Query Answering over Uncertain RDF Knowledge Bases: Explain and Obviate Unsuccessful Query Results

Ibrahim Dellal, Stéphane Jean, Allel Hadjali, Brice Chardin, Mickaël Baron

LIAS/ISAE-ENSMA - University of Poitiers
1, Avenue Clement Ader, 86960 Futuroscope Cedex, France
{firstname.lastname}@ensma.fr

Received: 30 May 2018
Revised: 21 Dec 2018
Accepted: 30 Dec 2018

Abstract. Several large *uncertain Knowledge Bases* (KBs) are available on the Web where facts are associated with a certainty degree. When querying these uncertain KBs, users seek high quality results i.e., results that have a certainty degree greater than a given threshold α. However, as they usually have only a partial knowledge of the KB contents, their queries may be *failing* i.e., they return no result for the desired certainty level. To prevent this frustrating situation, instead of returning an empty set of answers, our approach explains the reasons of the failure with a set of α*Minimal Failing Subqueries* (αMFSs), and computes alternative relaxed queries, called α*MaXimal Succeeding Subqueries* (αXSSs), that are as close as possible to the initial failing query. Moreover, as the user may not always be able to provide an appropriate threshold α, we propose three algorithms to compute the αMFSs and αXSSs for other thresholds, which also constitutes a relevant feedback for the user. Multiple experiments with the WatDiv benchmark show the relevance of our algorithms compared to a baseline method.

Keywords: Uncertain Knowledge Bases, RDF Quad, SPARQL queries, Empty answers, Named graph, Reification, Quad-store.

1 Introduction

A *Knowledge Base* (KB) is a collection of entities and facts about them. Recent advances in information extraction techniques have led to the construction of large KBs from Web sources. Well-known examples of academic KBs include SigmaKB [1], YAGO [2], and NELL [3]. Commercial KBs have also been built, such as Google's Knowledge Vault [4] and Microsoft's Probase [5]. These KBs contain billions of facts captured as RDF triples *(subject, predicate, object)* and are queried with the SPARQL [6] language. As these KBs have been constructed by mining the Web for information, their facts may be inconsistent, ambiguous

or uncertain. Therefore, efforts have been made to define extensions of RDF and SPARQL that support trust weighted data [7, 8]. In this context, an explicit degree of certainty is assigned to KB facts and query results. KBs in which each fact is associated with a degree of certainty are called *uncertain KBs*. The academic and commercial KBs previously mentioned are examples of real uncertain KBs.

When querying uncertain KBs, users expect to obtain high quality results i.e., results that have a certainty degree greater than a given threshold α. However, as they rarely know the underlying structure and content of a KB, they may be faced with an empty answer problem i.e., they obtain no result or results with a degree of certainty lower than α. This is not an uncommon problem when querying Web-accessible KBs. Indeed, the study conducted by Saleem et al. [9] on SPARQL endpoints shows that ten percent of queries submitted to DBpedia between May and July 2010 returned empty results. Instead of solely returning an empty set as the answer of a query, the system might help the user understand the reasons of this failure by providing him/her with a set of *Minimal Failing Subqueries* (MFSs). MFSs are the minimal parts of the query that failed. Enumerating them identifies all the failure causes of the query. In addition to MFSs-based information, alternative queries, called *MaXimal Succeeding Subqueries* (XSSs), might be suggested to the user by the system as well. XSSs are *successful* (i.e, non-failing) queries that have a maximal number of triple patterns from the initial query. Users can execute these XSSs to find alternative answers to their failing queries. In [10], the usefulness of XSSs as a feedback for the empty-answer problem has been evaluated by end-users, with an average satisfaction of 76%.

The problem of enumerating MFSs and XSSs of SPARQL queries expressed on traditional KBs is NP-hard [11], and has already been addressed by Fokou et al. [12]. In this paper, we consider a generalization of MFSs and XSSs in the context of uncertain KBs. We call αMFSs and αXSSs the failure causes and maximal succeeding subqueries of an uncertain query, i.e. a query that ignores results whose certainty degrees are below the provided threshold α. In this article, we first define the conditions under which the computation of MFSs and XSSs for traditional KBs can be directly adapted to αMFSs and αXSSs in an uncertain context. In this setting, the user has to provide the query threshold α. However, as she/he may not have an idea of the uncertainty level present in to the target KB, we also investigate the idea of suggesting relaxed queries with lower α thresholds. This kind of relaxation requires the computation of αMFSs and αXSSs for various other thresholds. To save computation time, some properties between αMFSs and αXSSs of different thresholds are established and exploited. Thus, and depending on which order the α values are considered, three approaches, called *Top-Down*, *Bottom-Up* and *Hybrid*, are discussed. We run several experiments on the WatDiv benchmark with Jena TDB[1] and Virtuoso [13], used as quadstores, to show the impact of our approaches compared to a baseline method.

[1] http://jena.apache.org/documentation/tdb/

This paper is an extension of our earlier conference work [14]. We have substantially developed, revised and improved our proposal. In particular, this article contains the following original contributions: (1) the proposition of a third approach (named *Hybrid*) to compute αMFS and αXSS, (2) the complete definitions and proofs of the properties on which our approaches are based, (3) a detailed analysis of existing work on the empty answer problem in the context of SPARQL and relational queries, (4) an evaluation of the complexity of our three approaches and (5) new experiments to evaluate their impact in different settings.

The paper is structured as follows. Section 2 provides some basic notions and formalizes the considered problem, along with a motivating and illustrative example. Section 3 defines the condition under which a previous work algorithm can be directly adapted to find the αMFSs and αXSSs of a failing RDF query for a given unique threshold α. Section 4 describes the three proposed approaches (*Top-Down*, *Bottom-Up* and *Hybrid*) to compute αMFSs and αXSSs for a set of thresholds. Section 5 provides a complexity analysis of these algorithms. Section 6 discusses the implementation and the experimental evaluation performed using the WatDiv benchmark. Section 7 details related work. Finally, we conclude and introduce some future work in Section 8.

2 Preliminaries and Problem Statement

In this section we define the notions required for the remainder of the paper. We use the notations on RDF and SPARQL given by Pérez et al. [15] and the trust model proposed by Hartig [7].

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* (or *triplestore*) stores a set of RDF triples (denoted by T_{RDF}) in a triples table or one of its variants. Each RDF triple has a *trust score* (or *certainty score*) representing the trustworthiness of the triple. This score is assigned with the function $tv : T_{RDF} \rightarrow [0, 1]$.

Example 1. Table 1 depicts an uncertain KB. This sample KB will be used throughout the paper.

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 $var(t) \subseteq V$ 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 $|Q|$ and its variables $var(Q) = \bigcup var(t_i)$.

Uncertain KB			
subject	**predicate**	**object**	**trust value**
b_1	author	Victor Hugo	0.5
b_1	editor	ACM	0.3
b_1	type	Book	0.9
b_1	nbPages	90	0.9
b_2	editor	Springer	0.7
b_2	type	Book	0.3
b_2	nbPages	90	0.7
b_3	type	Book	0.7
b_3	nbPages	88	0.7
b_4	author	Victor Hugo	0.5
b_4	type	Book	0.1
b_5	author	Abraham Lincoln	0.5
b_5	nbPages	90	0.3
b_6	author	Abraham Lincoln	0.3
b_6	editor	Springer	0.3
b_6	type	Book	0.3

Table 1: An uncertain RDF database D

Example 2. The following RDF query Q_1, expressed in SPARQL[2], has two variables $var(t) = \{?b, ?p\}$ and is composed of two triple patterns t_1 and t_2.

Q_1:
```
SELECT ?b ?a WHERE {
       ?b type Book .   (t1)
       ?b author ?a }   (t2)
```

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 in t its variables $var(t)$ by their mapping $\mu(var(t))$. 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: $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\}$. Let D be an RDF database and t be 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 (e.g. simple or RDF entailment regime). The examples and experiments presented in this article are based on the simple entailment regime. Let μ be a solution of the query $Q = t_1 \wedge \cdots \wedge t_n$ and $aggreg$ be an aggregation function (e.g, the minimum), the trust value of μ is defined by $tv(\mu, Q) = aggreg(tv(\mu(t_1)), \cdots, tv(\mu(t_n)))$. In

[2] To improve readability, we use names instead of URIs to identify query elements.

the context of uncertain KBs, the evaluation of Q over D returns trust weighted results filtered using a user-provided threshold α. This threshold is the minimum acceptable certainty value of the mapping, in the range $[0, 1]$. The evaluation of an uncertain query is therefore defined as: $[[Q]]_D^{\alpha} = \{\mu \in [[Q]]_D \mid tv(\mu) \geq \alpha\}$.

Example 3. Let us consider the evaluation of Q_1 for a degree $\alpha = 0.4$. Figure 1 illustrates the evaluation of each triple pattern $[[t_1]]_D$ and $[[t_2]]_D$, as well as their junction $[[Q_1]]_D = [[t_1]]_D \bowtie [[t_n]]_D$. The aggregation of trust values is then performed on the result of $[[Q_1]]_D$, based on the trust value associated with each mapping of single triple patterns. In this example, the chosen aggregation function is the minimum.

The evaluation of the uncertain query $[[Q_1]]_D^{0.4}$ would not include the last two mappings (that involve b_4 and b_6) since their aggregate trust values of respectively 0.1 and 0.3 are below 0.4.

$[[t_1]]_D$

?b	trust
b_1	0.9
b_2	0.3
b_3	0.7
b_4	0.1
b_6	0.3

$\ltimes$

$[[t_2]]_D$

?b	?a	trust
b_1	Victor Hugo	0.5
b_4	Victor Hugo	0.5
b_5	Abraham lincoln	0.5
b_6	Abraham lincoln	0.3

=

$[[t_1]]_D \ltimes [[t_2]]_D$

?b	?a	trust
b_1	Victor Hugo	min(0.9,0.5)=0.5
b_4	Victor Hugo	min(0.1,0.3)=0.1
b_6	Abraham lincoln	min(0.3,0.3)=0.3

Fig. 1: Evaluation process of Q_1

Notion of αMFSs and αXSSs 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 ($Q' \subset Q$).

Definition 1. *A Failing Subquery (MFIS) of a query Q is one of its failing subqueries that contains a minimal subset of its predicates. The set of all MFSs of a query Q, denoted by $mfs(Q)$, is therefore defined as:*

$$fis(Q) = \{Q^* \mid Q^* \subseteq Q \ \wedge \ [[Q^*]]_D = \emptyset \ \wedge \ \nexists\, Q' \subset Q^* \text{ such that } [[Q']]_D = \emptyset\}.$$

The se of MFIS is, all the minimal FIS. It's defined as:

$$mfis^k(Q) = \{Q^* \mid Q^* \subseteq faispartifis(Q) \ \wedge \nexists\, Q' \subset Q^* \text{ such that } [[Q']]_D^k \subset faitpartifis(Q)\emptyset\}.$$

Example 4. Let us now consider the evaluation of $[[Q_1]]_D^{0.6}$. The result of this query is empty as it does not include any mapping, either because their associated aggregate trust value is below 0.6 or because the mappings are not included in the evaluation of the non-uncertain query $[[Q_1]]_D$. Since $[[t_1]]_D^{0.6} \neq \emptyset$ and $[[t_2]]_D^{0.6} = \emptyset$, $mfs^{0.6}(Q_1) = \{t_2\}$, which means that any query that includes the predicate t_2 and this KB will fail for a threshold of 0.6.

Definition 2. *A Maximal Succeeding Subquery (XSS) of a query Q is one of its successful subqueries that contains a maximal subset of its predicates. The set of all XSSs of a query Q, denoted by $xss(Q)$, is therefore defined as:*

$$xss(Q) = \{Q^* \mid Q^* \subseteq Q \wedge [[Q^*]]_D \neq \emptyset \wedge \nexists\, Q' \text{ such that } Q^* \subset Q' \wedge [[Q']]_D \neq \emptyset\}.$$

By extension, the set of all αMFSs of a query Q is defined as:

$$xss^{\alpha}(Q) = \{Q^* \mid Q^* \subseteq Q \wedge [[Q^*]]_D^{\alpha} \neq \emptyset \wedge \nexists\, Q' \text{ such that } Q^* \subset Q' \wedge [[Q']]_D^{\alpha} \neq \emptyset\}.$$

Example 5. Following the previous example, $xss^{0.6}(Q_1) = \{t_1\}$, which means that there are no successful subqueries of Q_1 whose predicates are a strict superset of $\{t_1\}$ for a threshold of 0.6. The query $Q^* = t_1$ is therefore a candidate relaxed query.

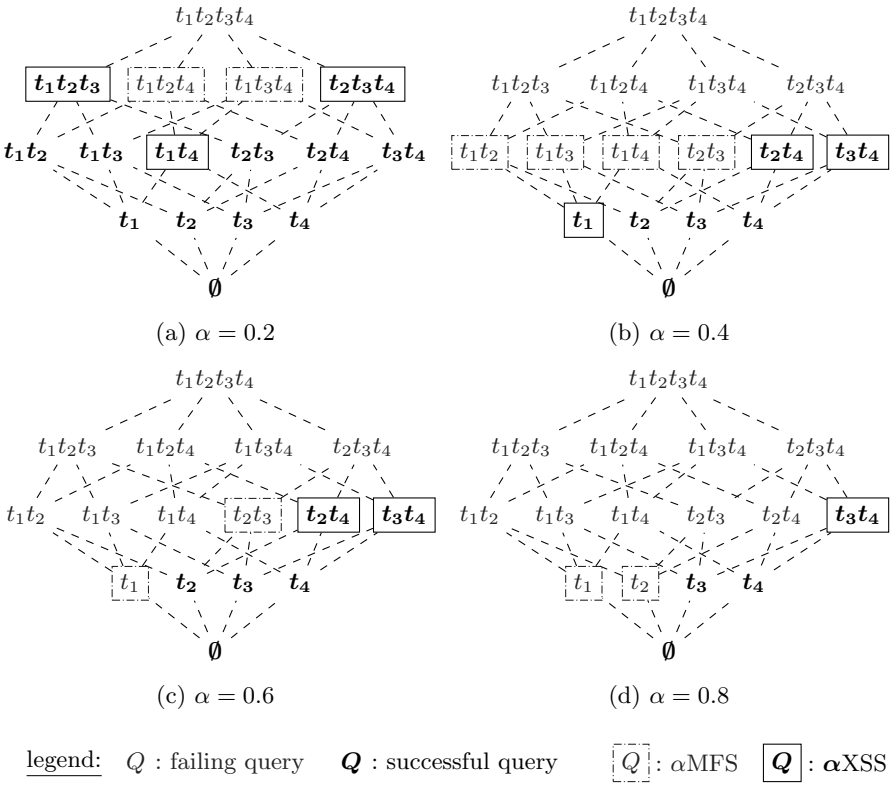
Comprehensive example Let us consider the following query Q_2 that searches for `Books` edited by `Springer` and authored by `Abraham Lincoln` with their number of pages. We denote this query by $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$ (or $t_1t_2t_3t_4$ in short).

Q_2:
```
SELECT ?b ?p WHERE {
    ?b author "Abraham Lincoln".   (t1)
    ?b editor "Springer" .         (t2)
    ?b type Book .                 (t3)
    ?b nbPages ?p }                (t4)
```

Figure 2 depicts the αMFSs and αXSSs of the query Q_2 for different thresholds on the lattice of subqueries of Q_2. Note that the enumeration of all αMFSs and αXSSs is sufficient to determine the status (successful or failing) of *every* subquery of Q_2.

We first consider that the user wants results with a certainty degree of at least 0.8. In our example, this query will fail; but, thanks to the 0.8 αMFSs ($mfs^{0.8}(Q_2) = \{t_1, t_2\}$) and αXSSs ($xss^{0.8}(Q_2) = \{t_3t_4\}$), we may provide the following explanations: for a certainty degree of 0.8, there is no item authored by `Abraham Lincoln` (t_1) and no item edited by `Springer` (t_2), meaning that any query that includes these predicates will fail for this threshold but there are some `Books` with their number of pages (t_3t_4), which is the only candidate relaxed query.

If we compute the αMFSs and αXSSs for lower degrees (e.g., 0.2, 0.4 and 0.6), we may find significant alternative queries and explanations for the user. For example the αXSSs may be used to provide the following explanations:



(a) $\alpha = 0.2$ (b) $\alpha = 0.4$

(c) $\alpha = 0.6$ (d) $\alpha = 0.8$

Fig. 2: Lattice of subqueries with failing and succeeding queries for different α

1. since $xss^{0.6}(Q_2) = \{t_2t_4, t_3t_4\}$, the user may find in the KB some items edited by `Springer` with their number of pages (t_2t_4) if he agrees to lower his certainty threshold, the previously identified αXSS (t_3t_4) is also still a viable alternative query for 0.6;
2. since $xss^{0.4}(Q_2) = \{t_1, t_2t_4, t_3t_4\}$, they user may find items authored by `Abraham Lincoln` (t_1) with a degree of 0.4;
3. $xss^{0.2}(Q_2) = \{t_1t_2t_3, t_1t_4, t_2t_3t_4\}$, which means that there are some books edited by `Springer` and authored by `Abraham Lincoln` ($t_1t_2t_3$), items authored by `Abraham Lincoln` with their number of pages (t_1t_4) and some books edited by `Springer` with their number of pages ($t_2t_3t_4$) for that certainty threshold.

As for αMFSs, $mfs^{0.6}(Q_2) = \{t_1, t_2t_3\}$, which means that, for this threshold, there are still no items authored by `Abraham Lincoln` and, while there are items edited by `Springer` (t_2 is not an αMFSs), there are no `Books` edited by `Springer` (t_2t_3).

This feedback may help the user have a better understanding of the uncertain KB content, reformulate her/his query or adjust her/his expectation. To provide this feedback, an efficient approach that computes αMFSs and αXSSs for a set of thresholds is needed. We first consider the problem of computing αMFSs and αXSSs for a single threshold α.

Problem statement We are concerned with computing $mfs^{\alpha_i}(Q)$ and $xss^{\alpha_i}(Q)$ of a failing RDF query Q over uncertain KBs efficiently for a set of thresholds $\alpha_i \in \{\alpha_1, \cdots, \alpha_n\}$.

3 αMFSs and αXSSs Computation for a Single Threshold α

In Fokou et al. [12], a *Lattice-Based Approach* (LBA) to compute MFSs and XSSs of a SPARQL query is proposed. As we show in Section 3.1, this approach can be directly adapted to compute the αMFSs and αXSSs of a query for a given α. This approach, called αLBA, has the same algorithmic complexity as LBA (see [12]). However the αLBA approach relies on an assumption that does not always hold in the context of uncertain KBs. Thus, we define, in Section 3.2, the condition under which αLBA can be used to compute the αMFSs and αXSSs of a query for a given α.

3.1 αLBA Approach

We present in this section the main principles and algorithms of αLBA as a direct adaptation of LBA in the uncertain KBs setting. αLBA explores the lattice of subqueries of a query Q by following a three-step procedure.

Step 1: Find an αMFS Q^* of Q Following Algorithm 1, αLBA removes iteratively each triple pattern t_i from Q, resulting in the proper subquery Q'. If Q' fails for α, then Q' contains an αMFS. Conversely, if Q' succeeds, then each αMFS of Q contains t_i. The proof of this property relies on the fact that a successful query cannot contain a failing query [12].

Algorithm 1: Find an αMFS of a failing RDF query Q

```
FindAn𝛼MFS(Q, D, 𝛼)
    inputs : A failing query Q = t1 ∧ ... ∧ tn; an RDF database D;
             a threshold 𝛼
    output: An 𝛼MFS of Q denoted by Q*
1   Q* ← ∅; Q' ← Q;
2   foreach triple pattern ti ∈ Q do
3       Q' ← Q' − ti;
4       if [[Q' ∧ Q*]]^𝛼_D ≠ ∅ then
5           Q* ← Q* ∧ ti;
6   return Q*;
```

Example 6. To illustrate this algorithm, we consider our running example with $\alpha = 0.2$ (see Figure 2(a)). The algorithm removes the triple pattern t_1 from the initial query resulting in the subquery $t_2t_3t_4$. As this query succeeds, t_1 is contained in the searched αMFS Q^*. The algorithm removes t_2 and executes the query $t_1t_3t_4$. This query fails and thus, the algorithm searches Q^* in $t_1t_3t_4$. t_3 is removed leading to the subquery t_1t_4 which succeeds, meaning that t_3 is included in the αMFS. Finally, t_4 is removed and the subquery t_1t_3 succeeds. Thus, t_4 is also an element of Q^*. The algorithm stops and returns the result $Q^* = t_1t_3t_4$.

Step 2: Compute *Potential αXSSs* i.e., the maximal queries that do not include the αMFS previously found. The set of *potential αXSSs* is denoted by $pxss(Q, Q^*)$ and can be computed as follows:

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

Example 7. Following our running example, $pxss(Q, Q^*) = \{t_2t_3t_4, t_1t_2t_4, t_1t_2t_3\}$.

This second step is based on the fact that all superqueries of $Q*$ (i.e., queries that include the αMFS Q^* found in the previous step) return an empty set of answers and thus, can be pruned from the search space. In the context of uncertain KBs, this property is true only if a successful query cannot contain a failing query for the given α, which will be discussed in Section 3.2.

Step 3: Execute the Potential αXSSs If a subquery is found during step 2 succeeds, it is then an αXSS. Otherwise, we apply the two previous steps on this subquery to find a new αMFS and update the potential αXSSs. This is illustrated by Algorithm 2. It is worth noting that this algorithm avoids finding the same αMFSs several times (lines 11-13).

Example 8. In our running example, the two potential αXSSs $t_2t_3t_4$ and $t_1t_2t_3$ succeed and thus, they are the αXSSs of Q. The potential αXSSs $t_1t_2t_4$ fails and therefore contains an αMFS, that FindAnαMFS determines to be the query itself. During this step, t_1t_4 is added to the set of potential αXSSs (line 13 of algorithm 2) since $pxss(t_1t_2t_4,\, t_1t_2t_4) = \{t_1t_2, t_1t_4, t_2t_4\}$, and both t_1t_2 and t_2t_4 are subqueries of previously identified αXSSs. the last potential αXSS t_1t_4 succeeds and is therefore an αXSS.

Algorithm 2: Find the αMFSs and αXSSs of a query Q

α**LBA(**Q**,** D**,** α**)**
inputs : A failing query $Q = t_1 \wedge ... \wedge t_n$; an RDF database D; a threshold α
outputs: The αMFSs and αXSSs of Q
1 $Q^* \leftarrow$FindAnαMFS(Q, D, α);
2 $pxss \leftarrow pxss(Q, Q^*)$;
3 $mfs^\alpha(Q) \leftarrow \{Q^*\}$; $xss^\alpha(Q) \leftarrow \emptyset$;
4 **while** $pxss \neq \emptyset$ **do**
5 $Q' \leftarrow pxss.element()$; `// choose an element of` $pxss$
6 **if** $[[Q']]^\alpha_D \neq \emptyset$ **then** `//` Q' `is an` α`XSS`
7 $xss^\alpha(Q) \leftarrow xss^\alpha(Q) \cup \{Q'\}$; $pxss \leftarrow pxss - \{Q'\}$;
8 **else** `//` Q' `contains an` α`MFS`
9 $Q^{**} \leftarrow$FindAnαMFS(Q', D, α);
10 $mfs^\alpha(Q) \leftarrow mfs^\alpha(Q) \cup \{Q^{**}\}$;
11 **foreach** $Q'' \in pxss$ *such that* $Q^{**} \subseteq Q''$ **do**
12 $pxss \leftarrow pxss - \{Q''\}$;
13 $pxss \leftarrow pxss \cup \{Q_j \in pxss(Q'', Q^{**}) \mid \nexists Q_k \in pxss \cup xss^\alpha(Q)$ *such that* $Q_j \subseteq Q_k\}$;
14 **return** $\{mfs^\alpha(Q), xss^\alpha(Q)\}$;

3.2 Aggregate Function Condition

As we have seen in the previous section, the αLBA approach relies on the fact that a successful query cannot contain a failing query. In the context of uncertain KBs, depending on the aggregate function ($aggreg$[3]) chosen, this property

[3] Aggreg is the function used for assigning trust values to query results.

does not always hold. Following the example query Q_1 whose evaluation is illustrated in Figure 1, Figure 3 provides the results of this query and one of its subqueries Q'_1 on the uncertain RDF database given in Table 1, for $\alpha = 0.6$. For the *max* and *avg* aggregate functions, the query Q_1 is successful since $max(0.9, 0.5) = 0.9 \geq 0.6$ and $avg(0.9, 0.5) = 0.7 \geq 0.6$ while its subquery Q'_1 fails in both cases: $max(0.5) = 0.5 < 0.6$ and $avg(0.5) = 0.5 < 0.6$, which contradicts our hypothesis. Thus, the αLBA algorithm cannot be used with these aggregate functions. As proven below, this algorithm can be applied only if the aggregate function *aggreg*, is monotonically decreasing with respect to the subset partial order.

Q_1:
```
SELECT ?b ?a WHERE {
      ?b type Book .
      ?b author ?a }
```
Q'_1:
```
SELECT ?b ?a WHERE {
      ?b author ?a }
```

aggreg	$[[Q']]_D^{0.6}$	$[[Q]]_D^{0.6}$
min	$\emptyset$	$\emptyset$
max	$\emptyset$	$\{b_1\}$
$\prod$	$\emptyset$	$\emptyset$
avg	$\emptyset$	$\{b_1\}$

Fig. 3: Results of subqueries evaluation for different aggregate functions

Definition 3. *Let* $aggreg : [0,1]^n \rightarrow [0,1]$ *be an aggregate function, aggreg is monotonically decreasing with respect to set*[4] *inclusion if for all sets A and B* $\in [0,1]^n$, $A \subseteq B \Rightarrow aggreg(A) \geq aggreg(B)$.

As examples of monotonically decreasing aggregate functions, we can cite the *minimum* or the *product* on values in $[0, 1]$.

Proposition 1. *Let aggreg be monotonically decreasing. If a proper subquery* Q' *of* Q *fails for a given* α *(using the aggreg function) then* Q *also fails for* α.

Proof. We consider a query $Q = t_1 \wedge \cdots \wedge t_n$ and its proper subquery $Q' = t_i \wedge \cdots \wedge t_j$ ($\{i, \cdots, j\} \subset \{1, \cdots, n\}$). Assume that $[[Q']]_D^\alpha = \emptyset$ and $[[Q]]_D^\alpha \neq \emptyset$. Thus, $\exists \mu \in [[Q]]_D^\alpha$. Since $[[Q]]_D^\alpha \subseteq [[Q]]_D$ and $[[Q]]_D \subset [[Q']]_D$, we have $\mu_{|var(Q')} \in [[Q']]_D$ where $\mu_{|var(Q')}$ is the restriction of the function μ to the variables of Q'. By definition, $tv(\mu, Q) = aggreg(tv(\mu(t_1)), \cdots, tv(\mu(t_n))) \geq \alpha$ and $tv(\mu_{|var(Q')}, Q') = aggreg(tv(\mu(t_i)), \cdots, tv(\mu(t_j)))$ (indeed, $tv(\mu, Q') = tv(\mu_{|var(Q')}, Q')$). Since *aggreg* is monotonically decreasing, $aggreg(tv(\mu(t_i)), \cdots, tv(\mu(t_j))) \geq aggreg(tv(\mu(t_1)), \cdots, tv(\mu(t_n))) \geq \alpha$. As a consequence, $tv(\mu_{|var(Q')}, Q') \geq \alpha$ and since $\mu_{|var(Q')} \in [[Q']]_D$ we deduce that $\mu_{|var(Q')} \in [[Q']]_D^\alpha$. This contradicts the assumption that Q' fails.

In this section, we have shown that if the aggregate function *aggreg*, used for assigning trust values to query results, is monotonically decreasing with respect to the subset partial order, then the αLBA approach can be applied to find the

[4] For simplicity, this definition is restricted to sets but could be extended to multisets.

αMFSs and αXSSs for the given threshold. In the next section, we consider the problem of finding the αMFSs and αXSSs for a set of thresholds. As highlighted in Section 2, these additional results could be useful to help users reformulate their queries and/or adjust their expectations.

4 αMFSs and αXSSs Computation for Different Thresholds α

To find αMFSs and αXSSs for a set of $\alpha_i \in \{\alpha_1, \cdots \alpha_n\}$, one can execute the αLBA algorithm for each α_i. This baseline method is named NLBA. In this section, we discuss different improvements of this approach. The idea is that the αMFSs and αXSSs discovered for a given threshold provide a set of hints to deduce some αMFSs and αXSSs for a higher (or lower) threshold. We start by investigating a bottom-up approach, from a lower to a higher threshold.

4.1 Bottom-Up Approach

In this section, we consider two thresholds α_i and α_j such that $\alpha_i < \alpha_j$. If Q^* is an α_iMFSs of the query Q, then Q^* also fails for α_j. However, this subquery is not necessarily minimal for α_j and therefore might not be an α_jMFS. The following proposition provides a condition under which an α_iMFS is also an α_jMFS.

Proposition 2. *Let α_i and α_j be two thresholds such that $\alpha_i < \alpha_j$ and Q^* be an α_iMFS of Q on an RDF database D. If $|Q^*| = 1$, then Q^* is also an α_jMFS of Q.*

Proof. If Q^* is an α_iMFS of Q on a dataset D, then $[[Q^*]]_D^{\alpha_i} = \emptyset$. Since $\alpha_i < \alpha_j$, we also have $[[Q^*]]_D^{\alpha_j} = \emptyset$. Q^* is minimal ($|Q^*| = 1$) and failing for α_j, thus Q^* is an α_jMFS of Q.

Thus, an α_iMFS Q^* of Q such that $|Q^*| = 1$ is an α_jMFS of Q. We now consider the general case where $|Q^*| > 1$. As pointed out previously, for a subquery Q^* to be an α_jMFS of a query Q, all its proper subqueries have to succeed. As stated in proposition 2, this property is always true if the query contains a single triple pattern. Checking if a query has a single triple pattern does not require any database access. Thus, we check this case first and put all discovered α_jMFS of Q in a set of *discovered αMFSs* denoted by $dmfs^{\alpha_j}(Q)$. Otherwise, proving that Q^* is an α_jMFS requires checking that all its subqueries succeed, by executing those $|Q^*|$ queries. In the worst case where Q^* is not an α_jMFS, $|Q^*|$ queries are executed without finding any α_jMFS. Conversely, the algorithm FindAnαMFS of αLBA (Algorithm 1) also requires $|Q^*|$ queries but guarantees that an αMFS will be found. Thus, our approach favors the algorithm FindAnαMFS over executing the subqueries of the α_iMFS to discover new α_jMFSs. This approach avoids starting over from the initial query to find new α_jMFSs and therefore executes less subqueries compared with the baseline method NLBA.

We have seen the properties that can be leveraged to deduce some α_jMFSs of Q from its α_iMFSs. We now consider the case of the αXSSs. An α_iXSS of Q may fail for α_j. The following proposition shows that if it succeeds, it is then an α_jXSS of Q.

Proposition 3. *Let α_i and α_j be two thresholds such that $\alpha_i < \alpha_j$ and Q^* be an α_iXSS of Q on an RDF database D. If $[[Q^*]]_D^{\alpha_j} \neq \emptyset$, then Q^* is an α_jXSS of Q.*

Proof. If Q^* is an α_iXSS of Q on an RDF database D, then all its superqueries are failing for α_i (otherwise, it is not maximal). As $\alpha_i < \alpha_j$, these superqueries also fail for α_j. If $[[Q^*]]_D^{\alpha_j} \neq \emptyset$, Q^* is successful and maximal for α_j. Thus, Q^* is an α_jXSS of Q.

Thus, discovering if an α_iXSS is also an α_jXSS only requires the execution of a single query (the α_iXSS with the new threshold α_j). This enables us to find a set of discovered α_jXSSs, denoted by $dxss^{\alpha_j}(Q)$. If the α_iXSS fails for α_j, we can still use it to find an α_jMFS thanks to the algorithm FindAnαMFS. Thus, the execution of the α_iXSS for the new threshold α_j is always worthwhile: if it succeeds, it is an α_jXSS; if it fails, we use it to find an α_jMFS.

Algorithm 3 presents our complete approach to find some α_jMFSs and α_jXSSs from the set of α_iMFSs and α_iXSSs. All α_iMFSs that have one triple pattern (*oneAtom*) are inserted in $dmfs^{\alpha_j}(Q)$ (line 1). Then, the algorithm iterates over the α_iMFSs with at least two triple patterns (the set FQ). It searches an α_jMFS Q^* in a query Q' of FQ with the FindAnαMFS algorithm (line 6). Then, it removes all the failing queries of FQ that contain Q^* since they cannot be minimal. This process stops when all the queries in FQ have been processed (they have been either used to find an α_jMFS or removed as they contain a found α_jMFS).

We then consider all the α_iXSSs that do not contain a discovered α_jMFS (the set $PXXS$). If a query Q' of this set succeeds, this is an α_jXSS (property 3. Otherwise we use it to find to find an α_jMFS with the FindAnαMFS algorithm (lines 16-17) and remove the queries of the set $PXSS$ that contain this discovered α_jMFS (lines 18-19).

Once some α_jMFSs and α_jXSSs have been discovered, an optimized version of αLBA is executed that takes these discovered α_jMFSs and α_jXSSs as inputs (see Algorithm 4). This algorithm computes the potential αXSSs ($pxss$) that do not contain any of the discovered αMFSs (lines 2-6). It removes from this set the discovered αXSSs (line 7). Then, it iterates over the set $pxss$ as done with the original version of αLBA (see Algorithm 2).

Example 9. To illustrate the Bottom-Up approach, we consider our running example (see Figure 2). First, we execute the αLBA algorithm for the 0.6 threshold in order to find the 0.6 αMFSs and αXSSs (Figure 2.c). Then, the Bottom-Up algorithm discovers the 0.8 αMFSs and αXSSs. As t_1 is a 0.6 αMFS and has a single triple pattern, this is a 0.8 αMFS (proposition 2). The other 0.6 αMFS is t_2t_3. As this query is necessarily failing for 0.8, we use the $FindAn\alpha MFS$ algorithm to identify that t_2 is a 0.8 αMFSs. Considering the 0.6 αXSSs, only

Algorithm 3: Find some α_jMFSs and α_jXSSs for Bottom-Up

DiscoverαMFSXSS($mfs^{\alpha_i}(Q)$, $xss^{\alpha_i}(Q)$ D, α_j)

inputs : The α_iMFSs $mfs^{\alpha_i}(Q)$ of a query Q for a threshold α_i;
the α_iXSSs $xss^{\alpha_i}(Q)$ of a query Q for a threshold α_i;
an RDF database D; a threshold $\alpha_j > \alpha_i$

outputs: A set of α_jMFSs of Q denoted by $dmfs^{\alpha_j}(Q)$;
A set of α_jXSSs of Q denoted by $dxss^{\alpha_j}(Q)$;

1 $oneAtom \leftarrow \{Q_a \in mfs^{\alpha_i}(Q) \mid |Q_a| = 1\}$;
2 $dmfs^{\alpha_j}(Q) \leftarrow oneAtom$;
3 $FQ \leftarrow mfs^{\alpha_i}(Q) - oneAtom$;
4 **while** $FQ \neq \emptyset$ **do**
5 $Q' \leftarrow fQ.dequeue()$;
6 $Q^* \leftarrow FindAn\alpha MFS(Q', D, \alpha_j)$;
7 $dmfs^{\alpha_j}(Q) \leftarrow dmfs^{\alpha_j}(Q) \cup \{Q^*\}$;
8 **foreach** $Q'' \in FQ$ *such that* $Q^* \subseteq Q''$ **do**
9 $FQ \leftarrow FQ - \{Q''\}$;
10 $PXXS \leftarrow \{Q_a \in xss^{\alpha_i}(Q) \mid \nexists Q^* \in dmfs^{\alpha_j}(Q)$ *such that* $Q^* \subset Q_a\}$;
11 **while** $PXXS \neq \emptyset$ **do**
12 $Q' \leftarrow PXXS.dequeue()$;
13 **if** $[[Q']]_D^{\alpha_j} \neq \emptyset$ **then** `// ` Q' `is an` α_j`XSS`
14 $dxss^{\alpha_j}(Q) \leftarrow dxss^{\alpha_j}(Q) \cup \{Q'\}$;
15 **else** `// ` Q' `contains an` α_j`MFS`
16 $Q^* \leftarrow FindAn\alpha MFS(Q', D, \alpha_j)$;
17 $dmfs^{\alpha_j}(Q) \leftarrow dmfs^{\alpha_j}(Q) \cup \{Q^*\}$;
18 **foreach** $Q'' \in PXSS$ *such that* $Q^* \subseteq Q''$ **do**
19 $PXSS \leftarrow PXSS - \{Q''\}$;
20 **return** $\{dmfs^{\alpha_j}(Q), dxss^{\alpha_j}(Q)\}$;

Algorithm 4: Optimized version of αLBA

Optimized-αLBA(Q, D, α, $dmfs^{\alpha}(Q)$, $dxss^{\alpha}(Q)$)

inputs : A failing query Q; an RDF database D; a threshold α
a set of αMFSs of Q denoted by $dmfs^{\alpha}(Q)$;
a set of αXSSs of Q denoted by $dxss^{\alpha}(Q)$;

outputs: The αMFSs and αXSSs of Q

1 $mfs^{\alpha}(Q) \leftarrow dmfs^{\alpha}(Q)$; $xss^{\alpha}(Q) \leftarrow dxss^{\alpha}(Q)$;

2 $Q^* \leftarrow dmfs^{\alpha}(Q).dequeue()$; $pxss \leftarrow pxss(Q, Q^*)$;

3 **foreach** $Q^* \in dmfs^{\alpha}(Q)$ **do**

4 **foreach** $Q' \in pxss$ *such that* $Q^* \subseteq Q'$ **do**

5 $pxss \leftarrow pxss - \{Q'\}$;

6 $pxss \leftarrow pxss \cup \{Q_i \in pxss(Q', Q^*) \mid \nexists Q_j \in pxss \cup xss^{\alpha}(Q) : Q_i \subseteq Q_j\}$;

7 $pxss \leftarrow pxss - dxss^{\alpha}(Q)$;

8 **while** $pxss \neq \emptyset$ **do**

`// same as the lines 5-13 of` α`LBA`

9 **return** $\{mfs^{\alpha}(Q), xss^{\alpha}(Q)\}$;

t_3t_4 is executed as t_2t_4 contains a previously found 0.8 αMFS (t_2). As t_3t_4 is successful for 0.8, this is a 0.8 αXSSs (proposition 3).

The discovered αMFSs and αXSSs given as inputs to the Algorithm 4 are respectively: $dmfs^{\alpha}(Q) = \{t_1, t_2\}$ and $dxss^{\alpha}(Q) = \{t_3t_4\}$. From these sets of discovered αMFSs and αXSSs, the Algorithm 4 will finally find that there are no potential αXSSs left (lines 1-7 of this algorithm) and thus, that all the 0.8 αMFSs and αXSSs have been found. Figure 4 gives an overview of this sequence of algorithms.

The Bottom-Up approach discovers some αMFSs and αXSSs (and thus, improves over executing αLBA for each threshold α), if some αXSSs remain the same for increasing threshold values or if the αMFSs have a single triple pattern. Otherwise, it will nonetheless use previously discovered αMFSs as starting points instead of starting over from the original query.

Cache Management In its original version, the LBA approach maintains a cache of queries identified as successful (resp., failing) [12]. So, before executing a subquery, this algorithm first checks whether it is a subquery (resp., superquery) of one of the queries contained in this cache. If this is the case, the subquery succeeds (resp., fails). Our approach extends this idea as follows. The successful (resp., failing) queries are associated with a threshold corresponding to the maximum (resp., minimum) threshold for which the query succeeds (resp., fails). Before executing a subquery for a given α, we first check if this is a subquery (resp., superquery) of one of the queries contained in the cache and if the associated threshold is greater than (resp., less than) or equal to α. If this is the case, the query succeeds (resp., fails). As our approach also discovers some αXSS

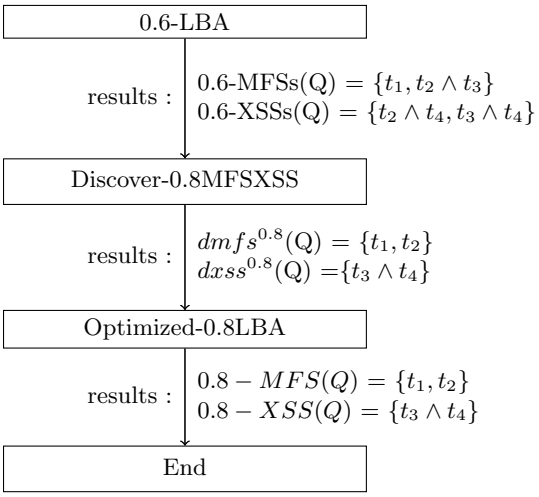


Fig. 4: Illustration of the Bottom-Up approach for two thresholds (0.6 and 0.8)

(resp., αMFSs) for a given α, we add to the cache all the direct/parent superqueries (resp., subqueries) of these αXSSs (resp., αMFSs) as they are necessary failing (resp., succeeding) for α.

4.2 Top-Down Approach

We now consider a Top-Down approach that computes the αMFSs and αXSSs with threshold values in a decreasing order. Thanks to the duality relation that holds between αMFS and αXSS, the properties used in this approach are dual to the ones used in the bottom-up approach. Thus, we only introduce them informally. Let α_i and α_j be two thresholds such that $\alpha_i > \alpha_j$, the following properties hold:

- the α_iMFSs that fail for α_j are α_jMFSs;
- the α_iXSSs of size $|Q|-1$ are α_jXSSs;
- the α_iXSSs of size $< |Q|-1$ also succeed for α_j and thus contain an α_jXSS. This α_jXSS is found using Algorithm 5 $FindAn\alpha XSS$ which is the dual of the Algorithm 1 $FindAn\alpha MFS$.

Once a set of α_jMFSs and α_jXSS have been found based on the aforementioned properties, the Optimized-αLBA algorithm (Algorithm 4) is executed to find the complete set of α_jMFSs and α_jXSS. This approach will improve over executing αLBA for each threshold α if some αMFSs remain the same for decreasing threshold values or if the αXSSs have a size of $|Q|-1$. Otherwise, it will nonetheless use previously discovered αXSSs as starting points instead of starting over from the original query.

Example 10. We illustrate the Top-Down approach by showing how it computes the 0.6 αMFSs and αXSSs of our running example (see Figure 2.*c*), knowing

Algorithm 5: Find an αXSS of Q from a successful subquery

FindAnαXSS(Q, Q^*, D, α)
inputs : The initial query $Q = t_1 \wedge ... \wedge t_n$;
a successful subquery Q^* of Q;
an RDF database D;
a threshold α;
output: An αXSS of Q denoted by Q'
1 $Q' \leftarrow Q^*$;
2 **foreach** *triple pattern* $t_i \in (Q - Q^*)$ **do**
3 $Q' \leftarrow Q' \wedge t_i$;
4 **if** $[[Q']]_D^\alpha = \emptyset$ **then**
5 $Q' \leftarrow Q' - t_i$;
6 **return** Q';

the ones for 0.8 (Figure 2.d). Firstly we execute the αLBA algorithm for the 0.8 threshold in order to find the 0.8 αMFSs and αXSSs. Secondly the Top-Down algorithm discovers the 0.6 αMFSs and αXSSs. The 0.8 αMFS t_1 is failing for 0.6. Thus, it is a 0.6 αMFS. The 0.8 αXSS t_3t_4 is necessarily successful for 0.6. We use it as parameter of the FindAnαXSS algorithm to find that t_3t_4 is also a 0.6 αXSS. Thus, thanks to the properties used by Top-Down, the discovered 0.6 αMFSs and αXSSs are respectively $dmfs^\alpha(Q) = \{t_1\}$ and $dxss^\alpha(Q) = \{t_3t_4\}$. Finally, they are used as parameters of the optimized version of αLBA (Algorithm 4) that will find the other 0.6 αMFS (t_2t_3) and αXSS (t_2t_4). Figure 5 gives an overview of this sequence of algorithms.

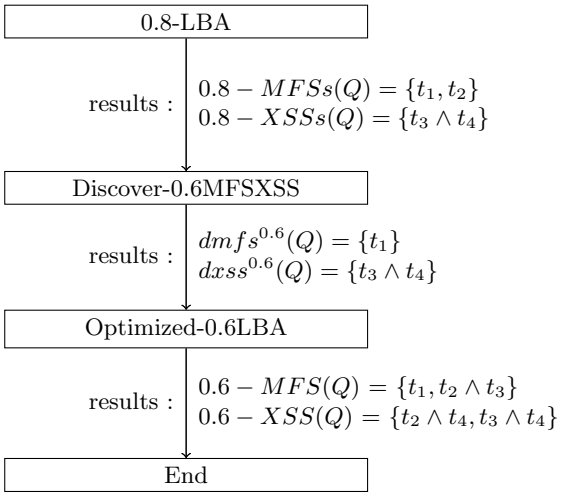


Fig. 5: Illustration of the Top-Down approach for two thresholds (0.8 and 0.6)

Thus, the Top-Down approach can be seen as the dual of the Bottom-Up approach. We now propose an approach that leverages the properties of both of these two approaches.

4.3 Hybrid Approach

The idea of the Hybrid approach is to combine the properties used in Bottom-Up and Top-Down to discover the greatest possible number of αMFSs and αXSSs. For an ordered sequence of thresholds $\{\alpha_1, \cdots \alpha_n\}$, the Hybrid algorithm first considers the lowest threshold α_1, followed by the greatest threshold α_n. Next, it iterates over the sequence $\{\alpha_1, \cdots \alpha_n\}$ by considering the middle threshold α_i where $i = \lfloor \frac{n+1}{2} \rfloor$. The algorithm then recursively considers thresholds on (1) the subset of thresholds lower than α_i: $\{\alpha_1, \cdots \alpha_i\}$ with its threshold in the middle position, and (2) the subset of thresholds greater than α_i: $\{\alpha_i, \cdots \alpha_n\}$ with its threshold in the middle position, until the αMFSs and αXSSs are computed for every threshold. In our running example with the thresholds $\{0.2, 0.4, 0.6, 0.8\}$, Hybrid considers these thresholds in the following order: $\{0.2, 0.8, 0.4, 0.6\}$. Thanks to this order, when searching the αMFSs and αXSSs for the thresholds 0.4 and 0.6, Hybrid has access to the αMFSs and αXSSs of both a lower and greater degree. Thus, it can benefit from the properties used in Bottom-Up and Top-Down to discover some αMFSs and αXSSs. Moreover, we use the following specific properties to find some additional αMFSs and αXSSs.

Proposition 4. *Let α_i, α_j and α_k be three thresholds such that $\alpha_i < \alpha_j < \alpha_k$. If a query Q^* is both an α_iMFS and α_kMFS of Q, then Q^* is an α_jMFS of Q. Similarly, if a query Q^* is both an α_iXSS and α_kXSS of Q, then Q^* is an α_jXSS of Q.*

Proof. If Q^* is an α_iMFS of Q, then Q^* necessarily fails for α_j. Moreover, if Q^* is also an α_kMFS of Q, then all its subqueries succeed for α_k and thus, also for α_j. We have proved that Q^* is both failing and minimal for α_j and thus, that Q^* is an α_jMFS. The corresponding property on αXSSs is proved in a similar way.

Thus, the Hybrid approach allows discovering a set of αMFSs and αXSS by using the properties of Bottom-Up and Top-Down as well as the property 4. As in previous approaches, the Optimized-αLBA algorithm (Algorithm 4) is executed using the discovered αMFSs and αXSS to find the complete set of αMFSs and αXSS for the considered threshold.

Example 11. We illustrate the Hybrid approach on our running example by showing how it computes the 0.6 αMFSs and αXSSs (see Figure 2.c), knowing the ones for 0.4 (Figure 2.b) and 0.8 (Figure 2.d). First, we execute the αLBA algorithm for the 0.4 and 0.8 threshold in order to find the 0.4/0.8 αMFSs and αXSSs. Then, the Top-Down algorithm discovers the 0.6 αMFSs and αXSSs. Hybrid finds that t_3t_4 is an αXSS for 0.4 and 0.8, thus it is an 0.6 αXSS (proposition 4). Then, it searches for 0.4 αMFSs with a single triple pattern and 0.8

αXSSs that have 3 triple patterns ($|Q| - 1$). As there are none in our example, it continues by searching for the 0.8 αMFSs that fail for 0.6 (property of the Top-Down approach). This is the case for t_1, which is a 0.6 αMFS. Similarly, it searches for the 0.4 αXSSs that succeed for 0.6 (property of the Bottom-Up approach) and finds the 0.6 αXSS t_2t_4. Next, it uses the algorithm FindAnαMFS with the 0.4 αMFSs. In our example, this algorithm is only applied to t_2t_3 as the other ones contain t_1 (a 0.6 αMFS). It finds that t_2t_3 is a 0.6 αMFS. Conversely, it uses the algorithm FindAnαXSS with the 0.8 αXSSs. In our example, all the 0.8 αXSSs have already been used. Finally, thanks to the properties of Bottom-Up and Top-Down as well as the proposition 4, the Algorithm 4 will find that all the 0.6 αMFSs and αXSSs had been discovered. Figure 6 gives an overview of this sequence of algorithms.

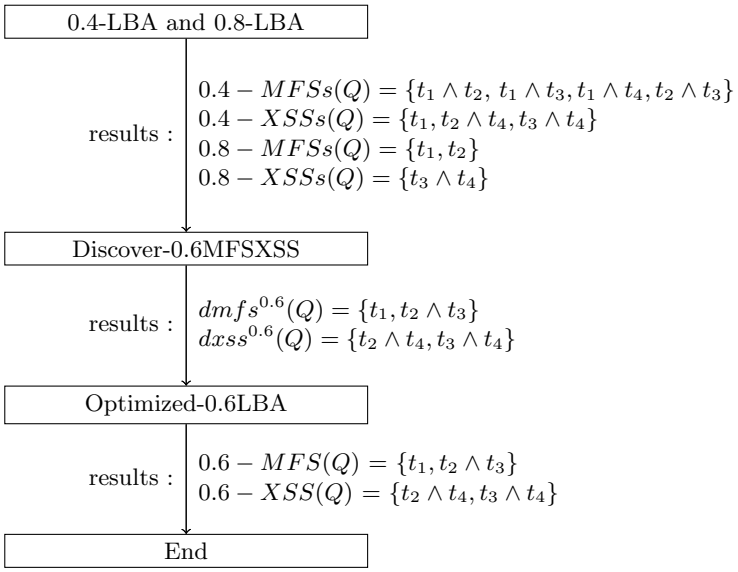


Fig. 6: Illustration of the Hybrid approach for three thresholds (0.4, 0.8 and 0.6)

5 Complexity of finding all αMFSs and αXSSs

In this section, we consider the time complexity of our algorithms as the number of executed queries, which is the time-consuming part. This analysis does not take into consideration individual response times of each queries (we assume that each execution of a query costs a time unit). Since the complexity of finding all αMFSs and αXSSs is exponential with respect to the number of predicates of the initial failing query [11], we evaluate this complexity with respect to the size

of the output. It has been shown that any algorithm computing αMFSs and αXSSs must use at least $|mfs^{\alpha}(Q)| + |xss^{\alpha}(Q)|$ queries [16].

NLBA We first analyse the complexity of the baseline method.

Theorem 1. *αLBA (algorithm 2) executes at most $|xss^{\alpha}(Q)| + |Q| * |mfs^{\alpha}(Q)|$ queries.*

Proof. FindAnαMFS (algorithm 1) discovers a new αMFS during each execution. FindAnαMFS executes exactly $|Q|$ queries (lines 2 to 4). For each iteration (line 4) of αLBA (algorithm 2), this algorithm finds either (i) a new αXSS Q' (line 7) or (ii) a new αMFS Q^{**} (line 9). In the first case, a single query is executed (line 6), whereas in the second case, FindAnαMFS is called with query Q' as a starting point. Therefore $|Q'| + 1$ queries are executed to find an αMFS, with $[Q'| < |Q|$. In total, αLBA executes at most $|xss^{\alpha}(Q)| + |Q| * |mfs^{\alpha}(Q)|$ queries.

Corollary 1. *For n thresholds α_i, NLBA executes at most $\sum_{i=1}^{n} |xss^{\alpha_i}(Q)| + |Q| * |mfs^{\alpha_i}(Q)|$ queries.*

Bottom-Up In the Optimized-αLBA algorithm, query execution is performed only during instructions copied from αLBA (lines 5-13). Therefore, its complexity can be deduced directly from Theorem 1.

Corollary 2. *Optimized-αLBA (algorithm 4) executes at most $|xss^{\alpha}(Q) - dxss^{\alpha}(Q)| + |Q| * |mfs^{\alpha}(Q) - dmfs^{\alpha}(Q)|$ queries.*

To show the impact of the Bottom-Up approach, we first show that, in the worst case, this algorithm does not perform worse than NLBA.

Lemma 1. *DiscoverαMFSXSS (algorithm 3) executes at most $|dxss^{\alpha_i}(Q)| + |Q| * |dmfs^{\alpha_i}(Q)|$ queries.*

Proof. Atomic α_iMFS discovery (lines 1-3) requires no database access. After that, α_iMFS discovery is split into two parts: (i) based on previous α_{i-1}MFSs (lines 4-9) and (ii) based on failing α_{i-1}XSSs (lines 15-19). For the first part, $|\alpha_{i-1}MFS|$ queries are executed by FindAnαMFS (line 6). For the second part, $|\alpha_{i-1}XSS| + 1$ queries are executed to discover that the α_{i-1}XSS is failing (line 13) and then by FindAnαMFS (line 16). Note that $|\alpha_{i-1}MFS| < |Q|$ and $|\alpha_{i-1}XSS| + 1 \leq |Q|$. Each α_iXSS discovery (lines 13-14) requires a single query. In total, DiscoverαMFSXSS therefore executes at most $|dxss^{\alpha_i}(Q)| + |Q| * |dmfs^{\alpha_i}(Q)|$ queries.

Bottom-Up relies on the algorithm DiscoverαMFSXSS followed by Optimized-αLBA. Summing the results of Lemma 1 and Corollary 2 directly gives the following theorem.

Theorem 2. *For n thresholds α_i, Bottom-Up executes at most $\sum_{i=1}^{n} |xss^{\alpha_i}(Q)| + |Q| * |mfs^{\alpha_i}(Q)|$ queries.*

NLBA and Bottom-Up have the same worst case complexity. Using Bottom-Up becomes beneficial when some αMFSs and αXSSs can be inferred between successive thresholds α_i. In the best case, αMFSs and αXSSs remain the same for each threshold.

Lemma 2. *In the best case, where αMFSs and αXSSs remain the same for each threshold, for each threshold α_i after the first, Bottom-Up executes at most $\left(\sum_{Q^* \in mfs^{\alpha_i}(Q) \wedge |Q^*|>1} |Q^*| \right) + |xss^{\alpha_i}(Q)|$ queries.*

Proof. The discovery of all the α_iXSSs requires the execution of each one of them ($|xss^{\alpha_i}(Q)|$ queries). For the discovery of α_iMFSs that have a single triple pattern, no database access is required. For other α_iMFSs Q^*, FindAnαMFS is called with the execution of $|Q^*|$ queries, where $|Q^*|$ is the size of the considered α_{i-1}MFS, which is also in this best case an α_iMFS.

For n thresholds, the complexity of Bottom-Up is directly deduced from the previous lemma.

Proposition 5. *In the best case, for n thresholds α_i where $\forall\, i \in \{2, 3, ..., n\}$, $mfs^{\alpha_i}(Q) = mfs^{\alpha_1}(Q)$, Bottom-Up executes at most the following number of queries:*

$$n * |xss^{\alpha_1}(Q)| + |Q| * |mfs^{\alpha_1}(Q)| + (n-1) \left(\sum_{Q^* \in mfs^{\alpha_1}(Q) \wedge |Q^*|>1} |Q^*| \right) \quad (1)$$

Therefore, Bottom-Up becomes more efficient when αMFSs have a small number of predicates – to minimize the cost associated with the rightmost term of equation 1. Intuitively, since an α_{i-1}MFS is a superquery of an α_iMFS to be discovered, the search space becomes smaller when the α_{i-1}MFS has a small number of predicates.

Top-Down We now consider the complexity of the Top-Down approach. The two following lemmas are directly adapted from results taken from the Bottom-Up approach.

Lemma 3. *FindAnαXSS (algorithm 5) executes exactly $|Q| - |Q^*|$ queries.*

Lemma 4. *During the Top-Down approach, the dual algorithm of Discover-αMFSXSS executes at most $|dmfs^{\alpha_i}(Q)| + |Q| * |dxss^{\alpha_i}(Q)|$ queries.*

Top-Down relies on the dual algorithm of DiscoverαMFSXSS followed by Optimized-αLBA. Summing the results of Lemma 4 and Corollary 2 directly gives the following theorem.

Theorem 3. *For each threshold α_i, Top-Down executes at most* $(|Q|-1)\left(|dxss^{\alpha_i}(Q)|-|dmfs^{\alpha_i}(Q)|\right)+|xss^{\alpha_i}(Q)|+|Q|*|mfs^{\alpha_i}(Q)|$ *queries.*

Note that increasing the amount of αMFSs discovered during Top-Down ($|dmfs^{\alpha}(Q)|$) reduces this complexity. In particular, if all αMFSs and αXSSs are discovered ($dmfs^{\alpha}(Q) = mfs^{\alpha}(Q)$ and $dxss^{\alpha}(Q) = xss^{\alpha}(Q)$), the total complexity becomes $|Q| * |xss^{\alpha}(Q)| + |mfs^{\alpha}(Q)|$, which is the dual of Bottom-Up. On the opposite, if no αMFSs and αXSSs are discovered ($dmfs^{\alpha}(Q) = \emptyset$ and $dxss^{\alpha}(Q) = \emptyset$), the total complexity becomes $|xss^{\alpha}(Q)| + |Q| * |mfs^{\alpha}(Q)|$, i.e., the same as αLBA.

Lemma 5. *In the best case, where αMFSs and αXSSs remain the same for each threshold, for each threshold α_i after the first, Top-Down executes at most*

$$\left(\sum_{\substack{Q^* \in xss^{\alpha_1}(Q) \\ \wedge |Q^*|<|Q|-1}} |Q| - |Q^*|\right) + |mfs^{\alpha_i}(Q)| \text{ queries.}$$

Proof. The proof is the dual of lemma 2 for Bottom-Up.

For n thresholds, the complexity of Top-Down is directly deduced from the previous lemma.

Proposition 6. *In the best case, for n thresholds α_i where $\forall\, i \in \{2, 3, ..., n\}$, $xss^{\alpha_i}(Q) = xss^{\alpha_1}(Q)$, Top-Down executes at most the following number of queries:*

$$|xss^{\alpha_1}(Q)| + (|Q| + n - 1) * |mfs^{\alpha_1}(Q)| + (n-1)\left(\sum_{\substack{Q^* \in xss^{\alpha_1}(Q) \\ \wedge |Q^*|<|Q|-1}} |Q| - |Q^*|\right) \quad (2)$$

Therefore, Top-Down becomes more efficient when αXSSs have a large number of predicates – to minimize the cost associated with the rightmost term of equation 2. Intuitively, since an α_{i-1}XSS is a subquery of an α_iXSS to be discovered, the search space becomes smaller when the α_{i-1}XSS has a large number of predicates.

Hybrid We now consider the complexity of the Hybrid approach. Since MFSs and XSSs can be discovered from both lower and higher thresholds with distinctive number of executed queries, the following lemma identifies $dmfs^{\alpha}_{B}(Q)$ as MFSs discovered from a lower threshold in a manner similar to Bottom-Up. $dxss^{\alpha}_{B}(Q)$, $dmfs^{\alpha}_{T}(Q)$, $dxss^{\alpha}_{T}(Q)$ are defined similarly according to their type (MFSs, XSSs) and their origin (lower threshold: B, higher threshold: T). The following lemma is the sum of the results of Lemma 1 and Lemma 4.

Lemma 6. *The discovery process of MFSs and XSSs during the Hybrid approach executes at most* $|Q| * |dmfs^{\alpha_i}_{B}(Q)| + |dxss^{\alpha_i}_{B}(Q)| + |dmfs^{\alpha_i}_{T}(Q)| + |Q| * |dxss^{\alpha_i}_{T}(Q)|$.

Note that $dmfs_B^{\alpha_i}(Q) \cap dmfs_T^{\alpha_i}(Q) = \emptyset$ and $dxss_B^{\alpha_i}(Q) \cap dxss_T^{\alpha_i}(Q) = \emptyset$.

Summing the results of Lemma 6 and Corollary 2 directly gives the following theorem.

Theorem 4. *For each threshold α_i, Hybrid executes at most* $(|Q|-1)\left(|dxss_T^{\alpha_i}(Q)| - |dmfs_T^{\alpha_i}(Q)|\right) + |xss^{\alpha_i}(Q)| + |Q| * |mfs^{\alpha_i}(Q)|$ *queries.*

This worst-case complexity is the same as Top-Down (Theorem 3). Intuitively, the Bottom-Up part is covered by Optimized-LBA, as highlighted in theorem 2. As for Top-Down, this complexity becomes the same as αLBA when no MFSs or XSSs are discovered.

As for the best case, the complexity of Hybrid is the same as Bottom-Up (theorem 5) for its first two thresholds (the lowest threshold is followed by the highest during this evaluation). Once these mfs^{α_1}, xss^{α_1}, mfs^{α_n} and xss^{α_n} are discovered, the algorithm executes no further queries to determine the αMFSs and αXSSs of every other threshold (proposition 4).

Corollary 3. *In the best case, for n thresholds α_i where $\forall\, i \in \{2, 3, ..., n\}$, $mfs^{\alpha_i}(Q) = mfs^{\alpha_1}(Q)$, Hybrid executes at most the following number of queries:*

$$2 * |xss^{\alpha_1}(Q)| + |Q| * |mfs^{\alpha_1}(Q)| + \left(\sum_{Q^* \in\, mfs^{\alpha_1}(Q) \wedge |Q^*|>1} |Q^*| \right)$$

In the best case, Hybrid is the only approach whose complexity is independent from the number of thresholds, which can be especially useful if many have to be evaluated.

6 Experimental Evaluation

In this section, we investigate the scalability of our proposed approaches and compare them with the baseline method NLBA (executing αLBA for each of the N thresholds).

Algorithms We have implemented the Top-Down, Bottom-Up and Hybrid algorithms as well as the baseline method NLBA in Oracle Java 1.8 64 bits. These algorithms take as inputs a failing query and a set of thresholds. They return the sets of αMFSs and αXSSs of this query for each threshold. In our current implementation, these algorithms can be run on top of Jena TDB and Virtuoso. Our implementation is available at https://forge.lias-lab.fr/projects/qars4ukb with a tutorial to reproduce our experiments.

Experimental Setup Our experiments were conducted on a Ubuntu Server 16.04 LTS system with Intel XEON CPU E5-2630 v3 @2.4Ghz CPU and 16GB RAM. For our experiments, we use arbitrarily the *min* aggregate function. Presented results are the average of five consecutive runs of the algorithms. To prevent a cold start effect, a preliminary run is performed but not included in the results.

Dataset and Queries We used six datasets of 20K, 100K, 20M, 40M, 60M and 80M triples generated with the Waterloo SPARQL Diversity Test Suite (WatDiv) [17] (the 20K dataset is a subset of the 100K dataset, and so on). The certainty degrees of the RDF triples were generated randomly. As future work, we plan to perform an evaluation with less synthetic or different datasets.

We consider 7 queries of the WatDiv benchmark that we have modified to get failing queries (see Table 2). These queries range between 1 and 15 triple patterns and cover the main query patterns: star (characterized by *subject-subject* joins between triple patterns), chain (composed of *object-subject* joins) and composite (made of other join patterns). These characteristics have been chosen according to the results obtained in the study of Arias Gallego et al. [18] on real-world SPARQL queries executed on the DBPedia and SWDF datasets. Indeed, they have shown that these queries are based on the star, chain and composite query patterns and range from 1 to 15 triple patterns.

Quadstore implementation The storage and retrieval of RDF triples are usually done with a triplestore such as Jena TDB or Virtuoso. We have considered different implementations on top of these triplestores to manage uncertain RDF triples and threshold queries:

- *quad filter implementation.* This implementation is specific to Jena TDB. We use the *named graph technique* [19] (also called *N-quads technique*) to represent triples with theirs degrees of certainty (quads), and the specific Jena TDB low level quad filter hook[5] to retrieve results satisfying the provided threshold;
- *named graph implementation.* As in the previous implementation, we use the named graph technique to manage uncertain RDF triples but we retrieve results satisfying the provided threshold by querying the set of the named graphs (more details are given in Section 6.4);
- *reification implementation.* Instead of using the named graph technique, the other standard way to represent quads is to use reification of RDF triples. This technique enables the definition of an RDF triple in which the subject is another RDF triple. Thus, it can be used to define the degrees of certainty of each RDF triple. Threshold queries then have to be rewritten to accommodate for the reification (more details are given in Section 6.5).

While the quad filter implementation is specific to Jena TDB, the named graph and reification implementations can be set on any triplestore. However, we have observed that query execution times are significantly longer on the named graph and reification implementations in comparison with the quad filter implementation. Thus our experiments on large datasets (Sections 6.1, 6.2 and 6.3) are run on top of Jena TDB with the quad filter implementation. In Sections 6.4 and 6.5, we will see, on smaller datasets, that our approaches still outperform the baseline method with the named graph and reification implementations.

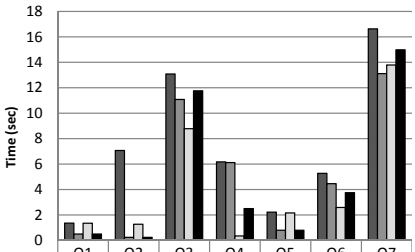
[5] http://jena.apache.org/documentation/tdb/quadfilter.html

Q1 (3TP)	`SELECT * WHERE { ?p friendOf ?f . ?f likes ?p . ?p type ProductCategory }`
Q2 (6TP)	`SELECT * WHERE { User666524 likes ?v0 . ?v0 hasGenre ?v1 . ?v1 tag Topic129 . ?v0 friendOf ?v2 . ?v2 Location ?v3 . ?v3 parentCountry Country17 }`
Q3 (7TP)	`SELECT * WHERE { ?v0 follows ?v1 . ?v1 follows ?v0 . ?v1 subscribes ?v2 . ?v0 subscribes ?v2 . ?v1 likes Product16770 . ?v0 nationality Country20 . ?v0 makesPurchase ?v3 }`
Q4 (8TP)	`SELECT * WHERE { ?v0 type User . ?v0 familyName 'Smith' . ?v0 subscribes Website36909 . ?v0 follows ?v1 . ?v0 friendOf ?v2 . ?v0 likes ?v3 . ?v0 userId ?v4 . ?v0 makesPurchase ?v5 . ?v0 Location ?v6 . ?v0 nationality ?v7 . ?v0 userId ?v8 }`
Q5 (10TP)	`SELECT * WHERE { ?p likes ?x . ?x likes ?p . ?p hasGenre SubGenre92 . ?x subscribe ?w1 . ?w1 language Language21 . Website121 hits ?h . ?x homepage Website120 . ?x familyName 'Smith' . ?x friendOf ?x2 . ?x2 email 'xxx@xxx.com' }`
Q6 (12TP)	`SELECT * WHERE { ?v0 eligibleRegion Country05 . ?v0 includes ?v1 . Retailer1257 offers ?v0 . ?v0 price '90' . ?v0 serialNumber ?v4 . ?v0 validFrom ?v5 . ?v0 validThrough ?v6 . ?v0 eligibleQuantity ?v8 . ?v0 priceValidUntil ?v11 . ?v1 tag ?v7 . ?v1 keywords ?v10 . ?v12 purchaseFor ?v1 }`
Q7 (15TP)	`SELECT * WHERE { ?v0 type ProductCategory7 . ?v0 tag Topic245 . ?v0 hasReview ?v4 . ?v0 contentSize ?v9 . ?v0 description ?v10 . ?v0 keywords ?v11 . ?v12 purchaseFor ?v0 . ?v2 tag ?v1 . ?v4 rating ?v5 . ?v4 reviewer ?v6 . ?v4 text ?v7 . ?v4 title ?v8 . ?v6 familyName ?v13 . ?v6 birthDate ?v14 . ?v0 gender ?v15 }`

Table 2: Failing queries considered for the experimentation (TP = Triple Patterns)

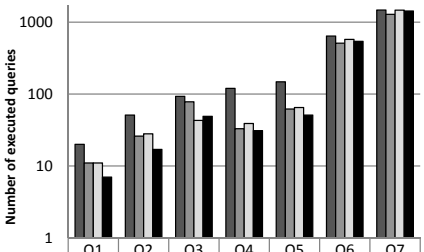
6.1 Algorithm Performance Comparison

Experiment description In this experiment, we have evaluated the performance of our algorithms Bottom-Up, Top-Down and Hybrid in comparison with the baseline method NLBA. This experiment has been run with the thresholds arbitrarily set to $\{0.2, 0.4, 0.6, 0.8\}$ on Jena TDB (quad filter implementation) with the 20M triples dataset.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
NLBA	1,36	7,07	13,08	6,17	2,22	5,27	16,63
Bottom-Up	0,5	0,24	11,08	6,12	0,8	4,46	13,11
Top-Down	1,35	1,27	8,78	0,35	2,16	2,59	13,79
Hybrid	0,5	0,24	11,76	2,5	0,8	3,75	14,98

Fig. 7: Execution time (20M triples, Jena TDB quad filter implementation)



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
NLBA	20	51	93	120	148	641	1475
Bottom-Up	11	26	78	33	62	510	1286
Top-Down	11	28	43	39	65	576	1470
Hybrid	7	17	49	31	51	541	1431

Fig. 8: # Executed queries (20M triples, Jena TDB quad filter implementation)

Results and discussion Figure 7 shows the execution time of each algorithm for each workload query. Figure 8 gives the number of executed queries by each algorithm. This experiment shows the improvement of our algorithms w.r.t the NLBA baseline method. In comparison with NLBA, our algorithms execute fewer queries to find the αMFSs and αXSSs of each workload query. Overall, Bottom-Up, Top-Down and Hybrid execute respectively 39%, 40% and 39% fewer queries than NLBA. As a consequence, these algorithms have shorter execution times (a decrease of respectively 30%, 42% and 33% execution times for Bottom-Up, Top-Down and Hybrid). For some queries, this improvement is important. For example, NLBA needs 7 seconds to find the αMFSs and αXSSs of Q2, whereas our algorithms need around 1 second. The difference of execution time depends heavily on the queries that our algorithms avoid executing. For example, our algorithms execute between 30 and 40 queries for Q4 whereas NLBA needs 120 queries. For Top-Down and Hybrid, this results in an important performance gain. This is not the case for Bottom-Up that has nearly the same execution time as NLBA. By analyzing the executed queries, we find that Bottom-Up only avoids executing queries that have short execution times and, then, still executes the most expensive queries. Thus, the overall execution time remains mostly unchanged.

This experiment also shows that no algorithm is better than the others for all queries. Bottom-Up and Hybrid have the best execution times for Q1, Q2 and Q5 whereas Top-Down is the best for Q3, Q4 and Q6. Despite executing the least number of queries, Bottom-Up has the worst execution time for this workload. Conversely, Top-Down executes the greatest number of queries but has the best execution time. This is due to the fact that our algorithms execute different queries that have distinct execution times. In particular, Top-Down starts by searching the αMFSs and αXSSs for the highest thresholds. The executed queries tend to be selective as the threshold is high and thus, have short execution times. Once the αMFSs and αXSSs for the highest thresholds are found, they avoid the execution of queries with a lower threshold that are likely to be more expensive. As Bottom-Up follows the dual approach, it tends to execute non-selective queries and has the overall worst performance. In this experiment, Hybrid never performs better than both Top-Down and Bottom-Up. Since the certainty degrees were randomly generated and the four considered thresholds are separated by a significant margin, queries share few αMFSs and αXSSs between the different thresholds. Thus, the specific property used by Hybrid (see Section 4.3) is rarely exploited.

6.2 Algorithm Performance w.r.t the Dataset Size

Experiment description The second experiment consists in evaluating the scalability of the algorithms when the size of the dataset increases. This experiment has been run with the same settings as the previous one (thresholds set to $\{0.2, 0.4, 0.6, 0.8\}$ on Jena TDB quad filter implementation) but with the 20M, 40M, 60M and 80M datasets.

Results and discussion Figure 9 and Table 3 present the execution time of the algorithms for Q2 with the 20M, 40M, 60M and 80M datasets. The execution times of our algorithms do not increase significantly between the 40M and 80M datasets. On these datasets, we have observed that the αMFSs and αXSSs of Q2 remain the same and thus the same queries are executed (around 25 queries for our algorithms and 46 for NLBA). As a consequence, in this case, the scalability of the algorithms depends only on the execution times of these queries.

For the 20M dataset, Q2 has an additional αMFS for all the thresholds. The number of αXSS are the same but they are shorter. As a consequence, the algorithms execute different queries. This has a direct impact on the performance of the algorithms. In particular, the execution time of Top-Down is around 1 second on the 20M dataset (5 seconds on other datasets) despite the fact that it executes more queries (28 queries on 20M and 24 queries on other datasets). Thus, when the αMFSs and αXSSs change, the scalability of the algorithms also depends on the executed queries and their respective response times.

	NLBA	Bottom-Up	Top-Down	Hybrid
20M	7.04	0.24	1.26	0.24
40M	6.86	1.59	4.62	1.57
60M	8.2	1.66	4.94	1.64
80M	9.58	1.72	5.29	1.71

Table 3: Execution time vs Dataset size

6.3 Algorithm Scalability w.r.t the Number of Thresholds

Experiment description In this experiment, we evaluate the scalability of the algorithms when the number of thresholds increases. This experiment was conducted with Q6 on the Jena TDB quad filter implementation and the 20M dataset. We have executed the algorithms for one threshold $\{0.1\}$, two thresholds $\{0.1, 0.2\}$, three thresholds $\{0.1, 0.2, 0.3\}$, and so on up to nine thresholds.

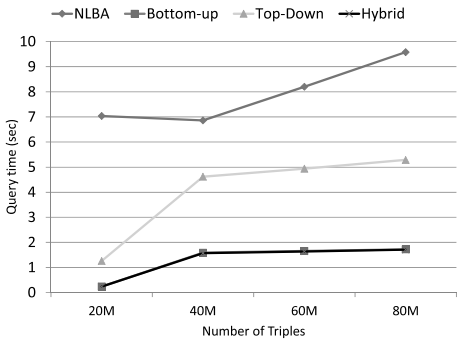


Fig. 9: Execution time vs Dataset size (thresholds $\{0.2, 0.4, 0.6, 0.8\}$, Jena TDB quad filter implementation)

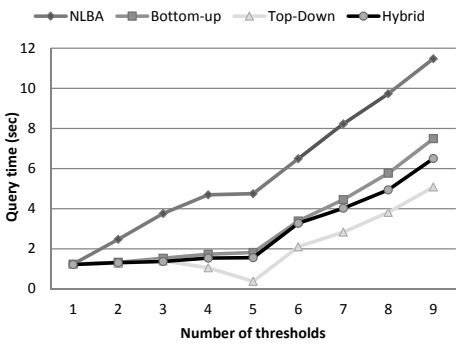


Fig. 10: Execution time vs Number of α (20M triples, Jena TDB quad filter implementation)

Results and discussion The result of this experiment is depicted in Figure 10. As shown by this experiment, our algorithms always outperform NLBA once two or more thresholds are considered. This is due to the fact that, for each new threshold, NLBA executes the original version of the αLBA algorithm while our algorithms execute an optimized version of αLBA thanks to the already discovered αMFSs and αXSSs. As a consequence, NLBA scales nearly linearly with the number of thresholds. In comparison, the scalability of our algorithms depends on the number of discovered αMFSs and αXSSs. If the αMFSs and αXSSs are rather the same between different thresholds, our algorithms only require milliseconds to find the αMFSs and αXSSs for a new threshold in this interval. This is the case in our experiments for the thresholds between 0.1 and 0.5. On the contrary, if the αMFSs and αXSSs change between different

thresholds (this is the case between 0.5 and 1), our algorithms scale almost like NLBA as the optimized version αLBA leverages few discovered αMFSs and αXSSs.

We can also observe in this experiment that the Top-Down algorithm has a better execution time with 5 thresholds than with only one threshold, which may be surprising. This behavior is explained as follows. The αLBA algorithm has a short execution time for 0.5 (as shown by the results of NLBA on Figure 10). Once the αMFSs and αXSSs are found for 0.5, as they are rather the same between 0.1 and 0.5, Top-Down only needs a few milliseconds for the other thresholds. In comparison, the execution of αLBA for 0.1 takes a longer time (as the executed queries are less selective than with 0.5).

6.4 Algorithm Performance on the Named Graph Implementation

The previous experiments were all run on the quad filter implementation only available in Jena TDB. As other triplestores, such as Virtuoso, may be used for storing and querying uncertain KBs, we have considered other implementations of quads. Our objective was twofold: on the one hand, to show that our approaches can be used on several triplestores and, on the other hand, to check whether our approaches still outperform the baseline method on these generic implementations. In this section, we consider the named graph implementation.

Experiment description In the named graph implementation, each RDF statement is represented as a quad *(subject, predicate, object, graphname)*. The named graph is used to represent the degree of certainty for each RDF triple. Then, threshold queries are rewritten to take into consideration the named graphs. For example, in a simplified way, the query Q_2 introduced in Section 2 is rewritten as Q'_2 (for $\alpha = 0.8$):

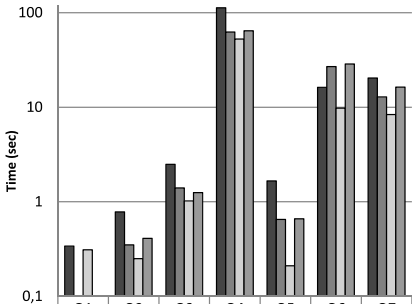
Q'_2:
```
SELECT ?b ?p WHERE {
      GRAPH ?g {
        ?b author "Abraham Lincoln".
        ?b editor "Springer" .
        ?b type Book .
        ?b nbPages ?p }
      FILTER (?g > 0.8) }
```

This experiment consists in evaluating the performance of our algorithms Bottom-Up, Top-Down and Hybrid in comparison with the baseline method NLBA on the named graph implementation done on both Jena TDB and Virtuoso. As stated previously, this implementation is largely slower than the quad filter implementation. Thus, we use a smaller dataset of of 100K quads, to account for the reduced performance. As in the previous experiments, we used arbitrarily the thresholds $\{0.2, 0.4, 0.6, 0.8\}$.

Results and discussion Figures 11 and 12 show the execution time of each algorithm for each query on, respectively, Jena TDB and Virtuoso. Figure 13

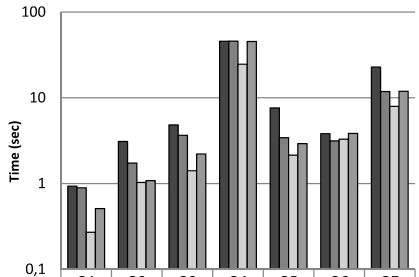
gives the number of executed queries by each algorithm. Note that the number of executed queries depends only on the algorithm, the dataset and the query. The chosen implementation – the underlying triple store and the representation of the trust value (named graph, reification) – has no impact on this result.

This experiment shows that our approaches still outperform the NLBA baseline method on the named graph implementation. In comparison with NLBA, our algorithms execute fewer queries for finding the αMFSs and αXSSs of each workload query. Overall, Bottom-Up, Top-Down and Hybrid execute respectively 32%, 25% and 28% fewer queries than NLBA. As a consequence, these algorithms have shorter execution times on both Jena TDB (a decrease of respectively 32%, 53% and 27% execution times for Bottom-Up, Top-Down and Hybrid) and Virtuoso (a decrease of respectively 20%, 54% and 23% execution times for Bottom-Up, Top-Down and Hybrid).



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
NLBA	0,34	0,78	2,49	112,67	1,66	16,24	20,36
Bottom-Up	0,1	0,35	1,4	62,4	0,65	26,91	12,85
Top-Down	0,31	0,25	1,02	52,57	0,21	9,76	8,37
Hybrid	0,1	0,41	1,25	64,26	0,66	28,64	16,31

Fig. 11: Execution time (100K triples, Jena TDB named graph implementation)



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
NLBA	0,93	3,09	4,82	45,57	7,6	3,81	22,78
Bottom-Up	0,89	1,73	3,65	45,72	3,42	3,14	11,83
Top-Down	0,27	1,03	1,41	24,65	2,14	3,29	7,94
Hybrid	0,51	1,08	2,21	45,3	2,92	3,84	11,9

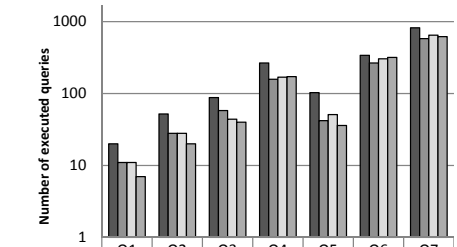
Fig. 12: Execution time (100K triples, Virtuoso named graph implementation)

6.5 Algorithm Performance on the Reification Implementation

Another standard way to represent quads in a triplestore consists in using reification. The advantage of this technique is that it is a standard of W3C[6] and it can be used on any triplestore [20]. This technique is used in several projects [2, 21–23] to represent and query added properties for RDF triples such as provenance, trust, certainty, time, and location.

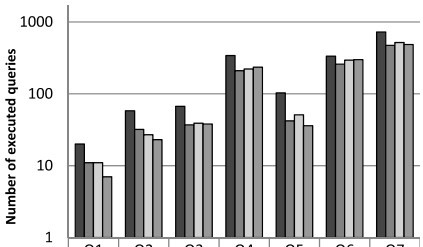
Experiment description We first briefly describe the principle of the reification implementation. Let us consider the triple $t = (b1, type, book)$. To associate

[6] https://www.w3.org/wiki/PropertyReificationVocabulary



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
NLBA	20	52	88	266	103	340	820
Bottom-Up	11	28	58	158	42	266	581
Top-Down	11	28	44	169	51	304	648
Hybrid	7	20	40	172	36	318	619

Fig. 13: # Executed queries (100K triples, Jena TDB and Virtuoso, named graph implementation)



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
NLBA	20	58	67	341	103	334	726
Bottom-Up	11	32	37	210	42	259	473
Top-Down	11	27	39	222	51	294	518
Hybrid	7	23	38	235	36	299	486

Fig. 14: # Executed queries (20K triples, Jena TDB and Virtuoso, reification implementation)

a certainty degree 0.4 to this RDF triple, we describe this statement (reification) with the following triples:

```
t rdf:type rdf:statement .
t rdf:subject b1 .
t rdf:predicate type .
t rdf:object book .
t trust "0.4"^^xsd:float
```

When querying the reifed dataset to find results with their trust degree, threshold queries are rewritten as follows:

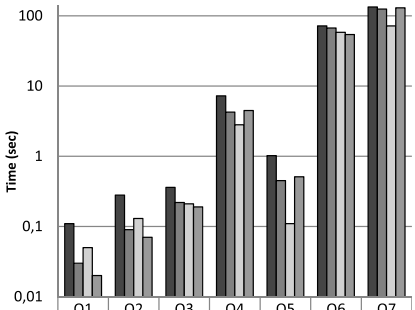
```
SELECT ?book WHERE {
    ?book type book }
α=0.4
```

⇒

```
SELECT ?book WHERE {
  ?t rdf:subject ?book .
  ?t rdf:predicate type .
  ?t rdf:object book .
  ?t rdf:trust ?trust
  FILTER (?trust > 0.4) }
```

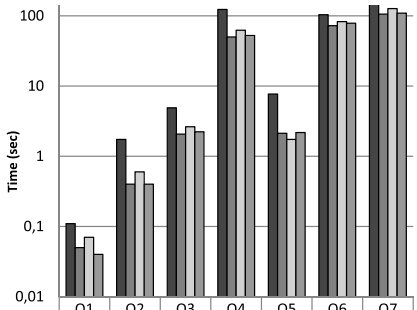
Compared with the named graph implementation, the reification method multiplies the size of the dataset by five (each triple is replaced by its reification and a triple to define the certainty degree), and the number of triple patterns in the query by four. As a consequence, we consider a dataset of 20K triples for this experiment to keep execution times reasonable. This experiment consists in evaluating the performance of our algorithms in comparison with the baseline method using the reification implementation on both Jena TDB and Virtuoso. As in previous experiments, we used the thresholds $\{0.2, 0.4, 0.6, 0.8\}$.

Results and discussion Figures 15 and 16 show the execution time of each algorithm for each query on, respectively, Jena TDB and Virtuoso. Figure 14 gives the number of executed queries by each algorithm.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
NLBA	0,11	0,28	0,36	7,25	1,02	71,86	133,84
Bottom-Up	0,03	0,09	0,22	4,26	0,45	67,09	124,61
Top-Down	0,05	0,13	0,21	2,81	0,11	58,29	71,97
Hybrid	0,02	0,07	0,19	4,49	0,51	54,19	130,21

Fig. 15: Execution time (20K triples, Jena TDB triple store, reification)



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
NLBA	0,11	1,74	4,9	123,27	7,68	103,83	182,37
Bottom-Up	0,05	0,4	2,07	50,07	2,13	72,43	105,69
Top-Down	0,07	0,6	2,63	62,39	1,74	82,65	127,11
Hybrid	0,04	0,4	2,23	52,58	2,18	78,53	109,13

Fig. 16: Execution time (20K triples, Virtuoso triple store, reification)

As for the named graph implementation, our algorithms outperform the NLBA baseline method. They execute fewer queries for finding the αMFSs and αXSSs of each workload query. Overall, Bottom-Up, Top-Down and Hybrid execute respectively 37%, 31% and 33% fewer queries than NLBA. As a consequence, these algorithms have shorter execution times on both Jena TDB (a decrease of respectively 11%, 37% and 8% execution times for Bottom-Up, Top-Down and Hybrid) and Virtuoso (a decrease of respectively 42%, 34% and 45% execution times for Bottom-Up, Top-Down and Hybrid).

By design, our three algorithms always execute less queries than the baseline method to compute αMFSs and αXSSs. Our experiments have shown that, regardless of the selected implementation, this always result in a significant performance improvement, with total execution times reduced by 8% to 54%.

7 Related Work

In this section, we provide a comprehensive review of the existing approaches to address the empty-answer problem in the context of KBs. First, a comparison between the main approaches is made. Then, a critical analysis of the closest approaches to our proposal is presented.

7.1 A Comparative Study

In the context of KBs, the empty-answer problem has been tackled by several approaches such as completing the KB using logical rules [24], checking the data during query formulation to avoid empty answers [25], deriving an emergent relational schema from the KB data to help users formulating queries [26] or relaxing the query to return alternative answers [27–33].

These approaches are compared in Table 4 on the basis of the following criteria:

- *target:* some approaches focus on improving the KB (completing it or extracting a relational schema from it) to avoid the empty answer problem, while other approaches focus on the user queries that raised this problem;
- *before/after query formulation:* this criteria specifies whether the considered approach anticipates the empty answer problem, i.e. try to solve it before query formulation or acts once this problem appears (during or after query formulation);
- *user involvement:* the approaches can ask more or less inputs from the user;
- *help to understand the KB:* the approaches may or may not help the user to better understand the content and/or the structure of the KB.

As we can see, two main categories of approaches appear.

- Completing the KB and deriving an emergent relational schema from it consider that the empty anwswer problem comes from the KB because it is incomplete and its structure is hidden to the user. Thus, these approaches anticipate the empty-answer problem by modifying the KB.
- Checking the data during query formulation and query relaxation focus on the user-provided query. The first approach helps the user formulate her/his query to prevent the empty answer problem. Its downside is that it requires many inputs from the user. Conversely, query relaxation takes the failing query as a starting point and provides the user with alternative answers. This process can be completely automatic [34] or guided by the user [33].

All these approaches can be complementary to tackle the empty answer problem. In the next section, we detail the closest approaches to our work.

Approach	**Target**	**Before/after query formulation**	**User involvement**	**Help to understand the KB**
Completing the KB	KB	Before	None	No
Interactive query formulation	User query	During	Strong	Yes
Emergent relational schema	KB	Before	None	Yes
Query relaxation	User query	After	More or less	Yes

Table 4: Comparison of different approaches to solve the empty answer problem

7.2 Relaxation-Driven Approaches

As our work is in the field of query relaxation for KBs, we summarise, in this section, the main contributions made in this domain. Several approaches proposed

relaxation operators in the RDF context. These operators are mainly based on RDFS semantics (e.g., generalizing triple patterns using class and property hierarchies) [27–30], similarity measures [31, 32] and user preferences [33]. These operators generate a set of relaxed queries, ordered by similarity with the original query and executed in this order [27, 28, 35]. Relaxation operators can be directly used by the user in her/his query [29, 30] or used in conjunction with query rewriting rules to perform relaxation [33]. In these approaches, the failure causes of the query are unknown, which may lead to executing unnecessary relaxed queries. Fokou et al. [12, 34] tackled this problem by firstly defining the LBA and MBA approaches to compute the MFSs and XSSs of the query [12] and secondly by proposing relaxation strategies based on MFSs that identify relaxed queries that necessarily fail [34]. Our approach is based on the LBA algorithm proposed in this work. We have extended this work by identifying the condition under which this algorithm can be used in the context of uncertain KBs where the query is associated with a threshold α and by defining several algorithms to compute αMFSs and αXSSs for several thresholds. Our work is among the pioneering works aiming at exploring the query relaxation issue in uncertain KBs. To the best of our knowledge, the only other work in this context is [35]. However, this work only uses the trust value to order results by their trustworthiness. They do not consider, as we do in this paper, queries that return no result satisfying the provided degree of trustworthiness.

The issue of computing MFSs and XSSs has also been tackled in the context of relational database [11], recommender systems [36] and fuzzy querying [37]. The closer work to ours is the one of Pivert and Smits [37] in the context of fuzzy querying. They have proposed an approach to compute gradual MFSs, i.e., MFSs that are only poorly satisfied as they do not return any answer with at least a satisfaction degree equal to a user-defined threshold as well as gradual XSSs. This approach is based on a summary of the relevant part of the database. It computes the gradual MFSs and XSSs for different thresholds as in our approach. However, while this work proposes an approach to compute MFSs and XSSs in the context of fuzzy querying on certain databases, our work targets classical queries on uncertain KBs. In this new context, a summary of the relevant part of the KBs cannot be efficiently computed [12].

8 Conclusion

In this paper, we have considered the empty answer problem in uncertain KBs. In this context, a query fails if it returns no result or results that do not satisfy an expected degree of certainty α. To provide the user with a relevant feedback, we have proposed to compute the αMFSs and αXSSs of the failing query as they give a clear overview of the query failure causes and a set of relaxed queries that she/he can execute to find some useful alternative answers. We have first defined the condition under which a previous work algorithm called αLBA can be directly adapted to the context of uncertain KBs. In this case, the user has to define her/his expected degree of certainty. However, the user may want to know

what happens if she/he relaxes the expected certainty. Thus, we have studied the problem of computing the αMFSs and αXSSs for multiple thresholds. The baseline method called NLBA consists in executing αLBA for each threshold. However, we have observed and proved that the αMFSs and αXSSs for a given threshold can be used to find others with lower or greater threshold. Thus, we have defined three alternative approaches to NLBA called Bottom-Up, Top-Down and Hybrid that consider α thresholds in different orders. We have done a complete implementation of these algorithms and shown experimentally on different datasets of the WatDiv benchmark that our approaches outperform the baseline method.

In our experiments, we have observed that none of our algorithms provide the best performance for all queries. As future work, we plan to study the conditions under which an algorithm may provide the best results. Our idea is to use the KB statistics and the cost model of the quadstore to find, on a case by case basis, the algorithm that is the most likely to have the best performance. A thorough analysis of the algorithms execution shows that many queries share some triple patterns. Thus, another perspective is to use multiple-query optimization and indexing techniques to further improve their execution times.

References

1. Rodríguez, M., Goldberg, S., Wang, D.Z.: Sigmakb: multiple probabilistic knowledge base fusion. Proceedings of the VLDB Endowment **9**(13) (2016) 1577–1580
2. 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
3. Carlson, A., Betteridge, J., Kisiel, B., Settles, B., Hruschka Jr, E.R., Mitchell, T.M.: Toward an architecture for never-ending language learning. In: AAAI. Volume 5. (2010) 3
4. 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: KDD'14. (2014) 601–610
5. Wu, W., Li, H., Wang, H., Zhu, K.Q.: Probase: A probabilistic taxonomy for text understanding. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM (2012) 481–492
6. Steve Harris, Garlik, A.S.: Sparql 1.1 query language (march 2013). W3C Recommendation (2013)
7. Hartig, O.: Querying Trust in RDF Data with tSPARQL. In: ESWC 2009. (2009)
8. Tomaszuk, D., Pak, K., Rybiński, H.: Trust in RDF graphs. In: ADBIS'13. (2013)
9. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.N.: LSQ: The Linked SPARQL Queries Dataset. In: ISWC'15. (2015) 261–269
10. Mottin, D., Marascu, A., Roy, S.B., Das, G., Palpanas, T., Velegrakis, Y.: A probabilistic optimization framework for the empty-answer problem. Proc. VLDB Endow. **6**(14) (September 2013) 1762–1773
11. Godfrey, P.: Minimization in Cooperative Response to Failing Database Queries. International Journal of Cooperative Information Systems **6**(2) (1997) 95–149
12. Fokou, G., Jean, S., Hadjali, A., Baron, M.: Handling Failing RDF Queries: From Diagnosis to Relaxation. Knowledge and Information Systems (KAIS) **50**(1) (2017)

13. Erling, O., Mikhailov, I.: Rdf support in the virtuoso dbms. In Pellegrini, T., Auer, S., Tochtermann, K., Schaffert, S., eds.: Networked Knowledge - Networked Media: Integrating Knowledge Management, New Media Technologies and Semantic Systems. Springer Berlin Heidelberg, Berlin, Heidelberg (2009) 7–24
14. Dellal, I., Jean, S., Hadjali, A., Chardin, B., Baron, M.: On addressing the empty answer problem in uncertain knowledge bases. In Benslimane, D., Damiani, E., Grosky, W.I., Hameurlain, A., Sheth, A., Wagner, R.R., eds.: Database and Expert Systems Applications, Springer International Publishing (2017) 120–129
15. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transaction on Database Systems (TODS) **34**(3) (2009) 16:1–16:45
16. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledge discovery. Data Mining and Knowledge Discovery **1**(3) (Sep 1997) 241–258
17. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified Stress Testing of RDF Data Management Systems. In: ISWC'14. (2014) 197–212
18. 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)
19. Gavin Carothers, Lex Machina, I.: Rdf 1.1 n-quads. W3C Recommendation (2014)
20. Guus Schreiber, VU University Amsterdam, Y.R.B.: Rdf 1.1 primer. In: W3C recommendation. (2014)
21. Sahoo, S.S., Nguyen, V., Bodenreider, O., Parikh, P., Minning, T., Sheth, A.P.: A unified framework for managing provenance information in translational research. In: BMC bioinformatics. Volume 12. (2011) 461
22. Schueler, B., Sizov, S., Staab, S., Tran, D.T.: Querying for meta knowledge. In: Proceedings of the 17th international conference on World Wide Web, ACM (2008) 625–634
23. Straccia, U., Lopes, N., Lukacsy, G., Polleres, A.: A general framework for representing and reasoning with annotated semantic web data. In: AAAI. (2010)
24. Galárraga, L., Teflioudi, C., Hose, K., Suchanek, F.M.: Fast Rule Mining in Ontological Knowledge Bases with AMIE+. VLDB Journal **24**(6) (2015) 707–730
25. Campinas, S.: Live SPARQL Auto-Completion. In: ISWC'14 (Posters & Demos). (2014) 477–480
26. Pham, M., Passing, L., Erling, O., Boncz, P.A.: Deriving an Emergent Relational Schema from RDF Data. In: WWW'15. (2015) 864–874
27. Hurtado, C.A., Poulovassilis, A., Wood, P.T.: Ranking Approximate Answers to Semantic Web Queries. In: ESWC'09. (2009) 263–277
28. 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
29. Fokou, G., Jean, S., Hadjali, A.: Endowing Semantic Query Languages with Advanced Relaxation Capabilities. In: ISMIS'14. (2014) 512–517
30. Calí, A., Frosini, R., Poulovassilis, A., Wood, P.: Flexible Querying for SPARQL. In: ODBASE'14. (2014) 473–490
31. Hogan, A., Mellotte, M., Powell, G., Stampouli, D.: Towards Fuzzy Query-relaxation for RDF. In: ESWC'12. (2012) 687–702
32. Elbassuoni, S., Ramanath, M., Weikum, G.: Query Relaxation for Entity-Relationship Search. In: ESWC'11. (2011) 62–76
33. Dolog, P., Stuckenschmidt, H., Wache, H., Diederich, J.: Relaxing RDF queries based on user and domain preferences. Journal of Intelligent Information Systems (JIIS) **33**(3) (2009) 239–260

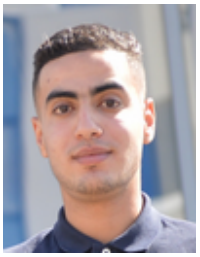
34. Fokou, G., Jean, S., HadjAli, A., Baron, M.: RDF Query Relaxation Strategies Based on Failure Causes. In: ESWC'16. (2016) 439–454

35. Reddy, K.B., Kumar, P.S.: Efficient Trust-Based Approximate SPARQL Querying of the Web of Linked Data. In: Uncertainty Reasoning for the Semantic Web II. Springer (2013) 315–330

36. Jannach, D.: Fast Computation of Query Relaxations for Knowledge-based Recommenders. AI Communications **22**(4) (2009) 235–248

37. 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. Springer (2015) 499–517

Authors

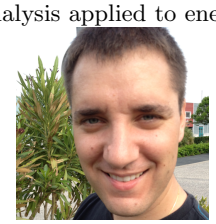


Ibrahim Dellal is a PhD student and member of the data engineering team of Data and Model Engi- neering team of the Laboratory of Computer Science and Automatic Control for Systems (LIAS) at the National Engineering School for Mechanics and Aerotech- nics (ISAE-ENSMA), France. He got a Bachelor's and a Master's degrees in Computer Science at the University of Poitiers, France. He spent his Master degree internship at LIAS where he started working on "Management and Exploitation of Large and Uncertain Knowledge Bases". His field of research includes Semantic Web, RDF data, Uncertainty, Knowledge bases, Cooperative research techniques. Web page is available on https://www.lias-lab.fr/members/ibrahimdellal

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

Brice Chardin is an associate professor at ISAE-ENSMA and a member of the Data and Model Engineering team at LIAS since 2013. He received his PhD in computer science from INSA Lyon in 2011, where he developed a distributed data management system on flash memories to handle sensor data produced by power plants. He then held a postdoctoral research position at LIRIS on pattern mining in databases. His current research interests include flash-aware DBMSs and predictive analysis applied to energy forecasting.



Mickael 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