On Addressing the Empty Answer Problem in Uncertain Knowledge Bases

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

On Addressing the Empty Answer Problem in Uncertain Knowledge Bases

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

Abstract. Recently, several large *Knowledge Bases* (KBs) have been constructed by mining the Web for information. As an increasing amount of inconsistent and non-reliable data are available, KBs facts may be *uncertain* and are then associated with an explicit 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 KBs contents, their queries may be failing i.e., they return no result for the desired certainty. 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 two algorithms to compute the αMFSs and αXSSs for other thresholds. Our experiments on the WatDiv benchmark show the relevance of our algorithms compared to a baseline method.

1 Introduction

A *Knowledge Base* (KB) is a collection of entities and facts about them. Well-known examples of KBs include Knowledge Vault [1] and YAGO [2]. These KBs contain billions of facts captured as RDF triples *(subject, predicate, object)* and are queried with the SPARQL language. As these KBs have been constructed by mining the Web for information, their facts are *uncertain* (i.e., potentially inconsistent). Therefore, an explicit degree of certainty is assigned to KB facts. 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 contents of a KB, they may be faced with the 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. Indeed, the study conducted by Saleem et al. on SPARQL endpoints shows that ten percent of queries submitted to DBpedia between May and July 2010 returned empty results [3]. 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). Moreover, interesting non-failing relaxed queries, called *MaXimal Succeeding Subqueries* (XSSs), might be suggested to the user by the system as well.

The problem of computing MFSs and XSSs of SPARQL queries expressed on KBs has already been addressed by Fokou et al. [4]. 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 relaxed subqueries of a query that filters results according to their certainty degree and a given threshold α. We first show under which conditions the computation of MFSs and XSSs can be directly adapted to αMFSs and αXSSs. In this setting, the user has to define the threshold α. However, as she/he may not have an idea of the certainty degrees assigned to the KB RDF triples, 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 multiple thresholds α. To save computation time, some properties between αMFSs and αXSSs of different thresholds are established and exploited. Thus, depending on which order the α values are considered, two approaches *Top-Down* and *Bottom-Up* are discussed. We run the experiments on the WatDiv benchmark with the Jena TDB quadstore to show the impact of our approaches.

The paper is structured as follows. Section 2 formalizes the problem. Section 3 defines the conditions under which a previous work algorithm can be directly adapted to find the αMFSs and αXSSs for a given α. Section 4 describes the two proposed approaches to compute αMFSs and αXSSs for a set of thresholds. Section 5 discusses the experimental evaluation performed. Section 6 details related work and Section 7 concludes.

2 Problem statement

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

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 $var(Q) = \bigcup var(t_i)$.

A *mapping* μ from V to T is a partial function $\mu : V \rightarrow T$. For a triple pattern t, we denote by $\mu(t)$ the triple obtained by replacing in t its variables $var(t)$ by their mapping $\mu(var(t))$. Let D be an *RDF* database, t a triple pattern. The evaluation of the triple pattern t over D denoted by $[[t]]_D$ is defined by: $[[t]]_D = \{\mu \mid dom(\mu) = var(t) \wedge \mu(t) \in D\}$. Let Q be a query, the evaluation of Q

over D is defined by: $[[Q]]_D = [[t_1]]_D \bowtie \cdots \bowtie [[t_n]]_D$. 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 $tv(\mu, Q) = aggreg(tv(\mu(t_1)), \cdots, tv(\mu(t_n)))$. The evaluation of Q over D that returns trust weighted results with a threshold α is defined by: $[[Q]]_D^\alpha = \{\mu \in [[Q]]_D \mid tv(\mu) \geq \alpha\}$.

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 ($Q' \subset Q$). An $\alpha Minimal\ Failing\ Subquery$ (αMFS) Q^* of a query Q for a given α is defined by: $[[Q^*]]_D^\alpha = \emptyset \wedge \nexists\, Q' \subset Q^*$ *such that* $[[Q']]_D^\alpha = \emptyset$. The set of all αMFSs of a query Q for a given α is denoted by $mfs^\alpha(Q)$. An $\alpha Maximal\ Succeeding\ Subquery$ (αXSS) Q^* of a query Q for a given α is defined by: $[[Q^*]]_D^\alpha \neq \emptyset \ \wedge \nexists\, Q'$ *such that* $Q^* \subset Q' \wedge [[Q']]_D^\alpha \neq \emptyset$. The set of all αXSSs of a query Q for a given α is denoted by $xss^\alpha(Q)$.

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

3 αMFSs and αXSSs computation for a single α

In this section, we first give a direct adaptation of the *Lattice-Based Approach* (LBA) proposed in [4] to compute the αMFSs and αXSSs of a query for a given α. It has the same algorithmic complexity as LBA (detailed in [4]). αLBA explores the lattice of subqueries by following a three-steps procedure.

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 [4].

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

```
FindAnαMFS(Q, D, α)
    inputs : A failing query Q = t_1 ∧ ... ∧ t_n; an RDF database D;
             a threshold α
    output: An αMFS of Q denoted by Q*
1   Q* ← ∅; Q' ← Q;
2   foreach triple pattern t_i ∈ Q do
3       Q' ← Q' − t_i;
4       if [[Q' ∧ Q*]]_D^α ≠ ∅ then
5           Q* ← Q* ∧ t_i;
6   return Q*;
```

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

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

3. Test potential αXSSs. If a subquery found during step 2 succeeds, it is then an αXSS. If it fails, we apply the two previous steps on this particular subquery to find a new αMFS and its associated potential αXSSs. This is illustrated by Algorithm 2. It is worth noting that this algorithm includes mechanisms to avoid discovering the same αMFSs multiple times (lines 11-13).

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

```
αLBA(Q, D, α)
    inputs : A failing query Q = t_1 ∧ ... ∧ t_n; an RDF database D;
             a threshold α
    outputs: The αMFSs and αXSSs of Q
1   Q* ←FindAnαMFS(Q, D, α);
2   pxss ← pxss(Q, Q*);
3   mfs^α(Q) ← {Q*}; xss^α(Q) ← ∅;
4   while pxss ≠ ∅ do
5       Q' ← pxss.element(); // choose an element of pxss
6       if [[Q']]^α_D ≠ ∅ then // Q' is an αXSS
7           xss^α(Q) ← xss^α(Q) ∪ {Q'}; pxss ← pxss − {Q'};
8       else // Q' contains an αMFS
9           Q** ←FindAnαMFS(Q', D, α);
10          mfs^α(Q) ← mfs^α(Q) ∪ {Q**};
11          foreach Q'' ∈ pxss such that Q** ⊆ Q'' do
12              pxss ← pxss − {Q''};
13              pxss ← pxss ∪ {Q_j ∈ pxss(Q'', Q**) | ∄Q_k ∈
                  pxss ∪ xss^α(Q) such that Q_j ⊆ Q_k};
14  return {mfs^α(Q), xss^α(Q)};
```

The αLBA algorithm relies on the fact that a successful query cannot contain a failing query. In the context of uncertain KBs, this property does not always hold depending on the chosen trust value aggregate function (*aggreg*). For example, with the maximum aggregate function, the degree of certainty of results potentially increases with additional triple patterns. Thus, a query may be failing but not its subqueries. The algorithm αLBA can only be used if the aggregate function *aggreg* is monotonic decreasing with respect to the subset partial order. We omit the proof of this property.

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

As examples of monotonic decreasing aggregate functions, we can cite the *minimum* or the *product* restricted to values $\in [0,1]$.

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

Proposition 1. *Let aggreg be monotonic decreasing.* $[[Q]]^{\alpha}_{D} = \emptyset \wedge Q' \subset Q \Rightarrow [[Q']]^{\alpha}_{D} = \emptyset$. *That is to say, if a proper subquery* Q' *of* Q *fails for a given* α *(using the aggreg function) then* Q *also fails for* α.

4 αMFSs and αXSSs computation for a set of α

To find αMFSs and αXSSs for a set of α: $\{\alpha_1, \cdots \alpha_n\}$, the αLBA algorithm can be applied for each α_i. This baseline method is named *NLBA*. In this section, we discuss various improvements of this approach. The idea is that the αMFSs and αXSSs for a given threshold provide a set of hints to deduce some αMFSs and αXSSs with higher (or lower) thresholds.

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_iMFS 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. Due to space constraints, proofs are omitted.

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

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, this case is checked first and all discovered α_jMFS of Q are put in a set of *discovered* α*MFSs* 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 FindAnαMFS over executing the subqueries of the α_iMFS to discover new α_jMFSs, as shown in Algorithm 3.

As for 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_i*XSS of* Q *on a dataset* D. *If* $[[Q^*]]^{\alpha_j}_{D} \neq \emptyset$, *then* Q^* *is an* α_j*XSS of* Q.

Thus, discovering if an α_iXSS is also an α_jXSS only requires the execution of a single query (α_iXSS with the new threshold α_j). This enables us to find a set of discovered α_jXSSs, denoted $dxss^{\alpha_j}(Q)$.

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 5). 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 either been used to find an α_jMFS or removed as they contain a found α_jMFS). Some α_jXSSs are then identified simply by executing each α_iXSS and keeping those that are succeeding (lines 9-10).

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 $dmfs^{\alpha_j}(Q) \leftarrow oneAtom$; $FQ \leftarrow mfs^{\alpha_i}(Q) - oneAtom$;
3 **while** $FQ \neq \emptyset$ **do**
4 $Q' \leftarrow fQ.dequeue()$;
5 $Q^* \leftarrow FindAn\alpha MFS(Q', D, \alpha_j)$;
6 $dmfs^{\alpha_j}(Q) \leftarrow dmfs^{\alpha_j}(Q) \cup \{Q^*\}$;
7 **foreach** $Q'' \in FQ$ *such that* $Q^{**} \subseteq Q''$ **do**
8 $FQ \leftarrow FQ - \{Q''\}$;
9 **foreach** $Q^* \in xss^{\alpha_i}(Q)$ *such that* $[[Q^*]]_D^{\alpha_j} \neq \emptyset$ **do**
10 $dxss^{\alpha_j}(Q) \leftarrow dxss^{\alpha_j}(Q) \cup \{Q^*\}$;
11 **return** $\{dmfs^{\alpha_j}(Q), dxss^{\alpha_j}(Q)\}$;

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, then computes the remaining α_jMFSs and α_jXSSs.

4.2 Top-Down Approach

We also consider a Top-Down approach that computes the αMFSs and αXSSs using threshold values in descending 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.

5 Experimental Evaluation

Here we investigate the scalability of our approaches and compare them with the baseline method *NLBA* (executing αLBA for each of the N thresholds).

Experimental Setup. We have implemented the proposed algorithms in JAVA 1.8 64 bits. In our current implementation, these algorithms are run on top of Jena TDB. We chose Jena TDB because it is a quadstore that allows us to store the degree of certainty for each triple. Moreover, Jena TDB provides a low level quad filter hook[2] that we use to retrieve results satisfying the provided threshold. Our implementation is available at https://forge.lias-lab.fr/projects/qars4ukb with a tutorial to reproduce our experiments.

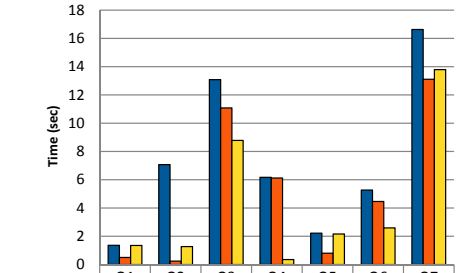
Our experiments were conducted on a Ubuntu Server 16.04 LTS system with Intel XEON CPU E5-2630 v3 @2.4Ghz CPU and 16GB RAM. For our experiments, we chose the *min* aggregate function.

Dataset and Queries. We used a dataset of 20M triples generated with the WatDiv benchmark [5]. The certainty degree of each RDF triple were generated randomly. We consider 7 failing queries[3]. 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).

Experiment. Figure 1 shows the execution times of each algorithm for each query on Jena TDB with the 20M triples dataset. Figure 2 gives the number of executed queries by each algorithm. This experiment has been run with the thresholds arbitrarily set to $\{0.2, 0.4, 0.6, 0.8\}$. In comparison with *NLBA*, our algorithms execute fewer queries for finding the αMFSs and αXSSs of each workload query. Overall, Bottom-Up and Top-Down execute respectively 39% and 44% fewer queries than *NLBA*. As a consequence, these algorithms have shorter execution times (a decrease of respectively 30% and 42% execution times for Bottom-Up and Top-Down). For some queries, this improvement is important. For example, *NLBA* needs 7 seconds to find the αMFSs and αXSSs of the query Q2, whereas our algorithms need around 1 second. Execution times depend heavily on the queries that our algorithms avoid executing. For example, our algorithms execute around 30 queries for Q4 whereas *NLBA* needs 120 queries. For Top-Down, this result is an important performance gain 94%. This is not the case for Bottom-Up that has nearly the same execution time than *NLBA*. By analyzing the executed queries, we find that Bottom-Up prevents the execution of queries that have short execution times but keeps the most expensive ones. Thus, the overall execution time is almost unchanged.

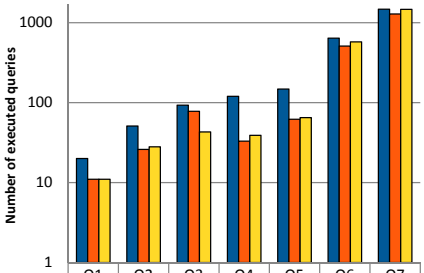
This experiment also shows that none of our algorithms provides the best result for every query. Bottom-Up offers the best execution times for Q1, Q2 and Q5 whereas Top-Down is the most efficient for Q3, Q4 and Q6. Despite executing the least number of queries, Bottom-Up does not offer the best total 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 algorithm executes different queries that have different 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

[2] http://jena.apache.org/documentation/tdb/quadfilter.html
[3] available at https://forge.lias-lab.fr/projects/qars4ukb/wiki/Doc#Queries



	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

Fig. 1. Execution time (20M triples)



	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

Fig. 2. # Executed queries (log scale)

and thus, they 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.

6 Related Work

Several approaches proposed relaxation operators in the RDF context. These operators are mainly based on RDFS semantics [6–8], similarity measures [9, 10] and user preferences [11]. They generate a set of relaxed queries, ordered by similarity with the original query and executed in this order [6, 7, 12]. Relaxation operators are directly used by the user in her/his query [8] or combined with query rewriting rules to perform relaxation [11]. In these approaches, the failure causes of the query are unknown, which may lead to executing unnecessary relaxed queries. Fokou et al. [4] tackled this problem by defining the LBA and MBA approaches to compute the MFSs and XSSs of the query. Our approach is based on the LBA algorithm. We have extended this work by identifying the condition under which LBA can be used in the context of uncertain KBs and by defining two 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 [12]. 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 trust threshold.

7 Conclusion

In this paper, we have considered the empty answer problem in the context of uncertain KBs. 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 can be directly adapted to the context of uncertain KBs. Then, we have studied the problem of computing the αMFSs and αXSSs for multiple thresholds by defining two approaches that consider α thresholds in different orders. We have done a complete implementation of these algorithms and shown experimentally on WatDiv benchmark that our approaches outperform the baseline method.

In our experiments, none of our algorithms has the best performance for all queries. As a future work, we plan to study the conditions under which an algorithm may provide the best results. An analysis of the queries executed by our algorithms shows that they share some triple patterns. Thus, we will investigate multiple-query optimization techniques to further improve their execution times.

References

1. 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
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. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.N.: LSQ: The Linked SPARQL Queries Dataset. In: ISWC'15. (2015) 261–269
4. Fokou, G., Jean, S., Hadjali, A., Baron, M.: Handling Failing RDF Queries: From Diagnosis to Relaxation. Knowledge and Information Systems (KAIS) **50**(1) (2017)
5. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified Stress Testing of RDF Data Management Systems. In: ISWC'14. (2014) 197–212
6. Hurtado, C.A., Poulovassilis, A., Wood, P.T.: Ranking Approximate Answers to Semantic Web Queries. In: ESWC'09. (2009) 263–277
7. 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
8. Calí, A., Frosini, R., Poulovassilis, A., Wood, P.: Flexible Querying for SPARQL. In: ODBASE'14. (2014) 473–490
9. Hogan, A., Mellotte, M., Powell, G., Stampouli, D.: Towards Fuzzy Query-relaxation for RDF. In: ESWC'12. (2012) 687–702
10. Elbassuoni, S., Ramanath, M., Weikum, G.: Query Relaxation for Entity-Relationship Search. In: ESWC'11. (2011) 62–76
11. 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
12. 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