

# On Relaxing Failing Queries over RDF Databases

Paper ID: 257

**Abstract.** In this paper, we present CADER, a novel approach for computing relaxations for failed queries over RDF databases. We show how the number of required database queries for determining all the possible relaxations can be limited to the search of failed subqueries of the user query. Then, we point out how the hitting set problem can be applied for determining the possible relaxed queries in an efficient way without querying databases. Finally, the efficiency and scalability of CADER are shown through extensive experiments on the well-known RDF benchmarks with a variety of queries of different shapes.

**Keywords:** Semantic Web · RDF Databases · Query Relaxation.

## 1 Introduction

RDF (Resource Description Framework) is a declarative data model for the Semantic Web. With the large scale of RDF data, specialized databases, called *RDF databases* (or triple-stores), have been developed to manage large amounts of web data. Many knowledge bases are now defined in the RDF format, e.g., DBpedia [2], Knowledge Vault [6], Bio2RDF [1], Yago2 [14], etc., and they shape a very large set of graphs having millions of triples linked with each other. Unfortunately, the nature of the RDF data structure generally prevents users from correctly formulating queries that return a result for their desired requests. This is why user RDF queries often fail and retrieve an empty set of answers.

In this context, failing queries are a key problem where users ignore whether their requests are too selective, well formulated or if the expected results are simply not present in the base. A promising method for efficiently querying RDF databases consists of relaxing failing RDF queries. *Query relaxation*, i.e., the removal of parts from the original user query, is beside similarity-based retrieval, one of the commonly used techniques to deal with the failing query problem. The main goal is to apply some transformations in basic queries produced by the user in order to acquire approximate answers for his/her initial need. This is especially important in applications such as information retrieval, where getting an alternative answer may be better than not getting an answer at all [5]. Moreover, in recommendation systems query relaxation can help the user to autonomously decide what is the best compromise when not all his wants and needs can be satisfied.

It is worth to mention that manually relaxing failing queries is a frustrating, tedious, and time-consuming process. An automated solution is then required to generate non-failing queries from the initial failed query. To be successful, the following requirements must then be fulfilled:

- **Completeness (R1):** the solution must provide all the possible relaxed queries;

- **Explainability (R2):** it is useful for the user to provide him with all the query’s failures in order to determine possible explanations for potentially wrong queries;
- **Scalability (R3):** the solution must provide an answer even for complex queries with a high number of triple patterns over large databases;
- **Performance (R4):** the solution must have a fast response time to enable users to retrieve the desired data.

Automated query relaxation algorithms have been studied in the context of relational databases and RDF data (see Section 5). To the best of our knowledge, only one work exists in the literature that proposed two approaches called Lattice-Based Approach (LBA) and Matrix-Based Approach (MBA) for the purpose of minimal failing subqueries and maximal succeeding subqueries computation in the RDF context [11]. The former is an adapted and extended variant of Godfrey’s ISHMAEL algorithm [12], while the latter is inspired by the work of Jannach’s method [19], and is based on a matrix called the relaxed matrix. Even though these approaches were shown to be successful, they only deal with limited requirements discussed above (i.e., R1 and R2 under some conditions). Indeed, they are complete and explainable to some extent as no answer is given for queries beyond 15 triple patterns in a reasonable response time. In practice, the results differ in their semantics according to the nature and the structure of the query (star, chain or composite). Besides, the relaxation calculus is extremely linked to the computation of the root causes of failure in the sense that it is not possible to split the two computational processes. Additionally, LBA and MBA focus less on delivering answers in a bounded runtime (R3) and they have a high computational cost, which comes from evaluating a huge number of subqueries against the entire database to compute the possible relaxations (R4).

**Contributions:** It is worthwhile to highlight the major contributions of this paper:

- We propose an algorithm to find all minimal failing subsets of the initial failing user query;
- We present a novel method to model the query relaxation problem as a hitting set problem;
- We propose an efficient algorithm for computing all the possible relaxations based on the strong relationship between minimal failing subsets and minimal hitting set problem;
- We experimentally show that the proposed approach outperforms existing query relaxation methods on synthetic (LUBM) and real world (DBpedia) benchmarks.

These contributions are the basis of our approach named CADER that fulfills the aforementioned requirements. Indeed, it is complete, explainable and scalable as it is able to deliver an answer in a linear response time for large RDF datasets (23 billion of triple patterns). Furthermore, we compared the efficiency of our approach against the previously fastest systems LBA and MBA [11], where we demonstrate that CADER is consistently faster and able to speed up the computation to several orders of magnitude.

## 2 Preliminaries

**RDF.** Given a set  $U$  of URI references, a set  $L$  of literals (constants), and a set  $B$  of blank nodes (unknown URIs or literals) such that  $U$ ,  $B$  and  $L$  are pairwise disjoint,

an RDF database consists of a set of triples  $\langle s, p, o \rangle$  where we refer to  $s \in U \cup B$  as *subject*,  $p \in U$  as *predicate*, and  $o \in U \cup B \cup L$  as *object*. Let  $V$  be an infinite set of variables, disjoint from  $U, B$ , and  $L$ . A *triple pattern*  $t$  is an element of  $(U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$ .  $\text{var}(t)$  denotes the set of variables occurring in  $t$ .

**Queries.** We consider conjunctive RDF queries (e.g., conjunctive SPARQL queries), a central class of queries in database research and widely considered in research but also in real-world applications [20,28]. A conjunctive RDF query (or simply *RDF query*)  $Q$  is defined as a *conjunction* of triple patterns, i.e.,  $Q = t_1 \wedge \dots \wedge t_n$ . Given a query  $Q = t_1 \wedge \dots \wedge t_n$ , a query  $Q' = t_i \wedge \dots \wedge t_j$  is a *subquery* of  $Q$  iff  $\{t_i, \dots, t_j\} \subseteq \{t_1, \dots, t_n\}$ . In addition,  $Q'$  is a *proper super-query* of the query  $Q$  iff  $\{t_1, \dots, t_n\} \subset \{t_i, \dots, t_j\}$ .

To define the semantics of query evaluation, we need to recall some further notation. A *mapping*  $\mu$  is a partial function  $\mu : V \rightarrow U \cup B \cup L$ . The domain of  $\mu$ , denoted by  $\text{dom}(\mu)$ , is the subset of  $V$  where  $\mu$  is defined. Given a mapping  $\mu$  and a triple pattern  $t$  s.t.  $\text{var}(t) \subseteq \text{dom}(\mu)$ , we have that  $\mu(t)$  is the result of replacing every variable  $x \in \text{var}(t)$  by  $\mu(x)$ . A mapping  $\mu_1$  is said to be *compatible* with a mapping  $\mu_2$ , written as  $\mu_1 \mu_2$ , if for all  $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ ,  $\mu_1(x) = \mu_2(x)$ . Let  $\Omega_1$  and  $\Omega_2$  be sets of mappings. Then, the *join* of  $\Omega_1$  and  $\Omega_2$  is defined as:  $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \mu_2\}$ . Last, given an RDF database  $\mathcal{D}$  and a query  $Q$ , the evaluation of  $Q$  over  $\mathcal{D}$ , denoted by  $[[Q]]_{\mathcal{D}}$ , is recursively defined as:

- if  $Q$  is a triple pattern, then  $[[Q]]_{\mathcal{D}} = \{\mu \mid \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in \mathcal{D}\}$ ,
- if  $Q$  is  $t_1 \wedge t_2$ , then  $[[Q]]_{\mathcal{D}} = [[t_1]]_{\mathcal{D}} \bowtie [[t_2]]_{\mathcal{D}}$ .

A common RDF query problem for users is to face with empty answers, i.e.,  $[[Q]]_{\mathcal{D}} = \emptyset$ . Usually, only some parts of  $Q$  are responsible of its failure. Finding such subqueries provides the user with an explanation of the empty result returned and a guide to relax the original query. More formally, such notion of causes of failure can be characterized as follows. Let us first denote by  $\Sigma = \{t_1, \dots, t_n\}$  the set of triple patterns contained in the query  $Q = t_1 \wedge \dots \wedge t_n$ . We refer to the conjunction of the set of triple patterns  $\{t_1, \dots, t_n\}$  of  $\Sigma$  as  $\bigwedge \Sigma$ . Obviously,  $\bigwedge \Sigma = Q$ .

**Definition 1.** Let  $\mathcal{D}$  be an RDF database. A *Minimal Failing Subset*  $\Gamma$  of a query  $Q$  over  $\mathcal{D}$ , denoted by *MFS*, is a set of triple patterns s.t. (i)  $\Gamma \subseteq \Sigma$ , (ii)  $[[\bigwedge \Gamma]]_{\mathcal{D}} = \emptyset$ , and (iii)  $\forall \Gamma' \subset \Gamma : [[\bigwedge \Gamma']]_{\mathcal{D}} \neq \emptyset$ .

That is, an *MFS* of  $Q$  over the RDF database  $\mathcal{D}$  is a subset of triple patterns that is *failing* and *minimal* in the sense that removing any one of its elements makes the remaining set of triple patterns *succeed*. Hence, an *MFS* can be seen as an irreducible cause of the failing of the original query.

A dual concept of *MFS* is the notion of *MaXimal Succeeding Subset* of an RDF query, which we define as follows:

**Definition 2.** Let  $\mathcal{D}$  be an RDF database. A *maXimal Succeeding Subset*  $\Gamma$  of a query  $Q$  over  $\mathcal{D}$ , denoted by *XSS*, is a set of triple patterns s.t. (i)  $\Gamma \subseteq \Sigma$ , (ii)  $[[\bigwedge \Gamma]]_{\mathcal{D}} \neq \emptyset$ , and (iii)  $\forall \Gamma' \text{ with } \Gamma \subset \Gamma' \subseteq \Sigma : [[\bigwedge \Gamma']]_{\mathcal{D}} = \emptyset$ .

Given a failing query  $Q$ , an *XSS*  $\Gamma$  for  $Q$  over the RDF database  $\mathcal{D}$  is a non-failing subset of  $Q$  (i.e.  $\bigwedge \Gamma$  is a succeeding subquery) and there exists no other query  $Q'$  which

is also a non-failing subquery of  $Q$  such that  $\bigwedge \Gamma$  is a subquery of  $Q'$ . Obviously, given a failing RDF query  $Q$  and  $\Gamma$  an XSS of  $Q$ , then  $\bigwedge \Gamma$  is a relaxed query of  $Q$ .

For convenience, we use XSSes and MFSes to denote the set of all maximal succeeding and minimal failing subsets of  $Q$ , respectively. Note that in the worst case the number of XSSes and MFSes is exponential in the number of triple patterns in  $Q$ .

*Example 1.* Consider the database inspired by the LUBM Benchmark [13]. Let us consider the following query  $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4 \wedge t_5 \wedge t_6$ :

```
SELECT * WHERE {
  ?Y10 hasAlumnus ?Y11 .           (t1)
  ?Y11 advisor ?Y12 .              (t2)
  ?Y12 advisor ?Y13 .              (t3)
  ?Y13 doctoralDegreeFrom ?Y14 .   (t4)
  ?Y14 hasAlumnus ?Y15 .           (t5)
  ?Y15 title 'Dr' }                (t6)
```

This query fails against LUBM since the sets  $\{t_1\}$ ,  $\{t_3, t_6\}$ ,  $\{t_2, t_3, t_4\}$ , and  $\{t_2, t_3, t_5\}$  are MFSes of  $Q$ . Hence,  $\text{XSSes}(Q) = \{\{t_2, t_3\}, \{t_3, t_4, t_5\}, \{t_2, t_4, t_5, t_6\}\}$ .

### 3 An Efficient Complete Approach for RDF Query Relaxation

In this section, we introduce a new efficient and complete technique, coined CADER (Complete Approach for RDF QuEry Relaxation), for determining possible relaxations over RDF databases in an efficient way to overcome the limitations of existing approaches [11]. Our method is mainly based on the strong duality between MFSes and XSSes. This duality is shown to be the cornerstone of the efficiency of our approach. CADER identifies the root causes of failure and computes all relaxations when the user query fails. In other words, we propose to evaluate the subqueries of the user query individually in a preprocessing step and base the subsequent computation of relaxations on these intermediate results. Our main idea is based on the fact that, in practice, it is more efficient to find minimal failing subsets of triple patterns than maximal succeeding ones. Further, we show how we can efficiently determine all the possible relaxations from minimal failing subsets directly without querying the RDF database. Besides, extracting minimal failing subsets is helpful in many applications, because it emphasize what is wrong with a failing query.

In a nutshell, our method can be summarized in the following steps: First, we calculate all minimal failing subsets of the user query before computing the entire set of relaxed queries. After that, we compute the hitting sets of these MFSes. These hitting sets are in fact the complement of all the possible relaxations of the failing user query. Accordingly, the complete set of relaxed queries is computed from the set of hitting sets in a direct way by taking the complement of each hitting set.

#### 3.1 Step 1: Finding all MFSes

The basic idea of this step is to search for all the triple patterns that raise a problem following a logical principle. A failing query could have multiple reasons for its infeasibility. In this case, the query would contain multiple MFSes, and fixing any single

MFS may not make it successful. As long as any MFS is contained in the query, it will remain infeasible.

Our approach keeps on searching for reasons of failure by browsing different subqueries of the original failing query. More precisely, it finds an MFS by decomposing the initial failing query into subqueries, starting with those with the lowest number of triples to those with the largest one, and all MFSes are computed gradually. The task is based on the following principle:

Let  $\mathcal{D}$  be an RDF database. Assume a failing user query  $Q = t_1 \wedge \dots \wedge t_n$  and  $Q_i = t_i$  ( $t_i \in \Sigma$ ,  $1 \leq i \leq n$ ) is a subquery of  $Q$ . Clearly, an initial search runs for query evaluation on these initial subqueries  $Q_i$  ( $1 \leq i \leq n$ ) (i.e.,  $[[Q_i]]_{\mathcal{D}} \stackrel{?}{=} \emptyset$ ) might encounter some actual MFSes. Our method makes use of a sliding approach allowing for an incremental search in the sense that it computes MFSes of increasing size. In this respect, if the subquery  $Q_i$  is failing, then  $\{Q_i\}$  is an MFS as it is minimal w.r.t. subset inclusion. Clearly, we do not need to test the proper super-queries of  $Q_i$  since, despite these super-queries are guaranteed to fail, they cannot be actual MFSes as they are not minimal w.r.t. set-theoretic inclusion. This progressive increment ensures that the exponential search space is reduced significantly.

Now, if the subquery  $Q_i$  succeed, this does not mean that all proper super-queries of  $Q_i$  are succeeding on their turn and an exhaustive search is performed. In other words, the evaluation check is repeated on the subsets  $Q_j = Q_i \wedge t_k$ ,  $t_k \in \Sigma \setminus \{Q_i\}$ , successively, until  $\Sigma$  is empty. Then, all MFSes have been found. Once again, when all the remaining MFSes are recorded, our incremental approach is able to exploit this information in a very valuable and efficient way. Algorithm 1, that we call **ALL\_MFSes** (Computing All MFSes), sketches out the pseudocode of the approach.

Algorithm 1 takes as an input an RDF database  $\mathcal{D}$ , a failing query  $Q$  and the set of triple patterns  $\Sigma$  of  $Q$ , and outputs the set of MFSes of  $Q$ . The algorithm starts with the set of sets  $\beta$  that initially contains the triple patterns of  $Q$  (line 1). Then, in each iteration, if the subquery  $\bigwedge \Gamma$  fails against  $\mathcal{D}$  (line 5), the subset  $\Gamma$  is added to the solution set  $\mathcal{M}$  as it is minimal w.r.t. subset inclusion (line 6); otherwise, Algorithm 1 augments  $S$  with  $\Gamma$  (line 8). Now, if  $S$  is not empty, we call the method **QueryComposition** (line 10) to get the next candidate MFSes as follows: for each pair of distinct sets  $X$  and  $Y$  from  $S$ , Algorithm 1 first checks if  $X$  and  $Y$  share exactly  $|X| - 1$  triple patterns s.t.  $(X \uplus Y)$  does not contain any MFS of  $\mathcal{M}$  (to prevent the same MFS from being computed again). If that is the case, this implies that  $(X \cup Y)$  could be a potential MFS and Algorithm 1 adds  $(X \cup Y)$  to  $\beta$ . Then, the evaluation check is repeated on the new subsets of  $\beta$  (lines 4-8). Finally, if  $S$  is empty, the entire set of MFSes has been found, and the algorithm terminates. Figure 1 shows the working of our algorithm on Example 1. All MFSes are depicted with red color and the set of proper super-queries of every MFS is omitted.

### 3.2 Step 2: Finding all Relaxed Queries

In this step, we present our complete approach for automatically finding all relaxed queries which approximately meet the original query intention. Interestingly, once all MFSes are computed in Step 1, we will show how these MFSes allow us to enumerate the complete set of maximal succeeding subsets without any access to the RDF

---

**Algorithm 1:** ALL\_MFSes (Computing All MFSes)
 

---

**Input:**  $\mathcal{D}$ : an RDF database,  $Q$ : a failing query,  $\Sigma$ : the set of triple patterns of  $Q$ 
**Output:**  $\mathcal{M}$ : the set of all MFSes of  $Q$ 

```

1  $\mathcal{M} \leftarrow \emptyset; S \leftarrow \emptyset; \beta \leftarrow \bigcup \{t_i\}, t_i \in \Sigma;$ 
2  $loop = True;$ 
3 while ( $loop = True$ ) do
4   for ( $\Gamma \in \beta$ ) do
5     if ( $(\llbracket \bigwedge \Gamma \rrbracket_{\mathcal{D}} = \emptyset)$ ) then
6        $\mathcal{M} \leftarrow \mathcal{M} \cup \{\Gamma\};$ 
7     else
8        $S \leftarrow S \cup \{\Gamma\};$ 
9   if ( $S \neq \emptyset$ ) then
10     $\beta \leftarrow QueryComposition(X, Y), (X, Y) \in S \text{ with } (X \uplus Y) \not\subseteq Z (Z \in \mathcal{M}),$ 
11     $|X \cap Y| = |X| - 1;$ 
12     $S \leftarrow \emptyset;$ 
13  else
14     $loop = False;$ 
15 return  $\mathcal{M};$ 

```

---

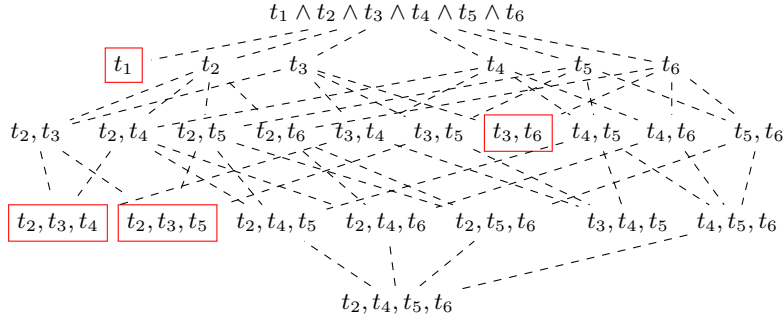


Fig. 1: This figure illustrates the search process in Algorithm 1 to compute all the MFSes of  $Q$  (Example 1)

database. To do so, we start firstly by searching the hitting sets of MFSes which correspond to a *set-theoretical complement* of XSSes w.r.t. a failing RDF query. Formally, we define this notion as follows.

**Definition 3.** Let  $Q = t_1 \wedge \dots \wedge t_n$  be a failing RDF query and  $\Sigma = \{t_1, \dots, t_n\}$  be the set of triple patterns of  $Q$ . Then, the complement of an XSS  $\Gamma$  of  $Q$ , denoted by CoXSS, is defined as  $\Sigma \setminus \Gamma$ .

As a shortcut, the set of all complement maximally succeeding subsets of  $Q$  will be identified with CoXSSes.

**Corollary 1.** *Let  $Q = t_1 \wedge \dots \wedge t_n$  be a failing RDF query and  $\Sigma = \{t_1, \dots, t_n\}$  be the set of triple patterns of  $Q$ . Then, we have  $\text{CoXSses}(Q) = \{\Gamma \subseteq \Sigma \mid (\Sigma \setminus \Gamma) \in \text{XSses}(Q)\}$ .*

The next property shows that a CoXSS is a set of triple patterns not included in some minimal failing subsets.

**Proposition 1.** *Let  $Q$  be a failing RDF query. Let  $X$  be the set of MFSes of  $Q$ . Then, a CoXSS of  $Q$  contains at least one triple pattern from each MFS  $\Gamma \in X$ .*

*Proof.* (Sketch) Because the presence of any MFS in a query  $Q$  makes  $Q$  failing, at least one triple pattern from every MFS in  $Q$  must be removed to make it successful. That is, given a failing query  $Q$ ,  $\Sigma$  the set of triple patterns of  $Q$ , and a set of triple patterns  $\Gamma \subseteq \Sigma$ ,  $\Gamma' = \Sigma \setminus \Gamma$  is successful if and only if  $\Gamma$  contains at least one triple pattern from every MFS in  $Q$ . Therefore, a set in  $\text{CoXSses}(Q)$  must contain at least one triple pattern from every MFS of  $Q$ .

Proposition 1 is based on the fact that any succeeding subset of a failing query cannot fully contain any failing subset, and thus this successful subset must exclude at least one triple pattern from every failing set. This relationship will be exploited in the relaxation analysis by using one type of result to guide searches for the other.

The efficiency of our approach relies on the crucial concept of *hitting set*, which appears to be a powerful ingredient of our technique to locate all XSses of the failing user query. For this, let us firstly introduce the concept of hitting sets.

**Definition 4.** *Given a collection  $\Omega$  of sets. A set  $H$  is a hitting set for  $\Omega$  if  $H \subseteq \bigcup \Omega$ , and  $\forall S \in \Omega, H \cap S \neq \emptyset$ . If no proper subset of  $H$  is a hitting set for  $\Omega$ , then  $H$  is a minimal hitting set for  $\Omega$ .*

Interestingly, the relationship between CoXSses and the set of MFSes of a failing RDF query stated by Proposition 1 can be described as a solution to a set covering problem. More specifically, the following result shows that our method computes the complete set of CoXSses from the set of MFSes via the hitting set problem.

**Proposition 2.** *Let  $Q$  be a failing RDF query. Then, a CoXSS of  $Q$  is a minimal hitting set of the set of MFSes of  $Q$ .*

*Proof.* Let  $Q$  be a failing RDF query and  $\Sigma$  the set of triples of  $Q$ . Clearly, each MFS is a subset of the triples of  $\Sigma$ . And, a hitting set of the collection of MFSes is a set of triples that contains at least one triple from every MFS of the query  $Q$ . By Proposition 1, a set in CoXSses is thus an irreducible hitting set of the set of the MFSes of  $Q$ .

Intuitively, Proposition 2 shows that the set of minimal failing sets implicitly encodes the entire set of relaxations of the original failing query. The relationship between MFSes and CoXSses is that they are *hitting set duals* of one another, that is, each CoXSS has at least one triple pattern in common with all MFSes (and is minimal in this sense), and vice versa. More precisely, a hitting set of a collection of sets, is a set that contains at least one element from each set in the collection. In this case, the

collection is the set of MFSes, and a CoXSS is a set of triple patterns that contains at least one triple pattern from every MFS. Note that a CoXSS is a hitting set of the MFSes with the additional restriction that it cannot be any smaller without losing its defining property: it is an *minimal hitting set*. Accordingly, a CoXSS is a minimal hitting set of the set of MFSes that represent minimal sets of triple patterns that should be dropped in order to restore consistency of the original query. In a dual way, every MFS of a failing query is also a minimal intersecting set of CoXSSes. In this way, although the minimal hitting set is a difficult problem [22], its irreducibility makes this problem less resource-intensive. So, from this particularity a CoXSS can even be deduced in polynomial time from the set of MFSes.

Now, once the entire set of CoXSSes has been computed, the second phase of Step 2 produces the set of all XSSes of the original failing RDF query, as stated in Corollary 1. More precisely, given a CoXSS  $\Gamma$  of the failing RDF query  $Q$  with  $\Sigma$  the set of triple patterns of  $Q$ , then the corresponding XSS of  $Q$  is  $\Gamma' = \Sigma \setminus \Gamma$ . Consequently,  $\bigwedge \Gamma'$  is a relaxed query of  $Q$ . Obviously enough, the number of relaxed queries of  $Q$  is equal to the cardinality of the set of CoXSSes. Interestingly, contrary to the existing approaches that query the RDF database at least once in order to find an XSS [11], our computing process of all relaxed queries is done in a direct way without querying databases.

In light of Proposition 2, we present our Algorithm 2, called **ALL\_RQ** (Computing All Relaxed Queries) to determine the possible relaxed queries. It is based on a counting heuristic grafted to the Minimal-to-Maximal Conversion Search (MMCS) algorithm [24], which explores the set of MFSes in order to compute efficiently the minimal hitting sets of these MFSes. These minimal hitting sets are all CoXSSes of the user query. Then, our algorithm computes the complete set of XSSes from the set of CoXSSes in a straightforward manner. Finally, all relaxed queries are generated using the set of XSSes. For instance, let us consider the set of MFSes returned by Algorithm 1:  $\text{MFSes} = \{\{t_1\}, \{t_3, t_6\}, \{t_2, t_3, t_4\}, \{t_2, t_3, t_5\}\}$ . At this step, the minimal hitting sets are computed to output the complete set of CoXSSes  $= \{\{t_1, t_3\}, \{t_1, t_2, t_6\}, \{t_1, t_4, t_5, t_6\}\}$ . For every CoXSS found previously, Algorithm 2 deletes this complement from the initial set of triple patterns  $\Sigma$  and only keeps those that are not included in the corresponding CoXSS. Then, we obtain  $\{t_2, t_3\}$ ,  $\{t_3, t_4, t_5\}$  and  $\{t_2, t_4, t_5, t_6\}$  which are the set of all XSSes of the initial query. The set of all relaxed queries of the original query  $Q$  are then  $t_2 \wedge t_3$ ,  $t_3 \wedge t_4 \wedge t_5$ , and  $t_2 \wedge t_4 \wedge t_5 \wedge t_6$ .

---

**Algorithm 2:** ALL\_RQ (Computing All Relaxed Queries)

---

**Input:**  $\mathcal{M}$ : the set of MFSes of the query  $Q$ ,  $\Sigma$ : the set of triple patterns of  $Q$

**Output:**  $\mathcal{RQ}$ : the set of all relaxed queries of  $Q$

---

```

1  $\mathcal{RQ} \leftarrow \emptyset;$ 
2  $\text{CoXSSes} \leftarrow \text{MMCS}(\text{MFSes});$ 
3 for  $\Gamma \in \text{CoXSSes}$  do
4    $\Gamma' \leftarrow \Sigma \setminus \Gamma;$ 
5    $\mathcal{RQ} \leftarrow \bigwedge \Gamma' \cup \mathcal{RQ};$ 
6 return  $\mathcal{RQ};$ 
```

---



**Proposition 3.** *Algorithm ALLRQ is sound and complete, i.e., it returns exactly all relaxed queries for a failing user query  $Q$ .*

*Proof.* Given the set of MFSeS of the failing query  $Q$ , it is guaranteed that MMCS algorithm returns exactly all the minimal hitting sets that correspond to the set of all CoXSSes of  $Q$ . That is, by Corollary 1 each XSS found is maximal (soundness) and Algorithm 2 finds all XSSes (completeness). Consequently, every relaxed query will be found by Algorithm 2.

## 4 Empirical Investigation

In this section, we report different experiments to evaluate the efficiency and scalability of CADER. For baseline comparison, we retain the dedicated algorithms LBA and MBA [11], which in turn are shown more competitive than all existing approaches by [12], and the depth-first search algorithm [10].

### 4.1 Experimental Settings

**Software & Hardware.** CADER has been implemented in Java, with Jena library. The database system used in the implementation is Apache Jena. For the second step of it, the complete search of all relaxed queries is based on the use of the MMCS algorithm [24], which is currently one of the best modern solver that enumerates all minimal hitting sets. Tests were performed on a 2Ghz Intel core i7 PC with 16GB of memory, and running Ubuntu Linux 16.04.

**Data.** We conducted experiments using *real* and *synthetic* datasets. The DBpedia [21] dataset consists of real data gathered from Wikipedia. The DBpedia ontology consists of 320 classes which form a subsumption hierarchy and are described by 1650 different properties. We also used the well-known LUBM dataset [13], a synthetic ontology developed to benchmark knowledge base systems w.r.t. large OWL applications. The ontology is situated in the university domain, and is based on 43 classes and 32 properties hierarchies. LUBM is widely used by the Semantic Web community to compare performance especially when arbitrarily large datasets are required. Using the LUBM data generator [13], we have created test sets ranging from 65 millions (LUBM500) to 2 billion (LUBM15k) triples, to obtain datasets of incremental sizes. The main characteristics of the different datasets are summarized in Table 1.

**Queries.** To evaluate the query relaxation performance of our system, we have generated twenty one queries of varying shape over the DBpedia dataset:  $Q_1$ - $Q_7$  star-shaped,  $Q_8$ - $Q_{15}$  chain-shaped, and  $Q_{16}$ - $Q_{21}$  composite-shaped queries. We also borrowed from [11] a set of 197 LUBM queries, having between 2 and 15 triple patterns.

### 4.2 Experimental Results

Various experiments have been carried out and all results are obtained by averaging over 5 runs. Queries whose evaluation requires more than 10 seconds were interrupted. All data, queries, results and a user interface are available from [goo.gl/1fZ8us](http://goo.gl/1fZ8us). The graph in Figure 2 shows, for each DBpedia query the number of triple patterns in the

Dataset	#Triples	#Instances
DBpedia	23B	4.2M
LUBM500	65M	11M
LUBM 1K	110M	28M
LUBM 3K	320M	90M
LUBM 5K	550M	140M
LUBM 7K	720M	195M
LUBM 10K	1.3B	270M
LUBM 15K	2B	360M

Table 1: Statistics of datasets used in experiments

query (in parentheses after the query name). For each method, the reported runtime is in milliseconds and it includes the time to compute all the MFSes and the relaxed queries <sup>1</sup> for each RDF query against DBpedia dataset. First, for all methods, the query relaxation response time generally increases as the number of triple patterns grows, except for *Q1*. In fact, the running time for *Q1* includes also the time spent by the algorithm to initialize the connection to the database. The same figure also shows that our algorithm is able to scale for all queries and returns all the possible relaxations in a very short period of time (at most 195 ms), which is not the case for LBA and MBA algorithms. For instance, the baselines were unable to compute all MFSes and XSSes of the query *Q15* within 10 seconds CPU time. Further, as we can see from Figure 2, the running time for LBA and MBA increases for some queries with large number of triple patterns. This time can reach 8401 ms and 1714 ms for baselines for *Q7* and *Q12*, respectively. This can be explained by the fact that for larger queries, the number of executed subqueries exponentially increases for LBA and MBA. Thus, the performance of these two baselines quickly decreases for these queries. Also, MBA used a matrix to identify subqueries of the original query which let the computation time more important than LBA for some queries (e.g., *Q6*), since the size of the matrix can be large for those queries. Moreover, as can be seen, LBA and MBA are both very much slower than CADER, specially for queries with large number of MFSes (e.g. *Q6* with 62 MFSes, and *Q7* with 78 MFSes), since they need to identify the set of XSSes once an MFS is found. These XSSes should not include the MFS previously computed, and this process is recursively applied by querying the RDF database once another MFS is found. Overall, Figure 2 further highlights that CADER is always the best regardless of the shape of DBpedia queries. Interestingly, by allowing complete sets of relaxed queries to be delivered in a shorter time from the set of MFSes, CADER is about 70X, 37X, 28X and 32X faster than baselines for the queries *Q9*, *Q10*, *Q11* and *Q17*, respectively. This confirms the important performance improvement brought by our system to query relaxation in RDF.

In addition, Figure 3 depicts the time spent by CADER to compute MFSes and relaxed queries separately. This figure demonstrates that the additional time to compute all relaxed queries from the set MFSes is often very small (at most 14 ms), thanks to

<sup>1</sup> In this section, we use the terms XSS and relaxed query interchangeably.

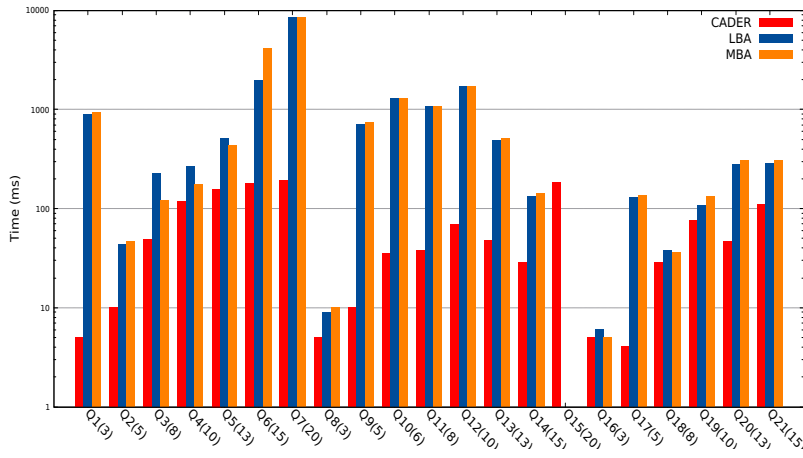


Fig. 2: Query relaxation times for DBpedia queries

the MMCS hitting set algorithm. This shows clearly that the query evaluation against the underlying database is typically the most costly operation.

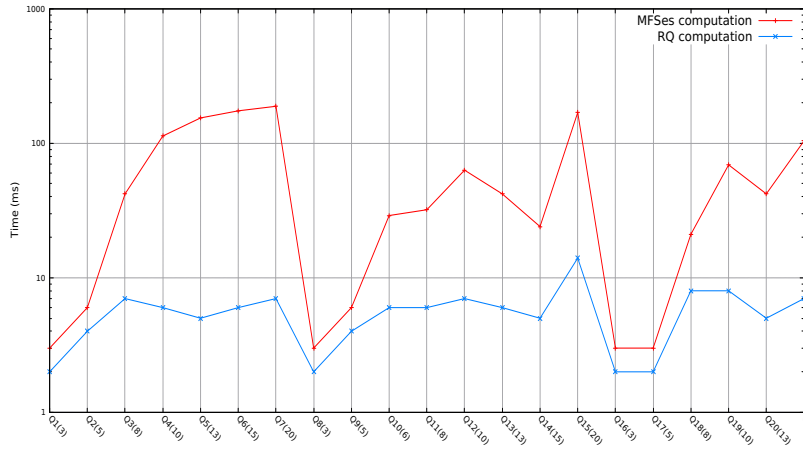


Fig. 3: MFSeS and RQ running times for DBpedia queries

**Evaluating scalability.** For more deeper analysis, in the next set of experiments, we evaluate the scalability of query relaxation methods by measuring each method’s running time on LUBM datasets as we increase the instance size. Figure 4 presents the average runtime of the different algorithms for the 197 LUBM queries (69 star-shaped, 65 chain-shaped, and 63 composite-shaped queries), when the dataset ranges from 65M to 2B triples. Clearly, a significant time gap can be observed in favor of CADER. The

efficiency gain ratio is even more significant when the size of the dataset increases. More precisely, for LUBM $x$ k ( $x \in \{7, 10, 15\}$ ), LBA and MBA running times become less effective and slower, as they need more than 100 ms to relax all the queries. In contrast, CADER’s performance is robust, the best in all cases. The running time is then very small, between 30 ms (on LUBM500) and 36 ms (on LUBM15k). Overall, CADER has a considerable advantage in scalability, as it is about 3 order of magnitude faster than baselines. This behavior is explained by the fact that CADER exploits earlier information (i.e., MFSes) to enumerate all relaxed queries without further querying the database. However, LBA and MBA need more additional scans of the RDF dataset to determine the set of MFSes and XSSes. Moreover, this issue can be interpreted by the fact that the LUBM composite queries are the most complicated ones (as they involve different complex join query patterns and not only object-subject join pattern like with chain-shaped queries) and the evaluation of this type of queries requires additional computation time. Hence, LBA and MBA can get into trouble for larger datasets.

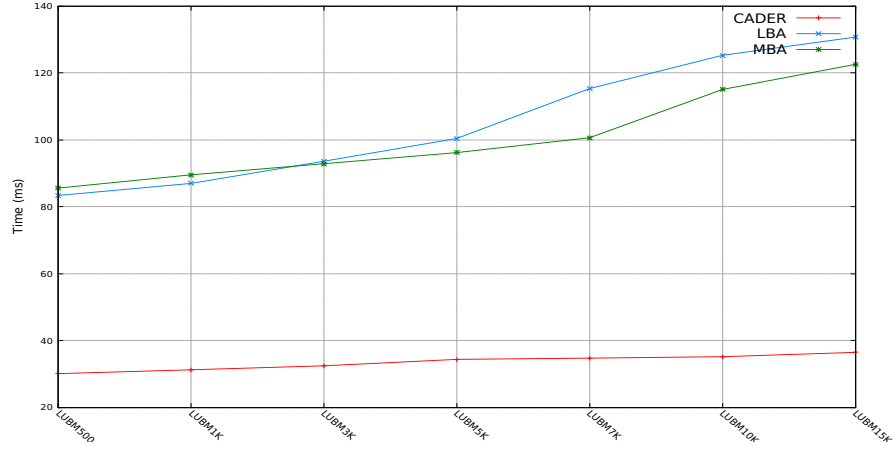


Fig. 4: Execution time vs LUBM data size

We end this section by claiming that the intuition of why our approach is clearly more efficient and scalable than the baselines, is the fact that finding relaxed queries can be done in a more efficient way by formulating this problem as a minimal hitting set problem. Experiments indicate also that the performance gains depend on a number of factors including the number of triple patterns of the query, the type of query itself and also the size of the RDF dataset. Hence, the poor performance of LBA and MBA correlates with the large datasets and the size of the query.

In summary, the following conclusions can be drawn from the empirical evaluation:

1. CADER is consistently faster and able to speed up computation to several orders of magnitude w.r.t. LBA and MBA;
2. CADER is able to handle large RDF queries with different shapes (star, chain, composite);

3. the efficiency gain ratio of CADER is even more significant when the size of the dataset increased;
4. the additional time spent by CADER to compute all relaxed queries from the set of MFSeS is often very small.

### 4.3 User Interface

We provide a demo application in which users can relax failing RDF queries (see Figure 5). We offer three different interaction scenarios in our user interface:

**Dataset Loading:** Users can select a dataset to load. The dataset will then be added to the list of benchmarks. After that, users choose from the list the dataset to querying.

**Query Evaluation:** Users can write any SPARQL query or choose one of the predefined queries. The predefined queries are either selected from a real query log (DBpedia, Freebase), or they are benchmark queries (LUBM and WatDiv). predefined queries are selected such that they provide a mixture of queries with varying structural characteristics and shapes. Once the query and the associated dataset are defined, the user can choose the underlying execution engine from LBA, MBA, or CADER. Then, the query is submitted through the GUI and relaxed using the chosen system.

**Performance Comparison:** By deploying different engines as well as CADER, we allow the user to compare these systems head-to-head (see Figure 6). The user can also download the summary of the comparison.

Fig. 5: Demonstration Interface

## 5 Related Work

Query relaxation has been extensively investigated in the Artificial Intelligence, database and information retrieval literature. In this section, we review the closest works related

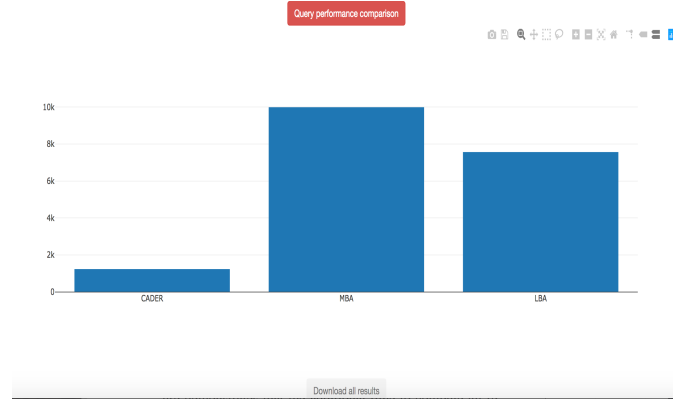


Fig. 6: Screenshot of Performance comparison

to our proposal done both in the context of relational databases and RDF query relaxation. In [12], Godfrey was the first author who proposed to look for constraints responsible for the query’s failure and defined the notion of minimal failing subquery in relational databases but with high computational cost considering that the problem of finding all minimal failing subqueries of a failing query is NP-hard. Additionally, McSherry [23] and Jannach [19] have applied the same relaxation techniques to the recommender systems domain where the former exploits a lattice of subqueries by searching the minimal causes of failure of the original query from the lowest number of constraints to the largest ones, while the latter leverages a particular matrix to find the minimum relaxations, and computes a successful subquery.

In addition, Muslea and Lee proposed an online approach based on bayesian causal structures discovery to generate non-failing queries that are highly similar to the original failed query [26,25]. Later, several works have been introduced to relax queries in RDF graphs by generalizing triple patterns using class and property hierarchies [17,4], similarity measures [15,7,8], query rewriting [3] and also user preferences that are used for getting Top- $k$  answers [18,16]. Other methods such as relaxation operators [9,4] and query rewriting rules have been also studied in order to perform the relaxation procedure. A recent work in [29] proposed a method to solve the empty-answer problem by projecting the RDF graph into a continuous vector space.

Unfortunately, as claimed in [11] the main limitation of these previous works on RDF query relaxation is that the user is still not able to identify neither the root causes of failure of his/her query, nor the possible reasons of the wrong behaviour. Therefore, the user can start by relaxing triple patterns that are not responsible of the main query’s failure, which can be a very time-consuming in processing and does not need to be modified in the original query. Hence, a long processing time to effectively relax the triple patterns responsible of the main query’s failure will be consumed; while, there is a trade-off between optimality of the relaxation and the response time.

To the best of our knowledge, the only work that investigated the issue of computing minimal causes of failure for failing RDF queries is the one proposed by Fokou and al

[11]. The authors introduced two methods called *Lattice-Based Approach* (LBA) and *Matrix-Based Approach* (MBA) for the purpose of minimal failing subqueries and maximal succeeding subqueries computation in the RDF context. The former is an adapted and extended variant of Godfrey’s ISHMAEL algorithm [12]. More precisely, LBA is a three-steps procedure: (i) finding an MFS of the original failing query, (ii) computing potential XSSes, i.e., the maximal queries that do not include the MFS previously found, and (iii) executing potential XSSes; if they retrieve results, they are XSSes, else this process is applied recursively on failing potential XSSes. MBA is inspired by the work of Jannach’s method [19], and is based on a matrix called the relaxed matrix. Then, the XSSes of the original failing query are obtained from that relaxed matrix.

In spite of that, the LBA and MBA techniques are not complete in the sense that they do not necessarily deliver all minimal failing and maximal succeeding subqueries for larger queries and become impractical for queries with complicated shape (chain and composite), since they explore an exponential search space. Moreover, the computation of MFSes and XSSes need to query many times the database in a repetitive way.

Conflicts and relaxations are studied and used in many areas of automated reasoning such as truth maintenance systems (TMS), non-monotonic reasoning, model-based diagnosis, intelligent search, and recently explanations for constraint satisfaction problems (CSPs). In particular in the field of Model-Based Diagnosis, causes of failure are often used as a basis to systematically determine possible explanations, i.e. diagnoses, for an unexpected behavior of the system under observation [27]. We thus based our contribution on these concepts to find the set of all relaxations of a failing RDF query driven by the causes of failure which can be considered as explanations that may help the user to understand what is wrong in his/her query.

## 6 Summary & Outlook

Computing all failure causes and query relaxations are highly intractable issues in the worst case. However, it can make sense to compute them for some real-life applications, e.g., user query relaxation in information retrieval. In this paper, we have developed an efficient technique for finding the set of all the possible relaxations of a failing RDF query as well as the set of all the causes of failure that fall to be assessed as explanations. Our approach is based on a strong relationship between failing subqueries and the minimal hitting set problem, so that finding all relaxed queries can be accomplished by finding the minimal hitting set of the MFSes of the original user query. This relationship has the advantage to reduce the number of the RDF database access, which is really very time-consuming and costly. Experimental results show that our approach clearly outperforms drastically the state-of-the-art techniques for RDF query relaxation on large benchmarks, e.g., LUBM10k and LUBM15k. Moreover, CADER scales up well for large queries, while the computation process studied in the related work can be stopped at any time while still having issues to scale with larger queries.

In the future, we plan to study how possible query relaxations could be computed progressively from the growing set of extracted MFSes. An optimization of Algorithm 1 to compute MFSes more efficiency is part of our future work. Moreover, we intend to study the computation of top- $k$  relaxed queries according to user preference functions.

We shall finally investigate the applicability of our approach and tackle the problem of query relaxation on RDF (uncertain) data streams for efficient processing.

## References

1. Belleau, F., Nolin, M., Tourigny, N., Rigault, P., Morissette, J.: Bio2rdf: Towards a mashup to build bioinformatics knowledge systems. *Journal of Biomedical Informatics* **41**(5), 706–716 (2008)
2. Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: Dbpedia - A crystallization point for the web of data. *J. Web Sem.* **7**(3), 154–165 (2009)
3. Bursztyn, D., Goasdoué, F., Manolescu, I.: Optimizing reformulation-based query answering in RDF. In: *EDBT*. pp. 265–276 (2015)
4. Cali, A., Frosini, R., Poulouvasilis, A., Wood, P.T.: Flexible querying for SPARQL. In: *CoopIS, and ODBASE*. pp. 473–490 (2014)
5. Carmel, D., Yom-Tov, E.: Estimating the Query Difficulty for Information Retrieval. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, Morgan & Claypool Publishers (2010)
6. Dong, X., Gabrilovich, E., Heitz, G., Horn, W., Lao, N., Murphy, K., Strohmman, T., Sun, S., Zhang, W.: Knowledge vault: a web-scale approach to probabilistic knowledge fusion. In: *KDD*. pp. 601–610 (2014)
7. Elbassuoni, S., Ramanath, M., Weikum, G.: Query relaxation for entity-relationship search. In: *ESWC*. pp. 62–76 (2011)
8. Ferré, S.: Answers partitioning and lazy joins for efficient query relaxation and application to similarity search. In: *ESWC*. pp. 209–224 (2018)
9. Fokou, G., Jean, S., Hadjali, A.: Endowing semantic query languages with advanced relaxation capabilities. In: *Foundations of Intelligent Systems - 21st International Symposium, ISMIS*. pp. 512–517 (2014)
10. Fokou, G., Jean, S., Hadjali, A., Baron, M.: Cooperative techniques for SPARQL query relaxation in RDF databases. In: *ESWC*. pp. 237–252 (2015)
11. Fokou, G., Jean, S., Hadjali, A., Baron, M.: Handling failing RDF queries: from diagnosis to relaxation. *Knowl. Inf. Syst.* **50**(1), 167–195 (2017)
12. Godfrey, P.: Minimization in cooperative response to failing database queries. *Int. J. Cooperative Inf. Syst.* **6**(2), 95–149 (1997)
13. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.* **3**(2-3), 158–182 (2005)
14. Hoffart, J., Suchanek, F.M., Berberich, K., Weikum, G.: YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.* **194**, 28–61 (2013)
15. Hogan, A., Mellotte, M., Powell, G., Stampouli, D.: Towards fuzzy query-relaxation for RDF. In: *ESWC*. pp. 687–702 (2012)
16. Huang, H., Liu, C., Zhou, X.: Approximating query answering on RDF databases. *World Wide Web* **15**(1), 89–114 (2012)
17. Hurtado, C.A., Poulouvasilis, A., Wood, P.T.: Query relaxation in RDF. *J. Data Semantics* **10**, 31–61 (2008)
18. Hurtado, C.A., Poulouvasilis, A., Wood, P.T.: Ranking approximate answers to semantic web queries. In: *ESWC*. pp. 263–277 (2009)
19. Jannach, D.: Fast computation of query relaxations for knowledge-based recommenders. *AI Commun.* **22**(4), 235–248 (2009)
20. Lanti, D., Rezk, M., Xiao, G., Calvanese, D.: The NPD benchmark: Reality check for OBDA systems. In: *EDBT*. pp. 617–628 (2015)



21. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web* **6**(2), 167–195 (2015)
22. Liffiton, M.H., Sakallah, K.A.: On finding all minimally unsatisfiable subformulas. In: SAT. pp. 173–186 (2005)
23. McSherry, D.: Retrieval failure and recovery in recommender systems. *Artif. Intell. Rev.* **24**(3-4), 319–338 (2005)
24. Murakami, K., Uno, T.: Efficient algorithms for dualizing large-scale hypergraphs. *Discrete Applied Mathematics* **170**, 83–94 (2014)
25. Muslea, I.: Machine learning for online query relaxation. In: KDD. pp. 246–255 (2004)
26. Muslea, I., Lee, T.J.: Online query relaxation via bayesian causal structures discovery. In: AAAI. pp. 831–836 (2005)
27. Reiter, R.: A theory of diagnosis from first principles. *Artif. Intell.* **32**(1), 57–95 (1987)
28. Stefanoni, G., Motik, B., Kostylev, E.V.: Estimating the cardinality of conjunctive queries over RDF data using graph summarisation. In: WWW. pp. 1043–1052 (2018)
29. Wang, M., Wang, R., Liu, J., Chen, Y., Zhang, L., Qi, G.: Towards empty answers in SPARQL: approximating querying with RDF embedding. In: ISWC. pp. 513–529 (2018)

## 7 Appendix

We report here the different DBpedia queries, range between 3 and 20 triple patterns, used to evaluate the different algorithms. The LUBM queries can be found in [goo.gl/1fZ8us](http://goo.gl/1fZ8us).

Star-shaped Queries	
Q1	SELECT * WHERE { ?Y1 <http://www.w3.org/2000/01/rdf-schema#label>"basketball league"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#subClassOf><http://dbpedia.org/ontology/SportsLeague> .?Y1 <http://www.w3.org/ns/prov#wasDerivedFrom><http://mappings.dbpedia.org/index.php/OntologyClass:AmericanFootballLeague> }
Q2	SELECT * WHERE { ?Y1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type><http://www.w3.org/2002/07/owl#Class> . ?Y1 <http://www.w3.org/2000/01/rdf-schema#subClassOf><http://dbpedia.org/ontology/SocietalEvent> . ?Y1 <http://www.w3.org/ns/prov#wasDerivedFrom><http://mappings.dbpedia.org/index.php/OntologyClass:GatedCommunity> . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> ?Y3 . ?Y1 <http://www.w3.org/2000/01/rdf-schema#comment> ?Y4 }
Q3	SELECT * WHERE { ?Y1 <http://www.w3.org/2000/01/rdf-schema#label>"australian football league"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#subPropertyOf><http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#hasCommonBoundary> . ?Y1 <http://www.w3.org/ns/prov#wasDerivedFrom> ?Y2 . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "vein"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#domain> ?Y4 . ?Y1 ?Y5 <http://dbpedia.org/ontology/AnatomicalStructure> . ?Y1 <http://www.w3.org/2002/07/owl#equivalentClass> <http://www.wikidata.org/entity/Q9609> . ?Y1 <http://www.w3.org/ns/prov#wasDerivedFrom> ?Y6 }
Q4	SELECT * WHERE { ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "(foto)model"@nl . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "relative"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#subPropertyOf><http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#sameSettingAs> . ?Y1 <http://www.w3.org/ns/prov#wasDerivedFrom> ?Y2 . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "vein"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#range> ?Y7 . ?Y1 <http://www.w3.org/2000/01/rdf-schema#domain> ?Y4 . ?Y1 ?Y5 <http://dbpedia.org/ontology/AnatomicalStructure> . ?Y1 <http://www.w3.org/2002/07/owl#equivalentClass> <http://www.wikidata.org/entity/Q9609> . ?Y1 <http://www.w3.org/ns/prov#wasDerivedFrom> ?Y6 }
Q5	SELECT * WHERE { ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "australian football league"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "Gottheit"@de . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "deity"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "forest"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "godheid"@nl . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "mine"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "dia"@ga . ?Y1 <http://www.w3.org/2000/01/rdf-schema#subClassOf><http://dbpedia.org/ontology/Agent> . ?Y1 <http://www.w3.org/2002/07/owl#equivalentClass> ?Y2 . ?Y1 <http://www.w3.org/ns/prov#wasDerivedFrom> ?Y3 . ?Y1 <http://www.w3.org/2000/01/rdf-schema#comment> ?Y4 . ?Y1 <http://www.w3.org/2002/07/owl#disjointWith> ?Y5 . Y1 <http://www.w3.org/2000/01/rdf-schema#comment> ?Y6 }

Q6	SELECT * WHERE { ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "australian football league"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "Gottheit"@de . ?Y1 <http://www.w3.org/2000/01/rdf- schema#label> "deity"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "for- est"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "godheid"@nl . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "mine"@en . ?Y1 <http://www.w3.org/2000/01/rdf- schema#label> "dia"@ga . ?Y1 <http://www.w3.org/2000/01/rdf-schema#subClassOf> <http://dbpedia.org/ontology/Agent> . ?Y1 <http://www.w3.org/2002/07/owl#equivalentClass> ?Y2 . ?Y1 <http://www.w3.org/ns/prov#wasDerivedFrom> ?Y3 . ?Y1 <http://www.w3.org/2000/01/rdf- schema#comment> ?Y4 . ?Y1 <http://www.w3.org/2002/07/owl#disjointWith> ?Y5 . ?Y1 <http://www.w3.org/2000/01/rdf-schema#subPropertyOf> ?Y6 . ?Y1 <http://www.w3.org/2000/01/rdf- schema#range> ?Y7 . ?Y1 <http://www.w3.org/2000/01/rdf-schema#domain> ?Y8 }
Q7	SELECT * WHERE { ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "australian foot- ball league"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "Gottheit"@de . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "deity"@en . ?Y1 <http://www.w3.org/2000/01/rdf- schema#label> "forest"@en . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "godheid"@nl . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "mine"@en . ?Y1 <http://www.w3.org/2000/01/rdf- schema#label> "dia"@ga . ?Y1 <http://www.w3.org/2000/01/rdf-schema#subClassOf> <http://dbpedia.org/ontology/Agent> . ?Y1 <http://www.w3.org/2002/07/owl#equivalentClass> ?Y2 . ?Y1 <http://www.w3.org/ns/prov#wasDerivedFrom> ?Y3 . ?Y1 <http://www.w3.org/2000/01/rdf- schema#comment> ?Y4 . ?Y1 <http://www.w3.org/2002/07/owl#disjointWith> ?Y5 . ?Y1 <http://www.w3.org/2000/01/rdf-schema#subPropertyOf> ?Y6 . ?Y1 <http://www.w3.org/2000/01/rdf- schema#range> ?Y7 . ?Y1 <http://www.w3.org/2000/01/rdf-schema#domain> ?Y8 . ?Y1 <http://www.w3.org/2002/07/owl#equivalentClass> <http://yago-knowledge.org/resource/ word- net_object_100002683 > . ?Y1 < http://dbpedia.org/ontology/owner>"Justin"@en . ?Y1 <http://www.w3.org/2002/07/owl#equivalentClass> <http://www.wikidata.org/entity/Q9609> . ?Y1 <http://www.w3.org/2000/01/rdf-schema#label> "forest"@en . ?Y1 <http://www.w3.org/2002/07/owl#disjointWith> <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#hasCommonBoundary> }

## Chain-shaped Queries

Q8	SELECT * WHERE { <http://dbpedia.org/resource/2Com> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y2 <http://www.w3.org/2002/07/owl#disjointWith> ?Y3 . ?Y3 <http://www.w3.org/2000/01/rdf- schema#comment> ?Y4 }
Q9	SELECT * WHERE { ?Y1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y2 <http://www.w3.org/2000/01/rdf- schema#subClassOf> ?Y3 . ?Y3 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y4 . ?Y4 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y5 . ?Y5 <http://www.w3.org/2002/07/owl#label> ?Y6 }

Q10	SELECT * WHERE { ?Y1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y2 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y3 . ?Y3 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y4 . ?Y4 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y5 . ?Y5 <http://www.w3.org/2002/07/owl#disjointWith> ?Y6 . ?Y6 <http://www.w3.org/2000/01/rdf-schema#comment> ?Y7 }
Q11	SELECT * WHERE { ?Y1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y2 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y3 . ?Y3 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y4 . ?Y4 <http://www.w3.org/2000/01/rdf-schema#comment> ?Y5 . ?Y5 <http://www.w3.org/2000/01/rdf-schema#label> ?Y6 . ?Y6 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y7 . ?Y8 <http://www.w3.org/2002/07/owl#equivalentClass> ?Y9 . ?Y9 <http://www.w3.org/2002/07/owl#disjointWith> ?Y10 }
Q12	SELECT * WHERE { ?Y1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y2 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y3 . ?Y3 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y4 . ?Y4 <http://www.w3.org/2000/01/rdf-schema#comment> ?Y5 . ?Y5 <http://www.w3.org/2000/01/rdf-schema#label> ?Y6 . ?Y6 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y7 . ?Y7 <http://www.w3.org/2002/07/owl#equivalentClass> ?Y8 . ?Y8 <http://www.w3.org/ns/prov#wasDerivedFrom> ?Y9 . ?Y9 <http://www.w3.org/2000/01/rdf-schema#domain> ?Y10 . ?Y10 <http://www.w3.org/2002/07/owl#ObjectProperty> ?Y11 }
Q13	SELECT * WHERE { <http://dbpedia.org/ontology/GrossDomesticProduct> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y2 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y3 . ?Y3 <http://www.w3.org/2000/01/rdf-schema#subClassOf> <http://dbpedia.org/resource/Advance-fee_scam> . <http://dbpedia.org/resource/Advance-fee_scam> <http://www.w3.org/2000/01/rdf-schema#comment> ?Y5 . ?Y5 <http://www.w3.org/2000/01/rdf-schema#label> ?Y6 . ?Y6 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y7 . ?Y7 <http://www.w3.org/2002/07/owl#equivalentClass> ?Y8 . ?Y8 <http://www.w3.org/ns/prov#wasDerivedFrom> ?Y9 . ?Y9 <http://www.w3.org/2000/01/rdf-schema#domain> ?Y10 . ?Y10 <http://www.w3.org/2002/07/owl#ObjectProperty> ?Y11 . ?Y11 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> . <http://dbpedia.org/ontology/Person> <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y13 . ?Y13 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://yago-knowledge.org/resource/wordnet_object_100002683> }

Q14	SELECT * WHERE { <http://dbpedia.org/resource/Hongsalmun> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y2 <http://www.w3.org/2000/01/rdf-schema#subClassOfQ> ?Y3 . ?Y3 <http://www.w3.org/ns/prov#wasDerivedFrom> <http://mappings.dbpedia.org/index.php/OntologyClass:Person> . ?Y4 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/resource/Advance-fee_scam> . <http://dbpedia.org/resource/Advance-fee_scam> <http://www.w3.org/2002/07/owl#disjointWith> ?Y6 . ?Y6 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y7 . ?Y7 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> . ?Y8 <http://www.w3.org/2000/01/rdf-schema#label> ?Y9 . ?Y9 ?YP9 <http://dbpedia.org/ontology/distanceToEdinburgh> . ?Y10 <http://www.w3.org/2000/01/rdf-schema#label> "(foto)moduLoo"nl . ?Y11 <http://www.w3.org/2002/07/owl#equivalentClass> ?Y12 . ?Y12 <http://www.w3.org/2002/07/owl#equivalentClass> ?Y13 . ?Y13 <http://www.w3.org/2002/07/owl#disjointWith> ?Y14 . ?Y14 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Place> . ?Y15 <http://www.w3.org/2000/01/rdf-schema#label> ?Y16 }
Q15	SELECT * WHERE { <http://dbpedia.org/resource/Hongsalmun> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y2 <http://www.w3.org/2000/01/rdf-schema#subClassOfQ> ?Y3 . ?Y3 <http://www.w3.org/ns/prov#wasDerivedFrom> <http://mappings.dbpedia.org/index.php/OntologyClass:Person> . ?Y4 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/resource/Advance-fee_scam> . <http://dbpedia.org/resource/Advance-fee_scam> <http://www.w3.org/2002/07/owl#disjointWith> ?Y6 . ?Y6 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y7 . ?Y7 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person> . ?Y8 <http://www.w3.org/2000/01/rdf-schema#label> ?Y9 . ?Y9 ?YP9 <http://dbpedia.org/ontology/distanceToEdinburgh> . ?Y10 <http://www.w3.org/2000/01/rdf-schema#label> "(foto)moduLoo"nl . ?Y11 <http://www.w3.org/2002/07/owl#equivalentClass> ?Y12 . ?Y12 <http://www.w3.org/2002/07/owl#equivalentClass> ?Y13 . ?Y13 <http://www.w3.org/2002/07/owl#disjointWith> ?Y14 . ?Y14 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Place> . ?Y15 <http://www.w3.org/2000/01/rdf-schema#label> "Justin"en . "Justin"@en <http://dbpedia.org/ontology/owner> ?Y17 . ?Y17 <http://www.w3.org/2002/07/owl#equivalentClass> ?Y18 . ?Y18 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y18 . ?Y18 <http://www.w3.org/2002/07/owl#disjointWith> ?Y19 . ?Y19 <http://www.w3.org/ns/prov#wasDerivedFrom> <http://mappings.dbpedia.org/index.php/OntologyClass:AmericanFootballLeague> }

Composite-shaped Queries	
Q16	SELECT * WHERE { ?Y1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#subClass> ?Y2 . ?Y4 <http://www.w3.org/1999/02/22-rdf-syntax-ns#subClass> ?Y3 }
Q17	SELECT * WHERE { <http://dbpedia.org/ontology/GrossDomesticProduct> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . <http://dbpedia.org/resource/Ilya_Mokretsov> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y3 . ?Y3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#subClass> ?Y4 . <http://dbpedia.org/ontology/Planet> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y4 }

Q18	SELECT * WHERE { <http://dbpedia.org/ontology/GrossDomesticProduct> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . <http://dbpedia.org/resource/Preston_is_My_Paris> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y3 . ?Y3 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y4 . ?Y5 <http://www.w3.org/2000/01/rdf-schema#range> ?Y4 . <http://dbpedia.org/resource/Hongsalmun> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y4 . <http://dbpedia.org/resource/Wrong_scale> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y4 . ?Y4 <http://www.w3.org/2000/01/rdf-schema#label> "company"@en }
Q19	SELECT * WHERE { <http://dbpedia.org/ontology/GrossDomesticProduct> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . <http://dbpedia.org/resource/Preston_is_My_Paris> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y3 . ?Y3 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y4 . ?Y5 <http://www.w3.org/2000/01/rdf-schema#range> ?Y4 . <http://dbpedia.org/resource/Hongsalmun> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y4 . <http://dbpedia.org/resource/Scoville_scale> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y4 . ?Y4 <http://www.w3.org/2000/01/rdf-schema#label> "company"@en . ?Y4 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.w3.org/2002/07/owl#Class> . <http://dbpedia.org/ontology/Company> <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y4 }
Q20	SELECT * WHERE { <http://dbpedia.org/ontology/GrossDomesticProduct> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . ?Y3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y2 . <http://dbpedia.org/resource/Preston_is_My_Paris> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y3 . ?Y3 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y4 . ?Y5 <http://www.w3.org/2000/01/rdf-schema#range> ?Y4 . <http://dbpedia.org/resource/Hongsalmun> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y4 . <http://dbpedia.org/resource/Scoville_scale> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?Y4 . ?Y4 . ?Y4 <http://www.w3.org/2000/01/rdf-schema#label> "companies"@en . ?Y4 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.w3.org/2002/07/owl#Class> . ?Y9 <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?Y4 . <http://dbpedia.org/resource/Connections_per_circuit_hour> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Country> . ?Y10 <http://www.w3.org/2000/01/rdf-schema#label> ?Y11 . ?11 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Placet> }

Q21	<pre> SELECT      *      WHERE      {      &lt;http://dbpedia.org/ontology/GrossDomesticProduct&gt; &lt;http://www.w3.org/1999/02/22-rdf-syntax-ns#type&gt; ?Y2 . ?Y3 &lt;http://www.w3.org/1999/02/22- rdf-syntax-ns#type&gt; ?Y2 . &lt;http://dbpedia.org/resource/Preston_is_My_Paris&gt; &lt;http://www.w3.org/1999/02/22-rdf-syntax-ns#type&gt; ?Y3 . ?Y3 &lt;http://www.w3.org/2000/01/rdf- schema#subClassOf&gt; ?Y4 . ?Y5 &lt;http://www.w3.org/2000/01/rdf-schema#range&gt; ?Y4 . &lt;http://dbpedia.org/resource/Hongsalmun&gt; &lt;http://www.w3.org/1999/02/22-rdf-syntax- ns#type&gt; ?Y4 . &lt;http://dbpedia.org/resource/Scoville_scale&gt; &lt;http://www.w3.org/1999/02/22- rdf-syntax-ns#type&gt; ?Y4 . ?Y4 &lt;http://www.w3.org/2000/01/rdf-schema#label&gt; "companies"@en . ?Y4 &lt;http://www.w3.org/1999/02/22-rdf-syntax-ns#type&gt; &lt;http://www.w3.org/2002/07/owl#Class&gt; . ?Y9 &lt;http://www.w3.org/2000/01/rdf- schema#subClassOf&gt; ?Y4 . &lt;http://dbpedia.org/resource/Connections_per_circuit_hour&gt; &lt;http://www.w3.org/1999/02/22-rdf-syntax-ns#type&gt; &lt;http://dbpedia.org/ontology/Country&gt; . ?Y10 &lt;http://www.w3.org/2000/01/rdf-schema#label&gt; ?Y11 . ?11 &lt;http://www.w3.org/1999/02/22- rdf-syntax-ns#type&gt; &lt;http://dbpedia.org/ontology/Placet&gt; . &lt;http://dbpedia.org/ontology/Placet&gt; &lt;http://www.w3.org/2000/01/rdf-schema#subPropertyOf&gt; ?Y12 . ?Y12 &lt;http://www.w3.org/2000/01/rdf-schema#range&gt; ?Y13 } </pre>
-----	--