measures [18, 19] and user preferences [20]. These operators are mainly used to obtain *top-k* answers (where k is a user-defined parameter) [12–14]. Indeed, they are applied to the user query resulting in a set of relaxed queries. These relaxed queries are ordered using a similarity measure and then executed in this ranking order. Other approaches proposed an extension of SPARQL to include relaxation operators in queries [15–17] or query rewriting rules to perform relaxation [20].

In these previous approaches on RDF query relaxation, the relaxation process is *blind* in the sense that it relaxes the user query without knowing the reasons for its failure. As a consequence, it may relax triple patterns that do not need to be modified in the user query and/or takes a long processing time to effectively relax the triple patterns responsible of the query's failure. To avoid this problem, the notion of *Minimal Failing Subquery (MFS)* was introduced in the context of relational databases to find the causes of the query's failure [21]. An MFS is a failing query that does not include a failing subquery. As several MFSs can be responsible of the query's failure, relaxing a failing query requires enumerating all these MFSs, which is an NP-hard problem [21].

Another cooperative technique introduced in the context of relational databases consists in finding relaxed queries, called *Maximal Succeeding Subqueries* (XSSs), that are as close as possible to the failing query semantically speaking. In the context of RDF queries, XSSs stand for non-failing relaxed queries that have a maximal number of triple patterns. Thus, each XSS provides a simple way to relax a failing query by removing or making optional the set of triple patterns that are not in an XSS.

To the best of our knowledge, no work exists in the literature that addresses the issue of computing MFSs and XSSs of failing RDF queries. In the literature on relational databases, one can distinguish two main approaches to compute MFSs and XSSs of a failing query. The first approach consists in exploring the subquery lattice of the failing query [21] while the second one relies on a particular matrix that records alternative answers to the failing query with the query conditions they satisfy [22]. These approaches cannot be directly used to compute MFSs and XSSs of an RDF query as they target a different data model and query language. In particular, as we will see in Section 4, the computation of the matrix is particularly challenging in the context of RDF as SQL and SPARQL differ in their semantics and assumptions [23]. Moreover, these previous approaches were designed to compute either MFSs or XSSs but not both of them at the same time.

Hence, we propose in this paper two algorithmic approaches called *Lattice-Based Approach* (LBA) and *Matrix-Based Approach* (MBA) for the purpose of MFSs and XSSs computation in the RDF context. LBA is an adapted and extended variant of Godfrey's ISHMAEL algorithm [21] that leverages properties of MFSs and XSSs to prune the subquery lattice search space. The main novelty of LBA compared to ISHMAEL is that it computes both MFSs and XSSs of a failing RDF query without an added complexity. MBA is inspired by the work of Jannach in recommender systems [22]. This approach uses a matrix containing alternative answers to the failing query with the triple patterns they satisfy. This matrix is computed using only n

queries over the target RDF database where n is the number of query triple patterns. This matrix can also be computed with only one query if the RDF database is implemented as a triples table. The skyline[3] of this matrix directly gives the XSSs of the query at hand. We also show that this matrix can improve the performance of LBA to find both MFSs and XSSs of an RDF query. The relevance of our propositions are evaluated through a set of experiments conducted on several datasets generated with the LUBM benchmark. They are also compared with a baseline method and an adapted version of the ISHMAEL algorithm.

This paper is an extension of our earlier conference work [24]. We have substantially developed, revised and improved the material presented here. In particular, the following new contributions are made: (i) we investigate a list of properties of our algorithms with their proofs, (ii) we present a full and critical review of existing works and (iii) we perform a thorough experimental evaluation of our proposed algorithms and optimizations on large datasets (ranging from 13M to 130M triples) with a set of generated RDF queries.

The paper is structured as follows. Section 2 introduces some basic notions and formalizes the problem we consider. Section 3 and 4 present our LBA and MBA approaches to find the MFSs and XSSs of a failing RDF query. Section 5 presents our implementation and experimental evaluation of our two approaches on the LUBM benchmark. Section 6 details related work on RDF query relaxation as well as other proposed approaches to address the empty answer problem. Finally, we conclude and introduce future work in Section 7.

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 [23].

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 [25].

RDF queries. 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 $|Q|$.

Query evaluation. A *mapping* μ from V to T is a partial function $\mu : V \to T$. 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 by: $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\}$. Let D be an RDF database and t a triple pattern. The evaluation of 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

[3] Given a set of objects described by a list of criteria. A skyline is a subset of objects that are not dominated (in the sense of Pareto) by any other object with respect to some criteria of interest.

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 $\{t_i, \cdots, t_j\} \subseteq \{t_1, \cdots, t_n\}$. If $\{t_i, \cdots, t_j\} \subset \{t_1, \cdots, t_n\}$, we say that Q' is a *proper subquery* of Q ($Q' \subset Q$). If a subquery Q' of Q fails, then the query Q fails.

Definition 1. *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 $mfs(Q)$. Each MFS is a minimal part of the query that failed.

Definition 2. *A Maximal Succeeding Subquery XSS of a query Q is defined as follows:* $[[XSS]]_D \neq \emptyset \ \wedge \nexists\ Q'$ *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 *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 is applied recursively on failing potential XSSs.

3.1 Finding an MFS

This step is performed with the *a_mel_fast* algorithm proposed in [21]. This algorithm is based on the following proposition (originally proved in [21]).

Proposition 1. *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 .*

Proof. Assume that $\exists MFS \in mfs(Q)$ such that $t_i \notin MFS$. Since $Q_i = Q - t_i$, MFS is a subquery of Q_i. Contradiction: a successful query cannot include a failing query.

This property is leveraged in Algorithm 1 to find an MFS in n steps (i.e., its complexity is of $\mathcal{O}(n)$). The algorithm removes a triple pattern t_i from Q, resulting in the proper subquery Q' (line 4). 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^* (line 6). 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 RDF 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^*

1 $Q^* \leftarrow \emptyset$;
2 $Q' \leftarrow Q$;
3 **foreach** *triple pattern* $t_i \in Q$ **do**
4 $\quad Q' \leftarrow Q' - t_i$;
5 $\quad$ **if** $[[Q' \wedge Q^*]]_D \neq \emptyset$ **then**
6 $\quad\quad Q^* \leftarrow Q^* \wedge t_i$;
7 **return** Q^*;

To illustrate this algorithm, we consider our running example $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$. We assume that this query has two MFSs: t_1 (people's ages are unknown in the KB) and $t_2 \wedge t_3$ (a lecturer cannot have a *researchInterest* in the KB) as well as two XSSs: $t_2 \wedge t_4$ (there are lecturers who teach a *DB* course) and $t_3 \wedge t_4$ (there are professors who both teach a *DB* course and are interested in *SW* research).

Figure 2 shows a possible execution of the Algorithm 1 that finds the MFS $t_2 \wedge t_3$ of our example query Q. The algorithm removes the triple pattern t_1 from Q, resulting in the subquery $Q' = t_2 \wedge t_3 \wedge t_4$. As this subquery returns an empty result, the algorithm searches an MFS in $t_2 \wedge t_3 \wedge t_4$. It removes the triple pattern t_2 from this subquery. The resulting 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 $Q^* = t_2 \wedge t_3$.

Figure 3 shows another possible execution of the Algorithm 1 that finds the MFS t_1 of Q. This time the algorithm removes the triple pattern t_3 from Q. As the query $t_1 \wedge t_2 \wedge t_4$ returns an empty result, t_3 does not belong to Q^*. The algorithm iterates over the triple pattern t_1. The query $t_2 \wedge t_4$ is successful, hence t_1 is added to Q^*. For each other triple pattern, the query $Q' \wedge Q^*$ returns an empty result and thus the MFS t_1 is returned.

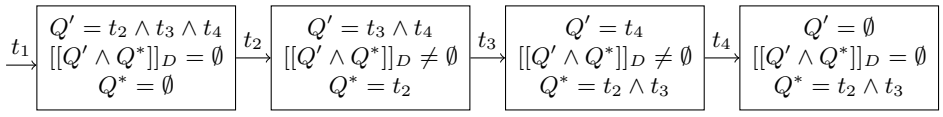


Fig. 2. An example of Algorithm 1 execution that finds the MFS $t_2 \wedge t_3$ of Q

3.2 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 lattice of subqueries continues with the largest

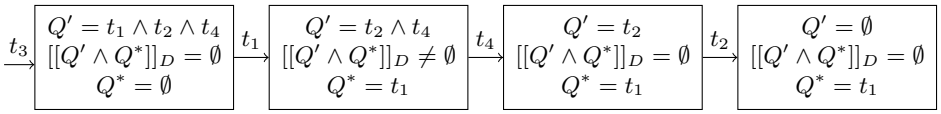


Fig. 3. An example of Algorithm 1 execution that finds the MFS t_1 of Q

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:

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

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 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 4 illustrates $pxss(Q, Q^*)$ and $pxss(Q, Q^{**})$ of our running example ($Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$, $Q^* = t_2 \wedge t_3$ and $Q^{**} = t_1$) 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 $t_1 \wedge t_3 \wedge t_4$. The maximal subquery of Q that does not contain t_1 is $t_2 \wedge t_3 \wedge t_4$.

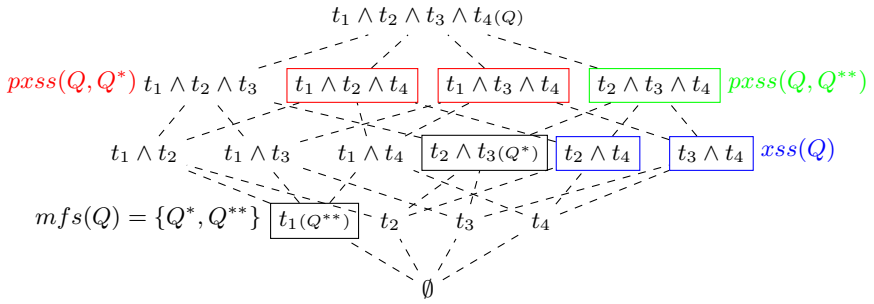


Fig. 4. The lattice of subqueries of Q with its MFSs and XSSs

3.3 Finding All XSSs and MFSs

As the following proposition shows, if Q has only a single MFS (which includes the case where Q is itself an MFS), the MFSs and XSSs of an RDF query can be computed with the two previous steps.

Proposition 2. *If Q has a single MFS Q^*, then $xss(Q) = pxss(Q, Q^*)$.*

Proof. By definition, queries of $pxss(Q, Q^*)$ are maximal. Now, 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 $pxss(Q, Q^*)$.

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
1 $Q^* \leftarrow FindAnMFS(Q, D)$;
2 $pxss \leftarrow pxss(Q, Q^*)$;
3 $mfs(Q) \leftarrow \{Q^*\}$;
4 $xss(Q) \leftarrow \emptyset$;
5 **while** $pxss \neq \emptyset$ **do**
6 $Q' \leftarrow pxss.element()$; `// choose an element of` $pxss$
7 **if** $[[Q']]_D \neq \emptyset$ **then** `//` Q' `is an XSS`
8 $xss(Q) \leftarrow xss(Q) \cup \{Q'\}$;
9 $pxss \leftarrow pxss - \{Q'\}$;
10 **else** `//` Q' `contains an MFS`
11 $Q^{**} \leftarrow FindAnMFS(Q', D)$;
12 $mfs(Q) \leftarrow mfs(Q) \cup \{Q^{**}\}$;
13 **foreach** $Q'' \in pxss$ *such that* $Q^{**} \subseteq Q''$ **do** `// avoid finding again` Q^{**}
14 $pxss \leftarrow pxss - \{Q''\}$;
15 $pxss \leftarrow pxss \cup \{Q_j \in pxss(Q'', Q^{**}) \mid \nexists Q_k \in pxss \cup xss(Q) : Q_j \subseteq Q_k\}$;
16 **return** $\{mfs(Q), xss(Q)\}$;

We now consider the general case, i.e., when Q has more than one MFS. For each query $Q' \in pxss(Q, Q^*)$, if $[[Q']]_D \neq \emptyset$ then Q' is an 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 a previously identified MFS (lines 13-15). Indeed, 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, each such query Q'' is replaced by their largest subqueries that do not contain Q^{**} (i.e., $pxss(Q'', Q^{**})$) and are not included in any query of $pxss$ or $xss(Q)$ (otherwise they are not the largest potential XSSs of Q).

Figure 5 shows two examples of Algorithm 2 executions to compute the MFSs and XSSs of our running example: $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$. In Figure 5(a), the MFS $Q^* = t_2 \wedge t_3$ is first found and the corresponding potential XSSs computed ($pxss(Q, Q^*) = \{t_1 \wedge t_3 \wedge t_4,\ t_1 \wedge t_2 \wedge t_4\}$). 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 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 Figure 4).

Figure 5(b) presents an alternative execution of Algorithm 2 where the MFS t_1 is first found. The corresponding potential XSS is $t_2 \wedge t_3 \wedge t_4$. This query returns an empty result and thus the second MFS $t_2 \wedge t_3$ is found. The query $t_2 \wedge t_3 \wedge t_4$ is replaced by its largest subqueries that do not contain $t_2 \wedge t_3$, i.e., $t_3 \wedge t_4$ and $t_2 \wedge t_4$. From this point, the execution is the same as the one presented in Figure 5(a).

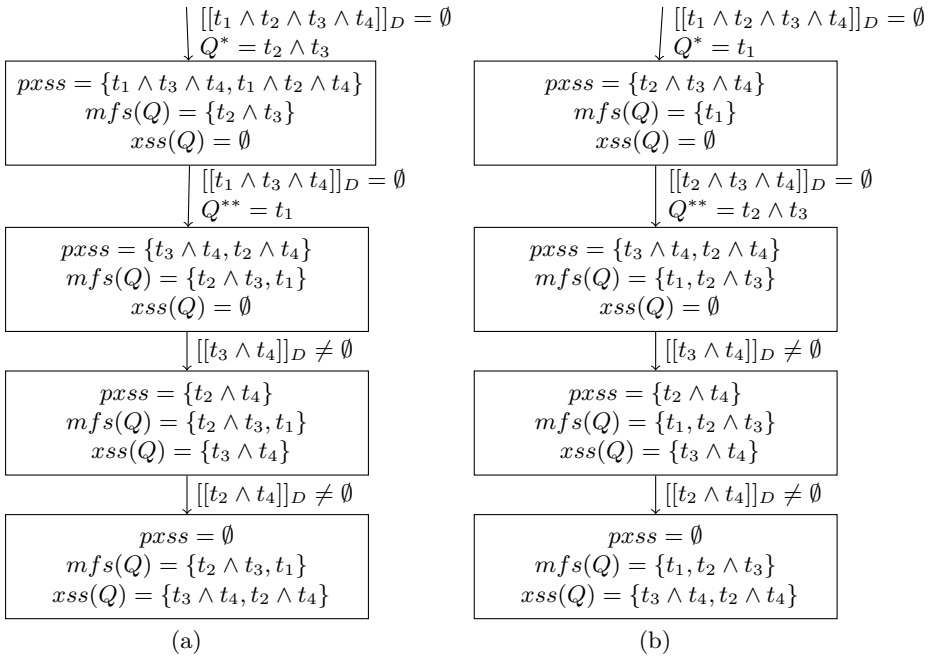


Fig. 5. Two examples of Algorithm 2 executions that find the MFSs and XSSs of Q

Proposition 3. *Algorithm LBA is sound and complete, i.e., it returns exactly all MFSs and XSSs of a failing RDF query Q.*

Proof. We use an induction on the size of Q. If $|Q| = 1$, the Algorithm 2 correctly returns Q as MFS[4] and an empty set of XSSs. Assuming LBA correctly returns the MFSs and XSSs of the queries of size n, we consider a query Q such that $|Q| = n + 1$. LBA finds an MFS Q^* using Algorithm 1 and computes its potential XSS $pxss$. It is guaranteed that XSSs and other MFSs of Q are in the sublattices having a query Q' of $pxss$ as top since these queries are the

[4] To ensure that subqueries of an MFS are successful, it is defined that $[[\emptyset]]_D \neq \emptyset$.

largest queries that do not contain Q^*. LBA explores each such sublattice by executing Q'. If Q' is successful, it is an XSS and this sublattice cannot contain an MFS (otherwise Q' would be failing). If Q' returns an empty result, thanks to the definition of $pxss(Q, Q^*)$, $|Q'| = n$ and thus LBA computes correctly $mfs(Q')$ and $xss(Q')$ (induction hypothesis). All MFSs of Q' are also MFSs of Q and so, LBA returns exactly all MFSs of Q. Conversely, all XSSs of Q' are not necessarily XSSs of Q. We need to show that these queries are only returned by LBA if they are XSSs of Q. Let Q_p be an XSS of Q'. If Q_p is not an XSS of Q, then there is a query in $pxss$ which contains an XSS of Q larger than Q_p (by construction, $pxss$ contains the largest potential XSSs). LBA does not add Q_p to $pxss$ and thus, this query is not returned as an XSS of Q. Now, if Q_p is an XSS of Q, there are two cases. If Q_p is not included in any query of $pxss$, LBA adds it to $pxss$ and thus this query will be returned as an XSS. Else, Q_p is included in a query Q_x of $pxss$. LBA does not add Q_p to $pxss$ but LBA will find it again when searching for the XSS of Q_x (induction hypothesis). This process may be repeated several times until Q_p is not included in a query of $pxss$ anymore. This necessarily happens, at worst, when $pxss$ is empty and LBA will correctly return it as an XSS of Q.

Proposition 4. *Assume that each execution of a query costs unit time, the algorithm LBA has a worst-case running time of $\mathcal{O}(\sqrt{n} * 2^n)$, where n is the length of the failing query.*

Proof. In the worst case, there is an exponential number of MFSs. This is for example the case if all the queries of size $\frac{n}{2}$ are MFSs (assuming that n is even, without loss of generality). Indeed, in this case, the number of MFSs is $\binom{n}{\frac{n}{2}}$. The XSSs are necessarily the direct subqueries of these MFSs. Thus the number of XSSs is $\binom{n}{\frac{n}{2}-1}$. To find an MFS, LBA needs at most n queries. Conversely, the XSS are found only by executing them. Thus the time complexity of LBA in the worst case is: $\mathcal{O}(n * \binom{n}{\frac{n}{2}} + \binom{n}{\frac{n}{2}-1}))$. By using the Stirling's approximation for the factorial, one can show that $\binom{n}{\frac{n}{2}} \approx \frac{\sqrt{2}}{\sqrt{\pi n}} * 2^n$ and thus $\mathcal{O}(n * \binom{n}{\frac{n}{2}} + \binom{n}{\frac{n}{2}-1})) = \mathcal{O}(\sqrt{n} * 2^n)$.

Since the problem of enumerating the MFSs of a failing query is NP-hard, this result is not surprising. Despite this time complexity, our experiments on the LUBM benchmark show that this algorithm can still have acceptable response time in practice (see Section 5). We have also defined strategies to improve the performance of this algorithm. They are presented in the next section.

3.4 Heuristics

In the following, we propose different heuristics to improve the performance of the LBA algorithm in average case.

Heuristic 1 (Cartesian Products). Proper subqueries of the initial query can be Cartesian products (the triple patterns do not share any variable), which are usually expensive to compute. Instead of executing a Cartesian product, it can be decomposed into connected parts. If one of these connected parts is failing, then the Cartesian product fails as well. Otherwise, it succeeds.

Heuristic 2 (Query Cache). By definition, if a query succeeds, all its subqueries are successful. Thus, once the LBA has executed a succesful query, it is unnecessary to execute

one of its subqueries. As a consequence, to minimize the number of executed queries, the LBA algorithm maintains a cache of the found successful queries. Before executing a subquery, the algorithm checks first if this is a subquery of one of the queries contained in this cache.

Conversely, if a query is failing, all its superqueries are failing as well. The LBA algorithm is designed to prune from the search space the queries that include a previously found MFS. Thus, the LBA algorithm does not need to check if a query is a superquery of an MFS. Yet, the LBA maintains a cache of failing queries for the ones that we found when decomposing Cartesian products (previous heuristic).

Heuristic 3 (Obvious Failing Queries). If there is an inconsistency in a query, this query is failing. For example, the following query is failing as the domain of the *researchInterest* property is *Professor* and not *Lecturer* (see Figure 1).

```
SELECT ?p WHERE { ?p type Lecturer . ?p researchInterest "SW" }
```

Once an obvious failing query is found, the $FindAnMFS$ algorithm can be run with this query as input. Compared to an execution of this algorithm with the initial query, this will minimize the number of executed queries.

Heuristic 4 (Potential XSSs Ordering). In the LBA algorithm, the order in which the potential XSSs are processed is not specified. As the $FindAnMFS$ algorithm needs to execute fewer queries for short queries than for large queries, we process the potential XSSs from the shorter to the larger ones. Moreover, using this order, the large potential XSS will most likely be replaced with shorter potential XSSs thanks to the found MFSs.

Heuristic 5 (Triple Patterns Ordering). The $FindAnMFS$ algorithm is non-deterministic as the order in which the triple patterns are processed is not defined. Empirically, we have observed that by removing the triple patterns in the order in which they are present in the initial query, it increases the chance of using the caching made by the triplestore. This is probably due to the fact that, in this order, the executed subqueries tend to be syntactically closer, which may help the query optimizer to detect data that they share.

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 [22] has proposed a solution to avoid this problem in the context of recommender systems. This approach is based on a matrix, we 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 [22], 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.

4.1 Definition of the Relaxed Matrix of a Query

Let $Q = t_1 \wedge ... \wedge t_n$ be an RDF query, the *potential solutions* of Q are the mappings (as defined in Section 2) that satisfy one triple pattern Q or a combination of those triple patterns. This set of potential solutions of Q is denoted by $ps(Q, D)$ and is formally defined by $ps(Q, D) = \{\mu \mid \exists\{i, ..., j\} \subset \{1, ..., n\} : \mu \in [[t_i]]_D \bowtie \cdots \bowtie [[t_j]]_D\}$. Using this definition, the relaxed matrix of a query is defined as follows.

Definition 3. *The relaxed matrix M of a query $Q = t_1 \wedge ... \wedge t_n$ over an RDF database D is a two-dimensional table created with mappings $\mu \in ps(Q, D)$ as rows and triple patterns $t_i \in Q$ as columns. For a mapping $\mu \in ps(Q, D)$ and a triple pattern $t_i \in Q$, $M[\mu][t_i] = 1 \Leftrightarrow \mu(t_i) \in D, else\ M[\mu][t_i] = 0$.*

Figure 6(c) presents the relaxed matrix of the query Q given in Figure 6(b) when it is executed on the RDF dataset presented in Figure 6(a). Each row of the matrix is a mapping that satisfies at least one triple pattern. For example, the first row corresponds to the mapping $\mu : ?p \rightarrow p_1$, which satisfies the triple pattern t_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.

t		
s	**p**	**o**
p_1	type	Professor
p_1	teacherOf	c_1
p_2	type	Professor
p_2	teacherOf	c_2
c_2	name	AI
p_3	type	Lecturer
p_3	teacherOf	c_3
c_3	name	DB
c_4	name	DB

(a) RDF triples

```
SELECT ?p ?c WHERE {
?p type Professor      (t1)
?p teacherOf ?c        (t2)
?c name "DB" }         (t3)
```

(b) The query Q

?p	?c	t_1	t_2	t_3	
p_1	null	1	0	0	
p_2	null	1	0	0	
p_1	c_1	1	1	0	*
p_2	c_2	1	1	0	*
p_3	c_3	0	1	1	*
p_1	c_3	1	0	1	*
p_2	c_3	1	0	1	*
p_1	c_4	1	0	1	*
p_2	c_4	1	0	1	*
null	c_3	0	0	1	
null	c_4	0	0	1	

(c) The relaxed matrix of Q

$xss(Q) = \{t_1 \wedge t_2, t_2 \wedge t_3, t_1 \wedge t_3\}$
$mfs(Q) = \{t_1 \wedge t_2 \wedge t_3\}$

(d) The MFSs and XSSs of Q

Fig. 6. Illustration of the Matrix-Based Approach (MBA)

4.2 Computing the Relaxed Matrix of a Query

As we have seen in Section 2, the evaluation of an RDF query consists in finding the mappings that satisfy *all its triple patterns* using join operations. The relaxed matrix contains mappings

that satisfy *at least one triple pattern*. Intuitively, one can think of using the outer join operator to compute these mappings. This operator is defined as follows.

Definition 4. *Let Ω_1 and Ω_2 be sets of mappings, the difference of Ω_1 and Ω_2 is defined by:* $\Omega_1 - \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2,\ \mu_1 \text{ and } \mu_2 \text{ are not compatible}\}$. *The full outer join of Ω_1 and Ω_2 is defined by:* $\Omega_1 \mathbin{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 - \Omega_2) \cup (\Omega_2 - \Omega_1)$.

However, as observed in the context of relational database by Galindo-Legaria [26], an outer join operation eliminates from its operands the mappings that satisfy the inner join operation. As an example, Figure 7 presents the result of the outer join operations between the triple patterns of the query given in Figure 6(b). As depicted, the operation $[[t_1]]_D \mathbin{⟕} [[t_2]]_D$ does not keep the mapping $\mu_1 : ?p \to p_1$ from $[[t_1]]_D$. Indeed, this mapping is compatible with the mapping $\mu_2 : ?p \to p_1, ?c \to c_1$ and thus it is not an element of $[[t_1]]_D - [[t_2]]_D$.

Keeping the mapping μ_1 is particularly important in the context of RDF. Indeed, contrary to SQL, the predicates used for joining/outer-joining relations in SPARQL are never *null-rejecting* [23]. A predicate p is *null-rejecting* if it evaluates to false (or undefined) whenever a null value is used in p. As SPARQL is never null-rejecting, the mapping $\mu_1 : ?p \to p_1$ is compatible with the mapping $?c \to c_3$. The union of these two mappings is $?p \to p_1, ?c \to c_3$, which is an element of $ps(Q, D)$ not computed by outer join operations. As a consequence, an *extended join operation* is defined as follows.

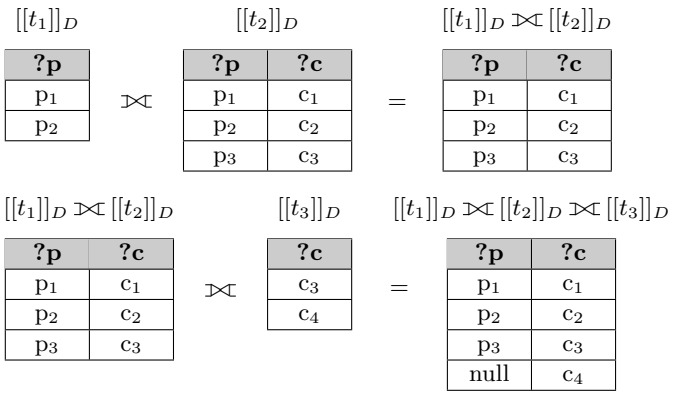


Fig. 7. Full outer join operations between sets of mappings

Definition 5. *Let Ω_1 and Ω_2 be sets of mappings, the extended join of Ω_1 and Ω_2 is defined by:* $\Omega_1 \mathbin{⟕^*} \Omega_2 = \Omega_1 \cup (\Omega_1 \bowtie \Omega_2) \cup \Omega_2$[5].

Proposition 5. *The set of potential solutions of Q can be computed with extended join operations:* $ps(Q, D) = [[t_1]]_D \mathbin{⟕^*} \cdots \mathbin{⟕^*} [[t_n]]_D$.

[5] As the semantics of SPARQL is never *null-rejecting*, contrary to the relational algebra, this expression is not equivalent to: $(\Omega_1 \cup \Omega_2) \bowtie (\Omega_1 \cup \Omega_2)$.

Proof. As the *join* operator distributes over *union* [23], it can be shown that:

$$\begin{aligned}[[t_1]]_D \bowtie^* [[t_2]]_D \bowtie^* [[t_3]]_D = &[[t_1]]_D \cup [[t_2]]_D \cup [[t_3]]_D \cup ([[t_1]]_D \bowtie [[t_2]]_D) \cup \\ &([[t_2]]_D \bowtie [[t_3]]_D) \cup ([[t_1]]_D \bowtie [[t_3]]_D) \cup \\ &([[t_1]]_D \bowtie [[t_2]]_D \bowtie [[t_3]]_D)\end{aligned}$$

Thus, the expression $[[t_1]]_D \bowtie^* [[t_2]]_D \bowtie^* [[t_3]]_D$ computes the mappings that satisfy t_1 or t_2 or t_3 or a combination of those triple patterns. This result can be generalized to n triple patterns and thus, the expression $[[t_1]]_D \bowtie^* \cdots \bowtie^* [[t_n]]_D$ computes the mappings that satisfy one triple pattern of Q or a combination of those triple patterns i.e., the set $ps(Q, D)$.

The Algorithm 3 is based on the previous definition. It computes the relaxed matrix using a nested loop algorithm. Each triple pattern t_i is evaluated over D to obtain the set $[[t_i]]_D$ (line 3). Then, each mapping μ of this set is compared with each mapping μ' in the matrix (line 5). If μ and μ' are compatible, the resulting mapping is inserted with a value set to 1 for the column t_i and values of the row corresponding to μ' for the other columns. If this mapping was already in the matrix, the corresponding row is just updated with a value set to 1 for the column t_i (line 6-12). To respect the semantics of the extended join operation, the mapping μ is inserted (if it was not already done) in the matrix with a value set to 1 for the column t_i and 0 for the other columns (line 13-15).

An example of Algorithm 3 execution for the query given in Figure 6(b) is presented in Figure 8. The algorithm executes the triple pattern t_1 of the query Q. Each result is inserted in the matrix with a value set to 1 for the column t_1 and 0 for other columns. Then, the algorithm executes the triple pattern t_2. Two results are compatible with the mappings contain in M. They are added to this matrix with a value set to 1 for the columns t_1 and t_2. The mapping $?p \rightarrow p_3, ?c \rightarrow c_3$ is not compatible with any mapping of M. It is added to the matrix with a value set to 1 only for the t_2 column. The same process is applied for t_3 to find the final relaxed matrix.

The Algorithm 3 only requires n queries where n is the number of triple patterns. Yet, our experiments conducted on the LUBM benchmark show that this algorithm can still take a notable amount of time as the size of the matrix can be large for queries involving triple patterns that are not selective. Moreover, proper subqueries of the initial query can lead to Cartesian products, which imply an expensive computation cost as well as a matrix of a large size (see Section 5 for details). As a first step to improve this approach, we have specialized it 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 [27].

4.3 Optimized Computation of the Matrix for Star-shaped Queries

In the following, we propose two approaches to compute the relaxed matrix of a star-shaped query. The first one, called NQ, is independent of the implementation of the RDF database while the second one, called $1Q$, requires an RDF database implemented as a triples table. As an illustrative example, we consider our initial query depicted in Figure 9(b).

The NQ approach. The computation of the relaxed matrix for star-shaped queries is simpler than in the general case. First, subqueries of a star-shaped query cannot be Cartesian

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

1 $M \leftarrow \emptyset$;
2 **foreach** *triple pattern* $t_i \in Q$ **do**
3 **foreach** $\mu \in [[t_i]]_D$ **do**
4 $isInserted \leftarrow false$;
5 **foreach** $\mu' \in M$ **do**
6 **if** μ *and* μ' *are compatible* **then**
7 **if** $(\mu' \cup \mu) \notin M$ **then**
8 $M \leftarrow M \cup \{\mu' \cup \mu\}$;
9 $M[\mu' \cup \mu][t_k] \leftarrow M[\mu'][t_k]$ for $k \in 1 \cdots n \wedge k \neq i$;
10 $M[\mu' \cup \mu][t_i] \leftarrow 1$;
11 **if** $(\mu \cup \mu') = \mu$ **then**
12 $isInserted \leftarrow true$;
13 **if** *not* $isInserted$ **then**
14 $M \leftarrow M \cup \{\mu\}$;
15 $M[\mu][t_k] \leftarrow 1$ if $k = i$, else 0; $(k \in 1 \cdots n)$
16 **return** M;

$M = \emptyset$

$[[t_1]]_D$ $M =$

?p	?c	t_1	t_2	t_3
p_1	null	1	0	0
p_2	null	1	0	0

$[[t_2]]_D$ $M =$

?p	?c	t_1	t_2	t_3
p_1	null	1	0	0
p_2	null	1	0	0
p_1	c_1	1	1	0
p_2	c_2	1	1	0
p_3	c_3	0	1	0

$[[t_3]]_D$ $M =$

?p	?c	t_1	t_2	t_3
p_1	null	1	0	0
p_2	null	1	0	0
p_1	c_1	1	1	0
p_2	c_2	1	1	0
p_3	c_3	0	1	1
p_1	c_3	1	0	1
p_2	c_3	1	0	1
p_1	c_4	1	0	1
p_2	c_4	1	0	1
null	c_3	0	0	1
null	c_4	0	0	1

Fig. 8. An example of Algorithm 3 execution that computes the relaxed matrix

t		
s	**p**	**o**
p_1	type	Professor
p_1	researchInterest	SW
p_1	teacherOf	DB
p_2	type	Lecturer
p_2	teacherOf	DB
p_3	teacherOf	DB
p_4	researchInterest	SW
p_5	type	Lecturer
p_5	teacherOf	AI
p_6	type	Professor
p_6	researchInterest	NLP
p_6	teacherOf	WS
p_7	type	Professor
p_7	name	John
p_7	researchInterest	NLP

(a) RDF triples

```
SELECT ?p ?a WHERE {
?p age ?a                    (t1)
?p type Lecturer             (t2)
?p researchInterest "SW"     (t3)
?p teacherOf "DB" }          (t4)
```

(b) The query Q

?p	t_1	t_2	t_3	t_4	
p_1	0	0	1	1	*
p_2	0	1	0	1	*
p_3	0	0	0	1	
p_4	0	0	1	0	
p_5	0	1	0	0	

(c) The relaxed matrix of Q

$xss(Q) = \{t_2 \wedge t_4, t_3 \wedge t_4\}$
$mfs(Q) = \{t_1, t_2 \wedge t_3\}$

(d) The MFSs and XSSs of Q

Fig. 9. Illustration of the MBA approach for star-shaped queries

products. Second, a single variable, denoted by x, is used to join all the triple patterns. Thanks to this latter property, we only need to record the values of the join variable (i.e., the restriction of the function μ to $\{x\}$ denoted by $\mu_{|\{x\}}$) in the relaxed matrix. The values of the join variable that satisfy a triple pattern t are evaluated as follows: $[[t]]_D = \{\mu_{|\{x\}} \mid dom(\mu) = var(t) \wedge \mu(t) \in D\}$ and the relaxed matrix is computed using the following proposition.

Proposition 6. *The set of potential solutions of a star-shaped query Q can be computed with full outer join operations: $ps(Q, D) = [[t_1]]_D \mathbin{⟗} \cdots \mathbin{⟗} [[t_n]]_D$.*

Proof. Let t_1 and t_2 be two triple patterns of a star-shaped query. If $\mu_1 \in [[t_1]]_D$ and $\mu_2 \in [[t_2]]_D$, μ_1 and μ_2 are compatible if they have the same value for the join variable, i.e., $[[t_1]]_D \bowtie [[t_2]]_D = [[t_1]]_D \cap [[t_2]]_D$. Using this property and the laws of set algebra, one can deduce that:

$$\begin{aligned}
[[t_1]]_D \mathbin{⟗} [[t_2]]_D &= ([[t_1]]_D \cap [[t_2]]_D) \cup ([[t_1]]_D - [[t_2]]_D) \cup ([[t_2]]_D - [[t_1]]_D) \\
&= ([[t_1]]_D \cap [[t_2]]_D) \cup [[t_1]]_D \cup [[t_2]]_D \\
&= [[t_1]]_D \mathbin{⟗}^* [[t_2]]_D
\end{aligned}$$

Thus, in the case of star-shaped queries, full outer joins are equivalent to extended join operations and we have shown in Proposition 5 that this latter operation computes the relaxed matrix of a query.

The Algorithm 4, called NQ, uses the previous proposition to compute the relaxed matrix of a star-shaped query. This algorithm executes one query for each triple pattern t_i (line 2).

For each result μ, the value of the join variable is added to the matrix, if it is not already present in it, and the values of this row are set to 1 for the column corresponding to t_i and 0 for the other columns (line 4-6). If this mapping was already in the matrix, the corresponding row is just updated with a value set to 1 for the column t_i (line 7).

An example of Algorithm 4 execution for our illustrative query (see Figure 9(b)) is given in Figure 10. The execution of the triple pattern t_1 returns an empty result and thus the matrix remains empty. Then the algorithm executes the triple pattern t_2 and adds the two resulting mappings restricted to the join variable $?p$ in the matrix with a value set to 1 for the column t_2 and 0 for other columns. The same process is applied for the triple pattern t_3. Finally, the triple pattern t_4 is executed. It returns the mappings $?p \rightarrow p_1$, $?p \rightarrow p_2$ and $?p \rightarrow p_3$. As the two first ones were already in the matrix, the corresponding rows are just updated with a value set to 1 for the column t_4. Conversely, the mapping $?p \rightarrow p_3$ is inserted in the matrix.

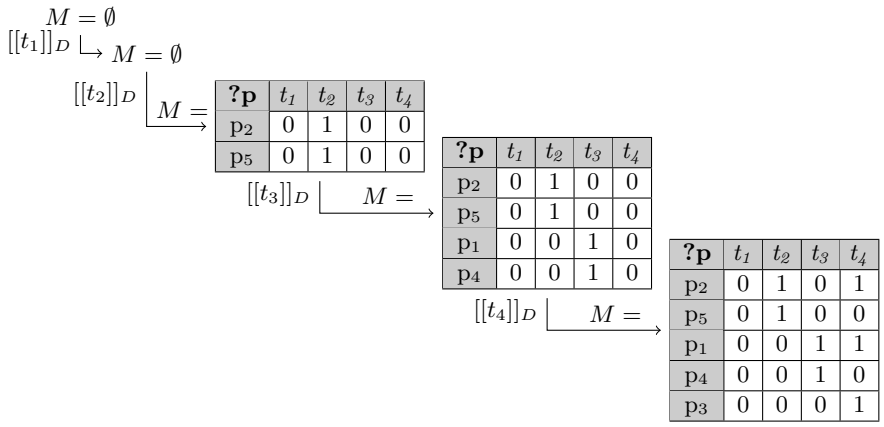
Algorithm 4: Computation of the matrix for star-shaped queries (NQ)

```
   ComputeMatrixStarQueryNQ(Q, D)
       inputs : A failing star-shaped query Q = t1 ∧ ... ∧ tn with x as join variable;
                An RDF database D
       output: The relaxed matrix M
1      M ← ∅;
2      foreach triple pattern ti ∈ Q do
3          foreach μ ∈ [[ti]]D do
4              if μ|{x} ∉ M then
5                  M ← M ∪ {μ|{x}};
6                  M[μ|{x}][tk] ← 0 for k ∈ 1 ··· n ∧ k ≠ i;
7              M[μ|{x}][ti] ← 1;
8      return M;
```

The 1Q approach. The algorithm 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 [28], we use SQL outer join operations to compute this expression and the *coalesce* function[6] to manage unbound values. In addition, we use the *case* operator to check if a triple pattern is matched and thus to get the matrix values (1 if it is matched, else 0).

We illustrate this approach with the query given in Figure 9(b). We denote by `ti` the query (or view) that computes the values of the join variable for the triple pattern t_i. For example, `t1` is the following query: `select distinct s from t where p='age'`. Using, this

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



?p	t_1	t_2	t_3	t_4
p_2	0	1	0	0
p_5	0	1	0	0

?p	t_1	t_2	t_3	t_4
p_2	0	1	0	0
p_5	0	1	0	0
p_1	0	0	1	0
p_4	0	0	1	0

?p	t_1	t_2	t_3	t_4
p_2	0	1	0	1
p_5	0	1	0	0
p_1	0	0	1	1
p_4	0	0	1	0
p_3	0	0	0	1

Fig. 10. An example of the algorithm NQ

notation, the SQL query used to compute the relaxed matrix of our example query is:

```
select coalesce(t1.s , t2.s, t3.s, t4.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,
       case when t3.s is null then 0 else 1 end as t3,
       case when t4.s is null then 0 else 1 end as t4
from t1 full outer join t2 on t1.s = t2.s
        full outer join t3 on coalesce(t1.s , t2.s) = t3.s
        full outer join t4 on coalesce(t1.s , t2.s, t3.s) = t4.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.

4.4 Computing the XSSs from the Relaxed Matrix

Abusing notation, we denote by $xss(\mu)$ the maximal proper subquery of Q that retrieves a mapping $\mu \in ps(Q, D)$. It is directly obtained from the relaxed matrix: $xss(\mu) = \{t_i \wedge ... \wedge t_j \mid \forall t_k \in \{t_i, ..., t_j\} : \ M[\mu][t_k] = 1\}$. Finding the XSSs of a query Q is done in two steps:

1. Computing the skyline SKY of the relaxed matrix: $SKY(M) = \{\mu \in ps(Q, D) \mid \nexists \mu' \in ps(Q, D)$ *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 [29] for a survey). In Figure 6(c) and Figure 9(c), all the rows composing the skyline of the relaxed matrices are marked with $*$.
2. Retrieving the distinct maximal proper subqueries of Q that return 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 examples are given in Figure 6(d) and Figure 9(d). They are framed in the relaxed matrices.

4.5 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 matrix computed by the MBA approach can be used as an index to improve the performance of the LBA approach. This latter 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 once the relaxed matrix has been computed.

5 Experimental Evaluation

In this section, we investigate the scalability of our proposed algorithms and compare them with one baseline method and a related work algorithm.

5.1 Experimental Setup

We have implemented the LBA and MBA algorithms in JAVA 1.8 64 bits. These algorithms take as input a failing SPARQL query and return the set of MFSs and XSSs of this query. In our current implementation, these algorithms can be run on top of Jena TDB (version 1.0.1) and Sesame Native Store (version 2.8.4). Integrating these algorithms with another triplestore is easy as soon as this latter supports the SPARQL language. This integration is simply done by implementing an interface that defines some basic operations for a triplestore (e.g., establishing a connection and executing a SPARQL query). 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. In this section we report the results of our experiments run on top of Jena TDB as the ones obtained with Sesame were similar. All times presented are the average of five consecutive runs of the algorithms. Before the actual measured run starts we run the algorithms once. 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 Java Virtual Machine.

5.2 Dataset and Queries

As in previous work on RDF query relaxation [13, 14], we used datasets generated with the LUBM benchmark [10]. It is based on a university ontology composed of 43 classes and 32 properties. The used datasets range from LUBM100 to LUBM1K. Table 1 gives the approximate number of triples and instances of these datasets (in millions).

As the LUBM workload mainly contains successful queries, we have generated a set of random failing queries. To define the characteristics of these queries, we have considered the study proposed by Arias Gallego and al. [27] on real-world SPARQL queries executed on the DBPedia and SWDF datasets. They have shown that these queries have three main query patterns *star*, *chain* and *composite* and range from 1 to 15 triple patterns. As a consequence,

	LUBM100	LUBM250	LUBM500	LUBM750	LUBM1K
Number of triples	13M	30M	65M	90M	130M
Number of instances	2M	5M	11M	16M	22M

Table 1. Datasets characteristics

we have generated 225 queries ranging from 1 to 15 triple patterns (5 queries for a given number of triple patterns) for these three main RDF query patterns (75 queries for a given query pattern). These queries are characterized as follows.

- Star queries are characterized by *subject-subject* joins between triple patterns as the join variable is on subject position.
- Chain queries are composed of *object-subject* joins, i.e., the join variable is on object position in one triple pattern and on subject position in the other.
- Composite queries are randomly made of *subject-subject*, *object-object* and *subject-object* (or *object-subject*) joins. We have not generated queries that include triple patterns with variables on predicate positions as they are infrequent in practice [27].

Table 2 presents an example of these three types of generated queries[7].

star	`SELECT * WHERE { ?X rdf:type ub:FullProfessor .`
	`?X ub:age ?Y1`
	`?X ub:memberOf ?Y2 .`
	`?X ub:doctoralDegreeFrom <University911> }`

chain	`SELECT * WHERE { ?Y1 ub:softwareDocumentation ?Y2 .`
	`?Y2 ub:publicationAuthor ?Y3`
	`?Y3 ub:headOf ?Y4 .`
	`?Y4 ub:affiliateOf <FullProfessor4> }`

composite	`SELECT * WHERE { <Dept8> ub:subOrganizationOf ?Y1 .`
	`?Y1 ub:subOrganizationOf <University17>`
	`?Y2 ub:headOf ?Y1 .`
	`?Y3 ub:affiliateOf ?Y2 }`

Table 2. Example of generated queries with four triple patterns

5.3 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 0.4.10 for this purpose [30]. As Table 2 shows, this data structure ensures that the matrix size remains small even if the number of matrix rows is large (around 1MB for 1.7M rows). Table 3 only includes results for star-shaped queries as other queries required too many resources due to Cartesian products.

[7] For readability, we shorten the URIs

	3 TP	5 TP	8 TP	10 TP	13 TP	15 TP
Computation time with NQ (in sec)	2.36	8.9	17.3	25.5	29.3	36.2
Computation time with 1Q (in sec)	2.36	9	15.5	19.6	23.6	29
Size (in KB)	80	320	480	620	620	1290
Number of rows (in K)	249	911	1326	1532	1532	1738

Table 3. Relaxed Matrix Properties on LUBM100 (TP stands for Triple Pattern)

For the computation of the MBA relaxed matrix, we have compared the two algorithms *1Q* and *NQ* described in Section 4. As the *1Q* approach requires an RDF database implemented on top of an RDBMS, we have used the Oracle 12c RDBMS to implement the triples table. As Table 3 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 30 seconds for a query with 15 TP. Despite this important computation time, the MBA approach can still be interesting as the matrix can be precomputed for usual failing queries identified thanks to query logs.

5.4 XSS and MFS Computation Time

In this section, we present experiments on the scalability properties of the following algorithms when both the size of the query and the size of the database increase.

- LBA: the algorithm described in Section 3 without any heuristics.
- LBA-OPT: the LBA algorithm with the heuristics described in Section 3.4.
- 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 for traversing the subquery lattice. For each failing (resp., succeeding) subquery reached during the search, we check if this is an MFS (resp., an XSS). This algorithm executes each subquery once (in total $2^n - 2$ queries, where n is the number of TP).
- ISHMAEL: the algorithm proposed in [21] that we have tailored to return both the XSSs and MFSs of an RDF query.

5.5 Experiments with Star-shaped Queries

In Figure 11, we illustrate the time to compute the MFSs and XSSs as a function of the query size and in Figure 12 as a function of the dataset size. These graphs are displayed in logarithmic scale for readability. Table 4 gives the number of executed queries for each algorithm. An algorithm that evaluates all the subqueries such as DFS can be used for queries with only a few triple patterns. 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 below 1 second for queries that do not have more than 11 triple

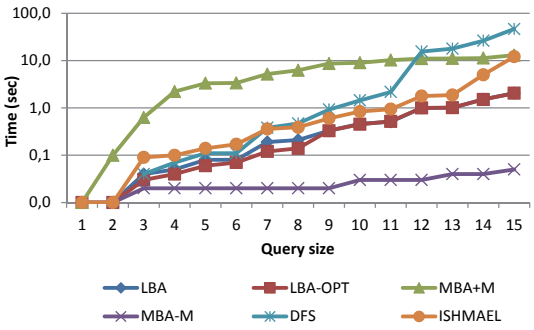


Fig. 11. Star-shaped query time (log scale) vs query size, for LUBM100

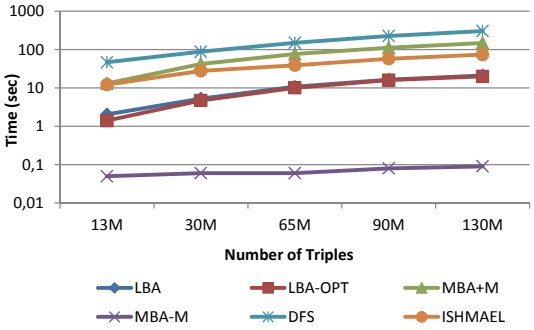


Fig. 12. Star-shaped query time (log scale) vs dataset size, for 15 TP

patterns on LUBM100. For larger queries, LBA outperforms ISHMAEL (recall that the results are presented in logarithmic scale). Indeed, for queries that have between 12 and 15 triple patterns, the response times of LBA range from 1 to 2 seconds while they exceed 10 seconds for ISHMAEL. As Figure 12 shows, these response times scale linearly with the size of the dataset. We have identified that the performance difference is due to the simplified computation of the potential XSSs that leads to a smaller number of executed queries (see Table 4).

MBA-M provides response times of some milliseconds even for queries with 15 triple patterns on LUBM1K. 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 (MBA+M), this computation time is important and thus MBA+M can hardly be used in practice on large datasets. Indeed, this computation time will only be acceptable if the matrix size is small, i.e., when the query only involve selective triple patterns (they can be identified using database statistics).

Finally, we observe that LBA and LBA-OPT have similar performance. This happens because subqueries of a star-shaped queries cannot lead to Cartesian products. Considering the query cache, we have observed that approximatively 10% less queries are executed in LBA-OPT compared to LBA. But, as these queries are included in successful queries, they are queries of small size and thus have an execution time insignificant compared to large size queries, which are not in the query cache. Despite these results, our proposed heuristics are still relevant for other kinds of RDF queries as we show in the next section.

5.6 Experiments with Chain-shaped Queries

Figure 13 and 14 as well as Table 5 present our results with chain-shaped queries. In the case of chain-shaped queries, all non contiguous subqueries of the initial query lead to Cartesian products. As a consequence, results of MBA are not provided as the size of the matrix exceeded the size of the main memory for most queries. The algorithm DFS also quickly becomes inapplicable in practice as it takes more than 3 minutes for queries with 6 TP on LUBM100.

		3 TP	5 TP	8 TP	10 TP	13 TP	15 TP
LBA	#Executed Queries	5	14	37	67	117	175
LBA-OPT	#Executed Queries	4	11	30	57	103	153
	#Cached Queries	1	4	7	10	14	22
DFS	#Executed Queries	6	30	254	1022	8190	32766
ISHMAEL	#Executed Queries	9	27	76	138	260	393

Table 4. Number of Executed Queries on LUBM 100 for Star Queries

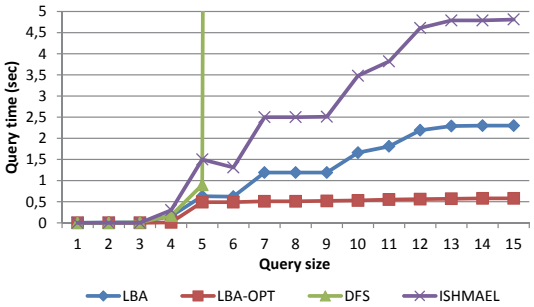


Fig. 13. Chain-shaped query time vs query size, for LUBM100

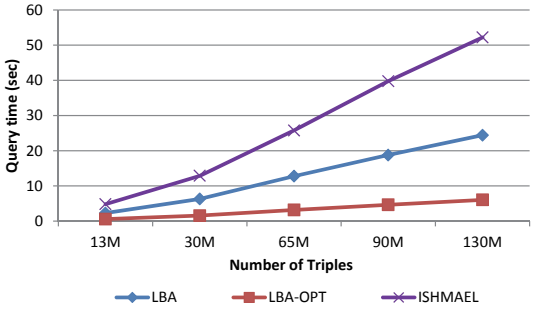


Fig. 14. Chain-shaped query time vs dataset size, for 15 TP

		3 TP	5 TP	8 TP	10 TP	13 TP	15 TP
LBA	#Executed Queries	5	14	33	50	75	100
LBA-OPT	#Executed Queries	4	10	24	36	58	75
	#Cached Queries	1	7	22	37	57	88
DFS	#Executed Queries	6	30	254	1022	8190	32766
ISHMAEL	#Executed Queries	11	31	74	113	169	225

Table 5. Number of Executed Queries on LUBM 100 for Chain Queries

As it executes an average of 75% fewer queries than DFS and 55% fewer queries that ISHMAEL, the LBA algorithm (without heuristics) has better performance. But this algorithm still may encounter expensive Cartesian products (i.e., Cartesian product whose connected components have a large number of rows). As a consequence, the safe way to compute MFSs and XSSs of a chain-shaped query is to use LBA-OPT that decomposes Cartesian product in its connected components. In our experiments, LBA-OPT is in average 67% faster than LBA. This behaviour is explained by the fact that this algorithm uses extensively the query cache (see Table 5) and does not execute Cartesian products. Indeed, as the decomposition of Cartesian product leads to an extended use of the query cache, we found that approximatively 30% less queries were executed in LBA-OPT compared to an execution without any query caching. The response time of LBA-OPT is below 1 second for all queries on LUBM100. As this algorithm scales linearly with the size of the dataset (see Figure 14), this response time is around 5 seconds for the largest tested queries on LUBM1K.

		3 TP	5 TP	8 TP	10 TP	13 TP	15 TP
LBA	#Executed Queries	6	15	35	51	90	118
LBA-OPT	#Executed Queries	5	11	26	38	67	86
	#Cached Queries	1	5	17	28	49	82
DFS	#Executed Queries	6	30	254	1022	8190	32766
ISHMAEL	#Executed Queries	12	29	76	116	212	285

Table 6. Number of Executed Queries on LUBM 100 for Composite Queries

5.7 Experiments with Composite-shaped Queries

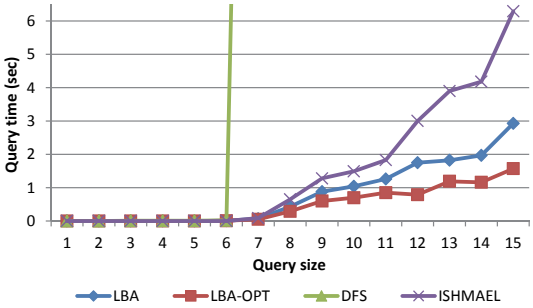


Fig. 15. Composite-shaped query time vs query size, for LUBM100

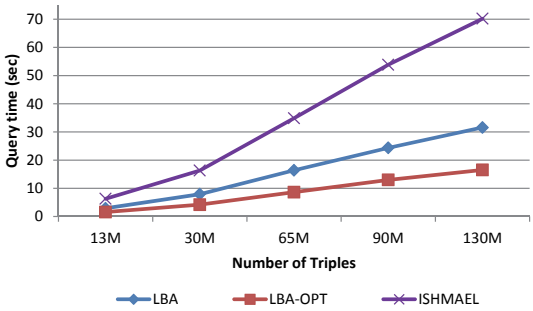


Fig. 16. Composite-shaped query time vs dataset size, for 15 TP

Figure 15 and 16 as well as Table 6 present our results with composite-shaped queries. With this type of queries, the probability of executing an expensive Cartesian product is smaller than in the case of chain-shaped queries. Yet, our experiments shows that the size of the matrix used in the MBA algorithm exceeded the size of the main memory for most queries. The DFS algorithm also quickly becomes extremely slow with queries that have more than 6 TP. As for chain-shaped query, the safe way to find MFSs and XSSs of a query is to use the LBA-OPT algorithm. In our experiments, it is in average 37% faster than LBA and 60% faster than ISHMAEL. Compared to its response time with chain-shaped queries, the response time of LBA-OPT was slower with composite-shaped queries (around 15 seconds for queries with 15 TP). This happens because composite queries involve different join query patterns and not only *object-subject* join pattern like with chain-shaped queries. In particular, we have observed that executing *object-object* join query patterns was particularly slow on Jena TDB.

6 Related Work

We review here the closest works related to our proposal done both in the context of RDF and relational databases.

– *RDF databases context.* Hurtado et al. [11] have addressed the relaxation of RDF queries through RDFS entailment. This work has been extended in [16, 17] to combine query approximation and relaxation and to consider conjunctive regular path queries. These relaxation and approximation techniques can be called in RDF queries using two operators: $RELAX$ and $APPROX$. We note that the types of relaxation encompassed by these techniques do not include a fine-grained relaxation of filters (a value in a triple can only be replaced by a variable and not by an approximate value). Moreover, few parameters are available in the proposed operators to control precisely the relaxation or approximation process. We have addressed these limitations in our previous work [15], where 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.

  Huang *et al.* have also addressed the problem of relaxing RDF queries in [13] and [14]. These approaches rely on the relaxation model of [11] (with the same limitation to relax filters). Two particular contributions are made in these papers: (i) ranking the relaxed queries using a distance-based method [13] or a measure based on information content [14] and (ii) optimizing the relaxation process to retrieve the top-k answers.

  Let us also mention the work done by Dolog *et al.* [20] who proposed a method for automatically relaxing over-constrained RDF queries based on background knowledge about the domain model and user preferences. The type of relaxation proposed consists in rewriting the original query by replacing some of its parts by applying specific rules. For instance, replacing a highly preferred value by a less preferred one. This requires defining *a priori* domain preferences and a posteriori user preferences. As an alternative to query relaxation, there have been works on *query auto-completion* [31–35], which check the data during query formulation to avoid empty answers. It is worth noting that none of the above works has considered the issue related to the causes of RDF query failure and then the issue of the MFSs computation.
– *Relational databases context.* Many works have been proposed for query relaxation in the setting of relational databases (see Bosc et al. [36] for an overview). In particular, Godfrey [21] has defined the algorithmic complexity of the problem of identifying the MFSs of failing relational queries 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. [36] and Pivert et al. [37] extended Godfrey's approach to the fuzzy query context. In [36], the notion of MFSs is studied in order to quickly find a relaxed fuzzy query with non-empty answers. While in [37], an approach based on a fuzzy-cardinality-based summary of the relevant part of the database, is proposed to identify a set of gradual MFSs[8]. Recently in [38], this approach is extended for computing gradual XSSs.

  In the recommendation systems setting, McSherry [39] and Jannach [22] have studied the concept of MFS and XSS. McSherry [39] proposed an incremental relaxation of the failing query based on the MFSs. These MFSs are computed with an algorithm that explores the subquery lattice. In contrast, the work of Jannach [22] leverages a matrix aiming at computing alternative answers to the failing query with the query conditions they satisfy.

[8] MFSs which are only poorly satisfied, i.e., that do not return any answer with at least a satisfaction degree equals to α (a user-defined threshold).

Our MBA approach is inspired by [22]. But, contrary to the Jannach's approach, the computation of the matrix rows is not straightforward in the context of RDF queries. Moreover, in [22], the matrix is only used to retrieve the XSSs of the query while, in our work, we show that the matrix can be used and stored as a bitmap index to improve the performance of LBA and thus retrieving both the MFSs and XSSs of the query.

7 Conclusion and Discussion

The paper addresses the problem of failing RDF Queries. More specifically, we have investigated two issues: on the one hand, the diagnosis and the identification of the causes of the query failure (by computing the MFSs) and, on the other hand, the computation of the relaxations of the query (by retrieving the XSSs).

Two approaches to efficiently compute the MFSs and XSSs of a failing RDF query are discussed. The first approach, called LBA, is a smart exploration of the subquery lattice of the failing query that leverages the properties of MFSs and XSSs. We have also proposed several heuristics to improve its performance. 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 several datasets generated with the LUBM benchmark. While a straightforward algorithm does not scale for queries with more than 5 triple patterns, the optimized LBA approach scales up well for queries up to 15 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, the computation of XSSs and MFSs is not time consuming at all even for queries with many triple patterns. If the matrix is computed at runtime, this approach will only be relevant for large queries when the cost of computing the matrix becomes acceptable in comparison with the optimization of LBA it permits.

The work presented in this paper opens many perspectives. A further optimization of the LBA and MBA approaches is subject to our future work such as using multi-query optimization techniques. As our algorithms may compute many XSSs, returning the results of all these XSSs to the end-user may not be helpful. Thus, we plan to define a ranking model for these XSSs to suggest or execute them in the ranking order established. For instance, this ranking model could be based on the result size of each XSS, then one has to study the problem of query containment over XSSs. As another perspective, we will investigate the concept of MFSs and XSSs over uncertain/trust-weighted RDF data. In this context, a response associated with a small level of certainty could be also considered as an unsatisfactory answer (since it does not serve the user needs due to its high uncertainty). To explain the failure query and repair the query, one can use particular variants of MFSs and XSSs that imply non-trivial extensions of our algorithms. Finally, we also plan to study whether our approach could be combined with other cooperative techniques that aim at handling the unsatisfactory-answer problem (e.g., *why-not queries* [40]).

Acknowledgement. The authors would like to thanks anonymous reviewers as well as Patrice Naudin and Pascal Richard for their very useful comments and suggestions.

References

1. Hoffart, J., Suchanek, F.M., Berberich, K., Weikum, G.: YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. Artificial Intelligence **194** (2013) 28–61
2. Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia - A crystallization point for the Web of Data. Journal of Web Semantics **7**(3) (2009) 154–165
3. 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: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'14). (2014) 601–610
4. Deshpande, O., Lamba, D.S., Tourn, M., Das, S., Subramaniam, S., Rajaraman, A., Harinarayan, V., Doan, A.: Building, Maintaining, and Using Knowledge Bases: A Report from the Trenches. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13). (2013) 1209–1220
5. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation 25 February 2014 (2014) http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.
6. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation 15 January 2008 (2008) http://www.w3.org/TR/rdf-sparql-query/.
7. Brickley, D., Guha, R.: RDF Schema 1.1. W3C Recommendation 25 February 2014 (2014) http://www.w3.org/TR/rdf-schema/.
8. Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: OWL Web Ontology Language Reference. W3C Recommendation 10 February 2004 (2004) http://www.w3.org/TR/owl-ref.
9. Bollacker, K.D., Evans, C., Paritosh, P., Sturge, T., Taylor, J.: Freebase: a Collaboratively Created Graph Database For Structuring Human Knowledge. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08). (2008) 1247–1250
10. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Web Semantics **3**(2-3) (2005) 158–182
11. Hurtado, C.A., Poulovassilis, A., Wood, P.T.: Query Relaxation in RDF. Journal on Data Semantics X **10** (2008) 31–61
12. Hurtado, C.A., Poulovassilis, A., Wood, P.T.: Ranking Approximate Answers to Semantic Web Queries. In: Proceeding of the 6th Extended Semantic Web Conference (ESWC'09). (2009) 263–277
13. Huang, H., Liu, C., Zhou, X.: Computing Relaxed Answers on RDF Databases. In: Proceedings of the 9th International Conference on Web Information Systems Engineering (WISE'08). (2008) 163–175
14. Huang, H., Liu, C., Zhou, X.: Approximating query answering on RDF databases. Journal of the World Wide Web: Internet and Web Information Systems (WWW) **15**(1) (2012) 89–114
15. Fokou, G., Jean, S., Hadjali, A.: Endowing Semantic Query Languages with Advanced Relaxation Capabilities. In: Proceeding of the 21st International Symposium on Methodologies for Intelligent Systems (ISMIS 2014), Roskilde, Denmark (2014) 512–517
16. Poulovassilis, A., Wood, P.T.: Combining Approximation and Relaxation in Semantic Web Path Queries. In: Proceedings of the 9th International Semantic Web Conference (ISWC'10). (2010) 631–646

17. Calí, A., Frosini, R., Poulovassilis, A., Wood, P.: Flexible Querying for SPARQL. In: Proceedings of the 13th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE'14). (2014) 473–490
18. Hogan, A., Mellotte, M., Powell, G., Stampouli, D.: Towards Fuzzy Query-relaxation for RDF. In: Proceeding of the 9th Extended Semantic Web Conference (ESWC'12). (2012) 687–702
19. Elbassuoni, S., Ramanath, M., Weikum, G.: Query Relaxation for Entity-Relationship Search. In: Proceeding of the 8th Extended Semantic Web Conference (ESWC'11). (2011) 62–76
20. Dolog, P., Stuckenschmidt, H., Wache, H., Diederich, J.: Relaxing RDF queries based on user and domain preferences. Journal of Intelligent Information Systems **33**(3) (2009) 239–260
21. Godfrey, P.: Minimization in Cooperative Response to Failing Database Queries. International Journal of Cooperative Information Systems **6**(2) (1997) 95–149
22. Jannach, D.: Fast Computation of Query Relaxations for Knowledge-based Recommenders. AI Communications **22**(4) (2009) 235–248
23. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems (TODS) **34**(3) (2009) 16:1–16:45
24. Fokou, G., Jean, S., Hadjali, A., Baron, M.: Cooperative Techniques for SPARQL Query Relaxation in RDF Databases. In: Proceeding of the 12th Extended Semantic Web Conference (ESWC 2015). (2015) 237–252
25. Sakr, S., Al-Naymat, G.: Relational Processing of RDF Queries: A Survey. SIGMOD Record **38**(4) (2009) 23–28
26. Galindo-Legaria, C.A.: Algebraic Optimization of Outerjoin Queries. PhD thesis, Harvard University (1992)
27. Gallego, M.A., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P.: An Empirical Study of Real-World SPARQL Queries. In: Proceedings of the USEWOD workshop co-located with WWW'11. (2011)
28. Cyganiak, R.: A relational algebra for SPARQL. HP-Labs Technical Report, HPL-2005-170, http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html (2005)
29. Hose, K., Vlachou, A.: A Survey of Skyline Processing in Highly Distributed Environments. VLDB Journal **21**(3) (2012) 359–384
30. Chambi, S., Lemire, D., Kaser, O., Godin, R.: Better bitmap performance with Roaring bitmaps. CoRR **abs/1402.6407** (2014)
31. Gombos, G., Kiss, A.: SPARQL Query Writing with Recommendations Based on Datasets. In: Human Interface and the Management of Information. Information and Knowledge Design and Evaluation. (2014) 310–319
32. Lehmann, J., Bühmann, L.: AutoSPARQL: Let Users Query Your Knowledge Base. In: Proceeding of the 8th Extended Semantic Web Conference (ESWC'11). (2011) 63–79
33. Campinas, S.: Live SPARQL Auto-Completion. In: Proceedings of the 13th International Semantic Web Conference (ISWC'14 Posters & Demos). (2014) 477–480
34. Möller, K., Ambrus, O., Josan, L., Handschuh, S.: A Visual Interface for Building SPARQL Queries in Konduit. In: Proceedings of the 7th International Semantic Web Conference (ISWC'08 Posters & Demos). (2008)
35. Clark, L.: SPARQL Views: A Visual SPARQL Query Builder for Drupal. In: Proceedings of the 9th International Semantic Web Conference (ISWC'10 Posters & Demos). (2010)
36. Bosc, P., Hadjali, A., Pivert, O.: Incremental Controlled Relaxation of Failing Flexible Queries. Journal of Intelligent Information Systems (JIIS) (2009) 261–283
37. Pivert, O., Smits, G., Hadjali, A., Jaudoin, H.: Efficient Detection of Minimal Failing Subqueries in a Fuzzy Querying Context. In: Proceedings of the 15th East-European Conference on Advances in Databases and Information Systems (ADBIS'11). (2011) 243–256
38. Pivert, O., Smits, G.: How to Efficiently Diagnose and Repair Fuzzy Database Queries that Fail. In: Fifty Years of Fuzzy Logic and its Applications, Studies in Fuzziness and Soft Computing. (2015) 499–517

39. McSherry, D.: Incremental Relaxation of Unsuccessful Queries. In: Advances in Case-Based Reasoning. Volume 3155. (2004) 131–148

40. Bidoit, N., Herschel, M., Tzompanaki, K.: Query-Based Why-Not Provenance with NedExplain. In: Proceedings of the 17th International Conference on Extending Database Technology (EDBT 2014). (2014) 145–156

Author Biographies

Géraud Fokou is a PhD student and a member of the Data and Model Engineering team of the Laboratory of Computer Science and Automatic Control for Systems (LIAS) at the National Engineering School for Mechanics and Aerotechnics (ISAE-ENSMA). He got a Master degree in computer science at Poitiers University and an Engineer degree in software engineering at African Institute of Computer Science of Libreville in Gabon. He spent his engineer internship at LIAS/ISAE-ENSMA working on RDF database benchmarking. His research interests include ontology engineering, semantic web, datatabases, benchmarking, knowledge extraction and query relaxation. Web page: http://www.lias-lab.fr/members/geraudfokou

Stéphane Jean is currently Assistant Professor at the University of Poitiers. He is a member of the Data and Model Engineering team of the Laboratory of Computer Science and Automatic Control for Systems (LIAS) at the National Engineering School for Mechanics and Aerotechnics (ISAE-ENSMA). With main research interests in ontologies, databases, semantic web, query optimization, model-driven engineering and cooperative answering, he has authored over 50 technical papers in well-known journals and conferences (e.g., Computers in Industry, ESWC, DEXA, etc.). During his PhD, he has designed the OntoQL language which is an extension of SQL to manage both ontologies and data stored in a database. This language has been successfully used in several projects with large companies in various domains such as the automotive and petroleum industries. Web page: http://www.lias-lab.fr/members/stephanejean

Allel Hadjali is Full Professor in Computer Science at the National Engineering School for Mechanics and Aerotechnics (ISAE-ENSMA), Poitiers, France. He has been an Associate Professor in Computer Science both in University of Rennes 1 (France) and University of Tizi-ouzou (Algeria). He is a member of the Data & Model Engineering research team of the Laboratory of Computer Science and Automatic Control for Systems (LIAS/ISEA-ENSMA). His research interests include Flexible Database Querying, Handling Preferences, Database Uncertainty, Recommendation Systems, Web Services, Qualitative and Uncertain/Approximate Reasoning with applications to Artificial Intelligence and Information Systems. His recent works were published in well-known journals (e.g., Fuzzy Sets and Systems or Annals of Mathematics and Artificial Intelligence). He also published several papers in International Conferences (e.g., ESWC, FQAS, SUM, Fuzz-IEEE, CoopIS, etc.). Web page: http://www.lias-lab.fr/members/allelhadjali.

Mickaël BARON is a Research Engineer and a member of the Data and Model Engineering team of the Laboratory of Computer Science and Automatic Control for Systems (LIAS) at the National Engineering School for Mechanics and Aerotechnics (ISAE-ENSMA). He got his PhD degree in computer science at Poitiers University. He spent one year at INRIA Rocquencourt into the MERLIN team for designing human computer interaction methods and tools. He spent also two years in high-tech companies on designing and validating software. He is the responsible of all the software research development conducted in the LIAS laboratory. His research interests include ontology engineering, data persistence, query relaxation, software validation, object oriented programming and software quality. Web page: http://www.lias-lab.fr/members/mickaelbaron