A Diagnosis and Repair Framework for *DL-Lite*$_\mathcal{A}$ KBs

Michalis Chortis, Giorgos Flouris

ICS-FORTH, Greece
{mhortis,fgeo}@ics.forth.gr

Abstract. Several logical formalisms have been proposed in the literature for expressing structural and semantic integrity constraints of Linked Open Data (LOD). Still, the integrity of the datasets published in the LOD cloud needs to be improved, as published data often violate such constraints, jeopardising the value of applications consuming linked data in an automatic way. In this work, we propose a novel, fully automatic framework for detecting and repairing violations of integrity constraints, by considering both explicit and implicit ontological knowledge. Our framework relies on the ontology language *DL-Lite*$_\mathcal{A}$ for expressing several useful types of constraints, while maintaining good computational properties. The experimental evaluation shows that our framework is scalable for large datasets and numbers of invalidities exhibited in reality by reference linked datasets (e.g., DBpedia).

Keywords: Repairing, diagnosis, *DL-Lite*$_\mathcal{A}$, integrity constraints

1 Introduction

Linked Open Data (LOD) published on the Web of Data are often associated with various structural (e.g., primary key) and semantic (e.g., disjointness) integrity constraints. These constraints are usually expressed in ontological [19, 22] or database [10] logic frameworks. However, LOD sources do not impose such constraints a priori, when data are created, so violations of integrity constraints must be detected and repaired a posteriori. As have been reported in [15], reference LOD sources, such as DBpedia[1] or LinkedGeoData[2], exhibit millions of violations (this is also verified by our own experiments – see Table 3).

In most of the cases, LOD are manually repaired by their curators or by their consuming applications, using, at best, diagnosis approaches or tools (e.g., [16, 19, 22], Stardog[3], QuOnto [1] etc.) for detecting violations of various types of integrity constraints. Obviously, the manual repair of millions of violations is a time-consuming and error-prone task, a fact that seriously limits the data quality of the available LOD sources. Thus, a major challenge is to automatically

[1] http://dbpedia.org
[2] http://linkedgeodata.org
[3] http://stardog.com/

detect and repair violations of both structural and semantic integrity constraints, especially when ontology reasoning is involved (i.e., detect and repair violations of constraints like disjointness, functional constraints etc., taking into account logical inference and its interaction with those constraints).

In this work, we propose a novel *automatic framework* for assisting curators in the arduous task of enforcing integrity constraints in large datasets. We provide an efficient methodology for detecting invalidities (*diagnosis*), as well as for automatically resolving them (*repairing*), in a manner that has minimal impact in terms of lost knowledge on the Knowledge Base (KB), according to the principles set out in earlier works [2, 8].

We consider detecting and repairing of invalidities attributed to constraints of a purely logical nature (e.g., class disjointness). Constraints are expressed in the language *DL-Lite*$_{\mathcal{A}}$ [4], which belongs to the *DL-Lite* family of ontology languages that forms the foundation of the popular *OWL 2 QL*[4] language. The choice of *DL-Lite*$_{\mathcal{A}}$ was motivated by the fact that it is arguably rich enough to capture several useful types of integrity constraints that are used in practice in LOD datasets, and their interaction with implicit knowledge, while at the same time supporting efficient query answering [4].

The main contributions of our work are the following:

– We propose a framework for detecting and automatically repairing invalidities, for constraints that are expressed in *DL-Lite*$_{\mathcal{A}}$, namely: concept/property disjointness constraints, property domain/range disjointness constraints and functional constraints. Diagnosis of invalidities related to both explicit and inferred constraints can be performed in linear time with respect to the dataset size, whereas repairing can be performed in polynomial time with respect to the number of invalidities.
– We have implemented an operational repairing system for real-world applications. Our implementation is modular, allowing each component to be implemented in a manner independent to the other components. This way, we managed to reuse off-the-shelf, state-of-the-art tools for many of the components, such as reasoning, storage, query answering, etc.
– We have experimentally evaluated the scalability and performance of our algorithms, using real and synthetic datasets. The main conclusion drawn is that our framework can scale for very large datasets, such as DBpedia, as well as for large numbers (millions) of invalidities.

The rest of the paper is structured as follows: in Section 2, we motivate the use of the *DL-Lite*$_{\mathcal{A}}$ language for this problem and explain its features; in Section 3, we describe our framework and explain how we address the problems of detecting and resolving invalidities; Section 4 describes our algorithms for diagnosis and repairing; in Section 5, we describe our experimental evaluation and report on the main conclusions drawn; finally, Section 6 compares our contributions to the related work and Section 7 concludes.

[4] `http://www.w3.org/TR/owl2-profiles/#OWL_2_QL`

2 Preliminaries

In *DL-Lite*$_{\mathcal{A}}$ [4], concept expressions, hereafter expressed by the letter C, and role expressions, denoting binary relations between concepts and hereafter expressed by the letter R, are formed according to the following syntax, where A denotes an atomic concept and P denotes an atomic role:

$$C \longrightarrow A \mid \exists R \qquad R \longrightarrow P \mid P^{-}$$

A *DL-Lite*$_{\mathcal{A}}$ TBox consists of axioms of the following form:

$$C_1 \sqsubseteq C_2 \qquad C_1 \sqsubseteq \neg C_2 \qquad R_1 \sqsubseteq R_2 \qquad R_1 \sqsubseteq \neg R_2 \qquad (\text{funct } R)$$

A *DL-Lite*$_{\mathcal{A}}$ ABox is a finite set of assertions of the following form:

$$A(x) \qquad P(x,y)$$

In order to guarantee good complexity results for reasoning tasks like consistency checking, *DL-Lite*$_{\mathcal{A}}$ imposes a limitation in the TBox, namely that a functional role cannot be specialized by using it in the right-hand side of a role inclusion assertion. This means that if a *DL-Lite*$_{\mathcal{A}}$ TBox contains an axiom of the form $R' \sqsubseteq R$, then it cannot contain (funct R) or (funct R^{-}) [5]. Note that *DL-Lite*$_{\mathcal{A}}$ assertions can be also expressed in OWL syntax.

DL-Lite$_{\mathcal{A}}$ follows the standard reasoning semantics of DLs [4–6]. A *DL-Lite*$_{\mathcal{A}}$ KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ is called *inconsistent* iff $\mathcal{T} \cup \mathcal{A}$ is inconsistent (in the standard logical sense). It is called *consistent* otherwise.

With respect to performance, *DL-Lite*$_{\mathcal{A}}$ has the important property of *FOL-Reducibility* [5], which essentially means that one can reduce the process of inconsistency checking and query answering to the evaluation of First-Order Logic (FOL) queries over the ABox, considered as a database; this makes both tasks tractable (in LogSpace with respect to the data) [5].

3 Diagnosis and Repair

3.1 Constraints in *DL-Lite*$_{\mathcal{A}}$

For the purposes of diagnosis and repair, we can distinguish three different types of *DL-Lite*$_{\mathcal{A}}$ TBox axioms, namely *positive inclusions* (of the form $C_1 \sqsubseteq C_2$, $R_1 \sqsubseteq R_2$), *negative inclusions* (of the form $C_1 \sqsubseteq \neg C_2$, $R_1 \sqsubseteq \neg R_2$) and *functionality assertions* (of the form funct R). This distinction is important for diagnosis and repair due to the fact that the ABox is viewed under the Open World Assumption (OWA), which is considered for Description Logics and the ontology languages of the Semantic Web in general (such as OWL – but see [22] for an effort to understand OWL under the Closed World Assumption, and the NRL language[5] for a similar analysis). Due to the OWA, a TBox consisting of positive inclusions only can never lead to an inconsistent KB; therefore, the only interesting (from the diagnosis perspective) constraints are the negative inclusions and the functionality assertions. In the following, the term *constraint* will be used to refer to negative TBox inclusions and functionality assertions.

[5] `http://www.semanticdesktop.org/ontologies/2007/08/15/nrl`

Despite that, positive inclusions are still relevant for the diagnosis process, because they may generate inferred information that should be taken into account. As an example, assume that the TBox contains the constraint $A_1 \sqsubseteq \neg A_3$ and the axiom $A_2 \sqsubseteq A_3$ (where A_1, A_2, A_3 are atomic concepts), and suppose that the ABox contains both $A_1(x)$ and $A_2(x)$ for some x. Even though no constraint is explicitly violated, the combination of the ABox contents with the aforementioned TBox would lead to inferring both $A_3(x)$ and $\neg A_3(x)$, i.e., an invalidity. Note that the positive inclusion $A_2 \sqsubseteq A_3$, albeit not violated itself, plays a critical role in creating this invalidity.

Rather than capturing such invalidities via the obvious method of computing the closure of the ABox, it is more efficient to identify the constraints implied by the explicitly declared constraints and the positive inclusions in the TBox. In our example, we could identify that the constraint $A_1 \sqsubseteq \neg A_2$ is a consequence of the two explicit axioms in the TBox, so the presence of $A_1(x)$ and $A_2(x)$ violates this implicit constraint.

This process amounts to computing all explicit and implicit constraints of the TBox (denoted by $cln(\mathcal{T})$) [5], i.e., the set of all the functionality assertions and the explicit and implicit negative inclusions present in the TBox. In fact, it has been proven that, in order to check the consistency of a $DL\text{-}Lite_{\mathcal{A}}$ KB, one has to take into account only the constraints in $cln(\mathcal{T})$ [6]. More formally, a $DL\text{-}Lite_{\mathcal{A}}$ KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ is inconsistent iff there is a constraint $c \in cln(\mathcal{T})$ and a pair of assertions $a_1, a_2 \in \mathcal{A}$ such that the $DL\text{-}Lite_{\mathcal{A}}$ KB $\mathcal{K}' = \langle \{c\}, \{a_1, a_2\} \rangle$ is inconsistent [6]. In the following, the triple (a_1, a_2, c) will be called an *invalidity* of $\mathcal{K}$. It is obvious by the above result that in order to render a KB consistent, for each invalidity (a_1, a_2, c), one of a_1, a_2 has to be removed from the ABox.

Example 1. Consider the following $DL\text{-}Lite_{\mathcal{A}}$ KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$:

$$\mathcal{T} = \{(\text{funct } P_1),\ A_1 \sqsubseteq \neg A_2,\ \exists P_2 \sqsubseteq A_1\}$$
$$\mathcal{A} = \{A_1(x_1), A_2(x_1), P_2(x_1, y_1), P_1(x_3, y_2), P_1(x_3, y_3), P_1(x_3, y_4)\}$$

The closure of negative inclusions and functionality assertions of $\mathcal{T}$ ($cln(\mathcal{T})$), computed in the way that was presented in [6], is the following:

$$cln(\mathcal{T}) = \{(\text{funct } P_1),\ A_1 \sqsubseteq \neg A_2,\ \exists P_2 \sqsubseteq \neg A_2\}$$

From the computed closure, we can easily deduce that $(A_1(x_1), A_2(x_1), A_1 \sqsubseteq \neg A_2)$ is one of the invalidities in the KB. □

3.2 Approach for Diagnosis and Repair

Diagnosis amounts to identifying the invalidities, i.e., the data assertions and the (possibly implicit) constraint that are involved in an invalidity. Using the property of FOL-Reducibility, the identification of invalidities in a $DL\text{-}Lite_{\mathcal{A}}$ KB can be reduced to the execution of adequately defined FOL queries over a database [5] – see also Table 1. Exploiting this property, diagnosis is performed by simply executing the queries corresponding to the constraints in $cln(\mathcal{T})$, to get all the invalidities of the KB under question.

Repairing is based on the aforementioned property that restoring consistency requires eliminating all invalidities from a KB via removing either one of the two data assertions that take part in each invalidity; formally:

Definition 1. *Given a DL-Lite*$_{\mathcal{A}}$ *KB* $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ *a* repairing delta *of* $\mathcal{K}$ *is a selection of data assertions RD, such that* $\mathcal{K}' = \langle \mathcal{T}, \mathcal{A} \setminus RD \rangle$ *is consistent. A repairing delta RD is called* minimal *iff there is no repairing delta* RD'*, such that* $RD' \subset RD$*.* □

The notion of minimality is important, as many authors have proposed the identification of *minimal* repairing deltas (under different forms of minimality) as one of the main concerns during repairing [2, 8]; as is obvious by Definition 1, minimal repairing deltas correspond to *subset repairs* in the terminology of [2].

Identifying the minimal repairing delta(s) is not trivial. The computation of such delta(s) is based on the fact that constraints expressed in *DL-Lite*$_{\mathcal{A}}$ allow the presence of interrelated invalidities, i.e., data assertions being involved in more than one invalidities. This implies that potential resolutions of such invalidities coincide, and that there exist resolutions which resolve more than one invalidity at the same time.

To help in the process of identifying the minimal repairing delta, the diagnosed invalidities are organized into an *interdependency graph*, which is used to identify assertions involved in multiple invalidities. Formally:

Definition 2. *The interdependency graph of a DL-Lite*$_{\mathcal{A}}$ *KB* $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ *is an undirected labelled graph* $IG(\mathcal{K}) = (V, E)$ *such that* $V = \{a \mid (a_1, a_2, c)$ *is an invalidity of* $\mathcal{K}$ *and* $a = a_1$ *or* $a = a_2\}$ *and* $E = \{(a_1, a_2, c) \mid (a_1, a_2, c)$ *is an invalidity of* $\mathcal{K}\}$*.* □

The use of the interdependency graph as a structure to represent the invalidities that are diagnosed in the KB gives the ability to get a better grasp of the form and complexity of the invalidities and their interrelationships, as well as to use methods and tools that come from graph theory in order to facilitate the repairing process. Note that an interdependency graph is different from a conflict-graph [9], as the interdependency graph does not contain every assertion in the ABox, having an obvious impact in the algorithm time-cost.

In terms of the interdependency graph, resolving an invalidity amounts to removing one of the two vertices that are connected by the edge representing this invalidity. Therefore, a minimal repairing delta is essentially the minimal *vertex cover* of the corresponding interdependency graph, which reduces the problem of repairing to the well-known problem of VERTEX COVER [11]. This fact forms the basis of our algorithms presented in the next section.

4 Algorithms for Diagnosis and Repairing

4.1 Diagnosis Algorithm

The diagnosis algorithm is used to detect all the invalidities in a KB, and provide them as output in the form of an interdependency graph. The steps needed to perform diagnosis are illustrated in Algorithm 1.

Algorithm 1 Diagnosis($\mathcal{K}$)

```
Input: A DL-Lite_A KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$
Output: The interdependency graph of $\mathcal{K}$, $IG(\mathcal{K}) = (V, E)$
 1: $V, E \leftarrow \emptyset$
 2: Compute the $cln(\mathcal{T})$
 3: for all $c \in cln(\mathcal{T})$ do
 4:     $q_c \leftarrow \delta(c)$
 5:     $Ans_{q_c} \leftarrow q_c^{\mathcal{A}}$
 6:     for all $\langle a_1, a_2 \rangle \in Ans_{q_c}$ do
 7:         $V \leftarrow V \cup \{a_1, a_2\}$
 8:         $E \leftarrow E \cup \{(a_1, a_2, c)\}$
 9:     end for
10: end for
11: return $IG(\mathcal{K}) = (V, E)$
```

The diagnosis algorithm starts by computing the closure $cln(\mathcal{T})$ of negative inclusions and functionality assertions of the TBox (line 2 of Algorithm 1), in order to get the full set of constraints that need to be checked over the ABox. Each of the constraints in $cln(\mathcal{T})$ is then transformed to a FOL query (line 4) using predefined patterns, as defined in Table 1 (see also [5]), whose answers determine the invalidities. These queries are executed over the ABox in line 5 (Ans_{q_c} contains pairs $\langle a_1, a_2 \rangle$ such that (a_1, a_2, c) is an invalidity). Note that these FOL queries can be easily expressed as SPARQL queries over an ABox stored in a triple store, so that off-the-shelf, optimized tools can be used for query answering. The last step of the algorithm encodes the invalidities in the form of an interdependency graph (lines 6-9) as specified in Definition 2.

Constraint (c)	**Transformation** ($\delta(c)$)
$c = A_1 \sqsubseteq \neg A_2$	$\delta(c) = q(x) \leftarrow A_1(x), A_2(x)$
$c = A_1 \sqsubseteq \neg\exists P_1$ (or $c = \exists P_1 \sqsubseteq \neg A_1$)	$\delta(c) = q(x) \leftarrow A_1(x), P_1(x, y)$
$c = A_1 \sqsubseteq \neg\exists P_1^-$ (or $c = \exists P_1^- \sqsubseteq \neg A_1$)	$\delta(c) = q(x) \leftarrow A_1(x), P_1(y, x)$
$c = \exists P_1 \sqsubseteq \neg\exists P_2$	$\delta(c) = q(x) \leftarrow P_1(x, y_1), P_2(x, y_2)$
$c = \exists P_1^- \sqsubseteq \neg\exists P_2^-$	$\delta(c) = q(x) \leftarrow P_1(y_1, x), P_2(y_2, x)$
$c = \exists P_1 \sqsubseteq \neg\exists P_2^-$	$\delta(c) = q(x) \leftarrow P_1(x, y_1), P_2(y_2, x)$
$c = P_1 \sqsubseteq \neg P_2$ (or $c = P_1^- \sqsubseteq \neg P_2^-$)	$\delta(c) = q(x, y) \leftarrow P_1(x, y), P_2(x, y)$
$c = P_1 \sqsubseteq \neg P_2^-$	$\delta(c) = q(x, y) \leftarrow P_1(x, y), P_2(y, x)$
$c =$(funct P)	$\delta(c) = q(x) \leftarrow P(x, y_1), P(x, y_2)$
$c =$(funct P^-)	$\delta(c) = q(x) \leftarrow P(y_1, x), P(y_2, x)$

Table 1: Transformation of *DL-Lite*$_\mathcal{A}$ constraints to FOL queries.

The following example illustrates the diagnosis algorithm in action:

Example 2. Consider the KB $\mathcal{K}$ and the $cln(\mathcal{T})$ of Example 1. The corresponding FOL queries to check for invalidities, according to Table 1 are:

$$\begin{aligned}
q_1(x) &\leftarrow P_1(x,y) \wedge P_1(x,z) \wedge y \neq z\\
q_2(x) &\leftarrow A_1(x) \wedge A_2(x)\\
q_3(x) &\leftarrow P_2(x,y) \wedge A_2(x)
\end{aligned}$$

From the execution of the above three queries over the ABox of Example 1, we get the following answers (each of which corresponds to an invalidity):

$$\begin{aligned}
Ans_{q_1} =& \{\langle P_1(x_3,y_2), P_1(x_3,y_3)\rangle,\\
& \langle P_1(x_3,y_2), P_1(x_3,y_4)\rangle,\\
& \langle P_1(x_3,y_3), P_1(x_3,y_4)\rangle\}\\
Ans_{q_2} =& \{\langle A_1(x_1), A_2(x_1)\rangle\}\\
Ans_{q_3} =& \{\langle A_2(x_1), P_2(x_1,y_1)\rangle\}
\end{aligned}$$

Figure 1 shows the corresponding interdependency graph. □

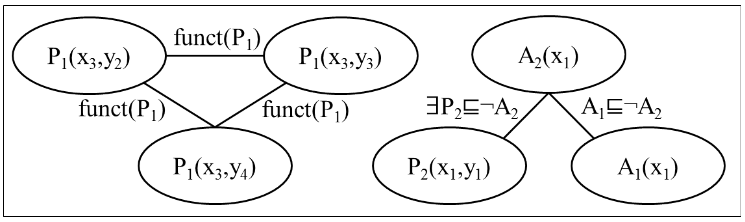


Fig. 1: Example of an interdependency graph.

As already mentioned, computing $cln(\mathcal{T})$ (line 2) is in LogSpace with respect to the data [5], whereas the remaining steps of the algorithm are linear with respect to the invalidities and the constraints in $cln(\mathcal{T})$.

4.2 Repairing Algorithm

The repairing algorithm (Algorithm 2) takes as input the interdependency graph and is responsible for automatically repairing the KB. As explained in Section 3.2, the main idea behind the repairing algorithm is the computation of the vertex cover of the interdependency graph.

To do so, the repairing algorithm first breaks the interdependency graph $IG(\mathcal{K})$ into the set of its connected components (line 2). Note that the computation of the vertex cover for each of the connected components is independent to the others, and can be parallelized for better performance.

This computation (vertex cover) is performed in lines 3-5. Recall that VERTEX COVER is a well-known NP-COMPLETE problem [20], but many approximation algorithms have been proposed, such as the *2-approximation* algorithm [20], or the approximation algorithm presented in [14].

Algorithm 2 Repair($IG(\mathcal{K}), \mathcal{A}$)

```
Input: An interdependency graph $IG(\mathcal{K})$ and a DL-Lite$_{\mathcal{A}}$ ABox $\mathcal{A}$
Output: $\mathcal{K}$ in a consistent state
1: $repairing\_delta \leftarrow \emptyset$
2: $CC \leftarrow ConnectedComponents(IG(\mathcal{K}))$
3: for all $cc \in CC$ do
4:     $repairing\_delta \leftarrow repairing\_delta \cup GreedyVertexCover(cc)$
5: end for
6: $\mathcal{A} \leftarrow \mathcal{A} \setminus repairing\_delta$
```

For our implementation and experiments below, we chose (for efficiency) to compute the vertex cover in a greedy manner, as presented in [20] (but any other algorithm for VERTEX COVER could be used instead). Greedy means that, in each step of the computation, the vertex that is chosen to be included in the cover is the vertex with the highest degree (in other words, the invalid data assertion that is part of the most invalidities). If there exist more than one vertices with the same degree, one of those vertices is arbitrarily chosen; this arbitrary choice avoids the need for complex and time-consuming selection conditions and guarantees that a single vertex cover is returned by the algorithm. This computation is performed in the *GreedyVertexCover* subroutine, which is omitted for brevity.

The output of lines 3-5 (*repairing_delta*) contains the data assertions to be removed from the dataset in order to render it valid. The actual repairing is performed in line 6, through a single SPARQL-Update statement[6] requesting the deletion of all the assertions in the repairing delta.

The correctness of our algorithms is guaranteed by our analysis in Section 3 and the results in [6]. The computational complexity of Algorithm 2 is dominated by the computation of the vertex cover, which is proven to achieve $O(\log n)$ approximation of the optimal solution (where n is the number of vertices of the graph), with a time complexity of $O(n \log n)$ [20].

The following concludes the running example for our framework:

Example 3. Consider the interdependency graph of Figure 1. The repairing algorithm will compute the following repairing delta:

$$repairing\_delta = \{A_2(x_1), P_1(x_3, y_2), P_1(x_3, y_3)\}$$

After the application of the repairing delta, the ABox $\mathcal{A}$ is in the following state:

$$\mathcal{A} = \{A_1(x_1), P_2(x_1, y_1), P_1(x_3, y_4)\}$$

which can be easily verified to be a consistent KB with respect to $\mathcal{T}$. □

[6] `http://www.w3.org/TR/2013/REC-sparql11-update-20130321/`

5 Experimental Evaluation

5.1 Overview of Experimental Evaluation

We have implemented our framework as a Java web application. More specifically, we have created a system that uses a triple store, which lies on a Virtuoso Open-Source Edition Server[7] version 07.10, as a storage for the ABox instances and as an endpoint for query answering. For storing the set of constraints, we used a main memory model, which, along with the communication with the Virtuoso Server, are handled by the Apache Jena[8] framework. Apache Jena is also used for the various reasoning tasks (e.g., computation of $cln(\mathcal{T})$). The system used for the experiments was an AMD Opteron 3280, 8-core CPU with 24GB RAM (we allocated 8GB for the JVM), running Ubuntu Server 12.04.

We have performed several experiments in order to measure the performance and scalability of our framework, as well as to determine the decisive factors for the performance of the different phases of the process. More specifically, we performed three sets of experiments: (*i*) the first set verified that our framework can handle millions of violations in real-world ABoxes that scale up to more than 2 billion triples, considering hundreds of thousands of constraints; (*ii*) the second set of experiments quantified the impact of ABox size on performance, by using real Tboxes with constraints and synthetic ABoxes of varying sizes; and, (*iii*) the third set quantified the impact of the number of invalid data assertions on performance, by using real TBoxes with constraints and synthetic ABoxes with varying number of invalid data assertions.

In all of the above sets of experiments, we measured the time needed to run the diagnosis algorithm and produce the interdependency graph (*diagnosis time*), the time needed by the repairing algorithm to compute the repairing delta (*repair computation time*) and the time needed to apply this repairing delta on the dataset, using a SPARQL-Update query (*repair application time*). All of our experiments were run in sets of 5 hot runs and the average times were taken.

5.2 Real and Synthetic Datasets Used

For the TBox, we used two versions (3.6, 3.9) of the DBpedia ontology, which is a reference dataset for LOD, already containing different amounts and types of constraints; this is illustrated in Table 2, which shows information on how many functional and (concept/domain/range) disjointness constraints exist in the original TBox, as well as how many of these exist in the closure of negative inclusions ($cln(\mathcal{T})$)[9], and how many queries need to be executed for diagnosis. Property disjointness is the only type of constraint supported by $DL\text{-}Lite_{\mathcal{A}}$ that

[7] `http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/`
[8] `http://jena.apache.org/`
[9] The big difference in the amount of disjointness constraints between the original DBpedia 3.9 and its closure is caused by the many positive inclusions and their interaction with the negative inclusions during the computation of the closure.

TBox version	**Constraints**		**Constraints in $cln(\mathcal{T})$**		**Queries**
	Functional	Disjointness	Functional	Disjointness	
DBpedia 3.6	18	0	18	0	18
DBpedia 3.9	26	17	26	323.389	323.415

Table 2: Constraints in DBpedia versions 3.6 and 3.9.

was not considered in our experiments, because we were unable to find any real TBoxes with this constraint (so it seems irrelevant for practical applications).

For running our experiments, we used the above TBoxes, together with both real and synthetic ABoxes. Real ABoxes were used to evaluate our system in realistic conditions, whereas synthetic ABoxes allow controlling the important factors for the performance of our algorithm, such as size and number of invalidities, and the appropriate evaluation of their effect on performance.

Real ABoxes were taken by the two DBpedia versions corresponding to the two aforementioned TBoxes, stored in a local Virtuoso instance. The DBpedia 3.6 ABox contains around 541 million triples, whereas the DBpedia 3.9 ABox contains more than 2 billion triples.

To generate synthetic ABoxes, we started from each TBox and an empty ABox, and added data and property instances of the classes/properties of the corresponding TBox, making sure to include some invalid pairs of assertions as well (taking into account the constraints). For the first set of generated ABoxes we created a fixed number of invalid data assertions (10K) and a varying ABox size (500K-5M triples, with a step of 500K triples). The second set of ABoxes had a fixed size (10M triples) and a varying number of invalid data assertions (50K-500K, with a step of 50K). The above two sets of ABoxes were used in the second and third set of experiments respectively.

5.3 Scalability and Performance Evaluation

The first set of experiments aimed at verifying the scalability of our framework in real-world settings, with ABoxes of billions of triples and with large numbers of constraints (up to hundreds of thousands). For this purpose, we used DBpedia versions 3.6 and 3.9 (TBox and real ABox). For each version, we measured the diagnosis time (identifying invalidities and creating the interdependency graph), the repair computation time (computing the repairing delta), the repair application time (applying the repairing delta) and the total time (sum of the above). We also measured the number of invalid data assertions that appear in the datasets, to see how well our framework scales with respect to that, as well as the size of the repairing delta, to verify that a manual repair by the curator would be infeasible in this context.

The results of this set of experiments are illustrated in Table 3. In the table, IDA denotes the number of invalid data assertions, Delta is the size of the repairing delta (in triples), t_d denotes the diagnosis time, $t_{r.c.}$ the repair computation

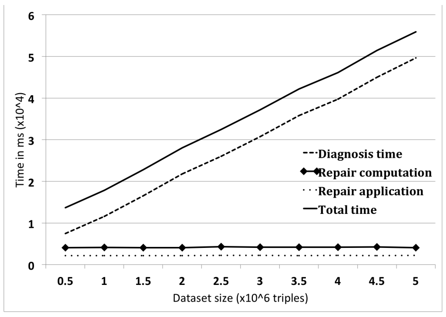
Version	Triples	IDA	Delta	t_d	$t_{r.c.}$	$t_{r.a.}$	t_t
DBpedia 3.6	541M	1.109	749	2.440	402	219	3.061
DBpedia 3.9	>2B	1.020.199	717.798	9.610.319	27.190.191	1.415.329	38.215.839

Table 3: Experiments performed on real datasets.

time, $t_{r.a.}$ the repair application time and t_t the total time needed for diagnosis and repairing. All times are in milliseconds.

The results show that our framework is scalable, for both large datasets and big numbers of invalid data assertions, and that it can be applied in real-world settings. It also proves that already deployed and massively used reference KBs, such as DBpedia, don't have sufficient mechanisms for preventing the introduction of invalid data or for detecting and repairing such invalid data. Moreover, our experiments illustrate that the number of invalid data assertions and the size of the repairing delta would be prohibitive for manual repairing.

Our second set of experiments evaluated the effect of ABox size on performance using synthetic ABoxes of varying sizes and a fixed number of invalid data assertions. The results of this set of experiments appear in Figure 2. Note that some of the curves in the graphs are difficult to distinguish, either because they are too close to the start of the x-axis (e.g., the repair computation time and the repair application time in the left figure), or because they are too close with another curve (e.g., the diagnosis time and the total time in the right figure).



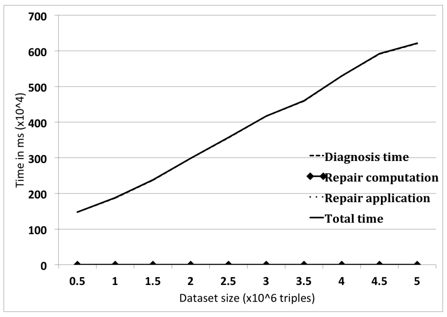


Fig. 2: Performance for DBpedia 3.6 (left) and 3.9 (right) with 10K invalidities.

From the results of this set of experiments, we conclude that diagnosis time grows linearly with respect to the ABox size and that it is the dominating factor of the total time, when the number of invalid data assertions is fixed. This is an important conclusion because it shows that, overall, our framework scales linearly with respect to the dataset size.

The third set of experiments evaluated the effect of the number of invalidities using ABoxes of fixed size, but with a varying number of invalid data assertions. The results of this set of experiments are illustrated in Figure 3.

From these results, we can conclude that the number of invalid data assertions has no immediate impact on the diagnosis time. On the contrary, it is the main impact factor of the repair computation time. That was an expected behaviour, as the repair computation is done by computing the vertex cover of the interdependency graph. A bigger number of invalid data assertions leads to a bigger graph and this leads to a more costly computation of the vertex cover.

Another significant impact factor of the repair computation time is the amount of interdependencies in the interdependency graph. We can see that the repair computation time increases with a higher rate in the left graph of Figure 3 than in the right one, which can be explained by the fact that the DBpedia 3.6 TBox contains only functional constraints, which form cliques in the interdependency graph (thus, more interdependencies), whereas the DBpedia 3.9 TBox contains mainly disjointness constraints, which cause less interdependencies, therefore less "touching" edges in the interdependency graph.

Moreover, the repair application time seems to be negligible in all of the experiments. This is due to the fact that the repair application is performed by executing a single SPARQL-Update query requesting the deletion of all the triples in the repairing delta, which is very efficient due to the optimizations for batch operations of Virtuoso.

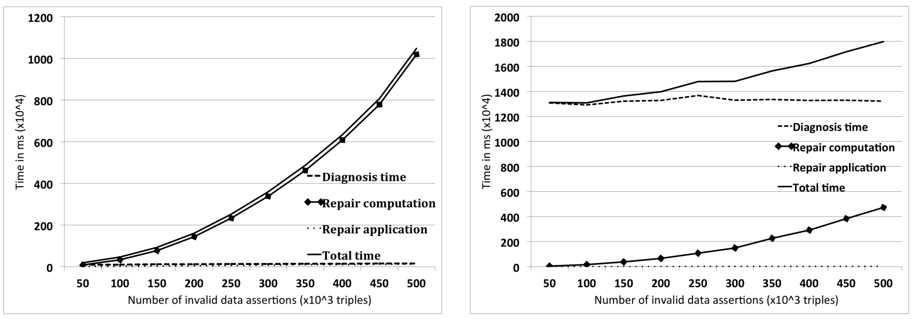


Fig. 3: Performance for DBpedia 3.6 (left) and 3.9 (right) with 10M triples.

The last significant conclusion comes from the comparison of the times measured for the two different DBpedia TBox versions. We see that the diagnosis times for version 3.9 are two orders of magnitude higher compared to the respective times of version 3.6. This is due to the fact that the closure of version 3.9 contains 323.415 constraints, whereas the closure of version 3.6 only 18; more constraints require the generation of more queries to be executed by the diagnosis algorithm, eventually causing this big difference in the measurements.

The following main conclusions can be distilled from our evaluation:

– Diagnosis can be performed in linear time with respect to the ABox size.
– Repair computation can be performed in polynomial time with respect to the number of invalid data assertions that appear in the dataset.

– Our implementation enjoys a decent performance in real-world settings with large datasets, numbers of constraints and invalidities, being able to repair the huge DBpedia 3.9 (>2B triples) in about 10 hours, which is a reasonable amount of time, given that repairing is expected to be an offline process.

It should be noted that the experimental evaluation of the only other work in the literature that performs automated repairing of inconsistent *DL-Lite*$_{\mathcal{A}}$ KBs ([18] – see Section 6), only considers datasets of size up to 30.000 triples, whereas we consider datasets of up to 5 orders of magnitude larger; thus, the results are not comparable.

6 Related Work

The problem of inconsistencies appearing in KBs can be tackled either by providing the ability to query inconsistent data and get consistent answers (*Consistent Query Answering - CQA*) [3], or by actually *repairing* the KB, which leads to a consistent version of it [12]. Both these approaches have attracted researchers' attention, mostly in the context of relational databases and, lately, in the context of linked data and ontology languages as well.

In the context of relational databases, CQA has been studied in various works dealing with different classes of conjunctive queries and denial constraints, mainly key constraints (e.g., [13, 23]). These works underline the main advantages of using First-Order query rewriting for the validation of integrity constraints. Note that CQA techniques systematically drop all information involved in a constraint violation, whereas repairing techniques, like ours, make explicit decisions on what to keep and what to drop, in accordance with the principles set out in [2], that require preserving as much information as possible.

Different semantics have been studied for the repairing of inconsistent relational databases, considering different kinds of constraints. For example, [9] studied the problem of repairing by allowing only tuple deletions and, in this way, resolving violations of denial constraints and inclusion dependencies, which is a more expressive set of constraints than the one we consider in this work. However, as proven in [9], the unrestricted combination of those constraints leads to intractability issues.

In the context of linked data and the corresponding languages and technologies, there has been research on the topic of using ontological languages to encode integrity constraints (ICs) that must be checked over a dataset. In [22], the authors present a way to integrate ICs in OWL and they show that IC validation can be reduced to query answering, for integrity constraints that fall into the $\mathcal{SROI}$ DL fragment. A similar approach has been followed in [19]. In [16], the presented approach integrates constraints that come from the relational world (primary-key, foreign-key) into RDF and provides a way to validate these constraints. IC validation is also an important part of some of the current OWL reasoners, such as Stardog[10]. The above approaches address, essentially, only

[10] `http://stardog.com/`

the KB satisfiability problem and do not consider detection and repairing of invalidities.

In the field of diagnosis for *DL-Lite* KBs, there has been some work regarding inconsistency checking. The *DL-Lite*$_{\mathcal{A}}$ reasoner QuOnto [1] has the ability to check the satisfiability of a *DL-Lite*$_{\mathcal{A}}$ KB. However, it does not detect the invalid data assertions in the ABox, neither repairs it. A problem very similar to repairing (but in a different setting) is addressed in the context of *DL-Lite* KB evolution (e.g., [7], [21]), where the objective is to identify the minimal set of assertions to remove in order to render a *DL-Lite* KB consistent during evolution.

Recently, there has also been some research on CQA for inconsistent knowledge bases expressed in Description Logic languages, using query rewriting techniques. For example, [17] deals with different variants of inconsistency-tolerant semantics to reach a good compromise between expressive power of the semantics and computational complexity of inconsistency-tolerant query answering.

Finally, [18] is (to our knowledge) the only work addressing the automatic repairing of an inconsistent *DL-Lite*$_{\mathcal{A}}$ KB, and thus the closest to our work. It is based on the inconsistency-tolerant semantics studied in [17] and resolves each invalidity by removing both data assertions that take part in it. On the contrary, our repairing algorithm considers the removal of only one of two involved data assertions. Thus, [18] removes more information than necessary from the original KB. In addition, the work of [18] has only been evaluated with datasets that are unrealistically small (up to 30.000 triples).

7 Conclusion and Future Work

We presented a novel, fully automatic and modular diagnosis and repairing framework, which can be used on top of already deployed datasets to assist the curators in the task of enforcing the validity of logical integrity constraints, taking into account logical inference, in order to maintaining their consistency. Our experimental evaluation showed that our framework is scalable for large dataset sizes, often found in real reference linked datasets such as DBpedia.

As future work, we will try to improve the scalability properties of our algorithms, possibly using a parallel implementation relying on the MapReduce model. In addition, we will consider different models of interaction with the curator, to allow him to influence the repairing process (e.g., via user guidelines or preferences) without being overwhelmed with the complete set of invalidities; the ultimate goal is to develop an interactive repairing process that will combine the quality of manual curation with the efficiency of automatic repairing. Another possible extension is to experiment with more LOD datasets, and provide a comprehensive study of the number and types of violations that exist in different popular datasets.

Acknowledgement. This work was partially supported by the EU project DIACHRON (ICT-2011.4.3, #601043), and by the State Scholarships Foundation.

References

1. Acciarri, A., Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Palmieri, M., Rosati, R.: QUONTO: querying ontologies. In: AAAI. Volume 5. (2005)
2. Afrati, F., Kolaitis, P.: Repair checking in inconsistent databases: algorithms and complexity. In: ICDT. (2009)
3. Arenas, M., Bertossi, L., Chomicki, J.: Consistent query answers in inconsistent databases. In: PODS. (1999)
4. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rosati, R.: Linking data to ontologies: The Description Logic DL-Lite$_{\mathcal{A}}$. In: OWLED. (2006)
5. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in Description Logics: The DL-Lite family. Journal of Automated Reasoning. 39(3). (2007)
6. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodriguez-Muro, M., Rosati, R.: Ontologies and databases: The DL-Lite approach. In: Reasoning Web. (2009)
7. Calvanese, D., Kharlamov, E., Nutt, W., Zheleznyakov, D.: Evolution of DL-Lite Knowledge Bases. In: ISWC. (2010)
8. Chomicki, J., Marcinkowski, J.: On the computational complexity of minimal-change integrity maintenance in relational databases. Inconsist. Toler. (2005)
9. Chomicki, J., Marcinkowski, J.: Minimal-change integrity maintenance using tuple deletions. Information and Computation. 197(1). (2005)
10. Deutsch, A.: FOL modeling of integrity constraints (dependencies). In: Encyclopedia of Database Systems. Springer. (2009)
11. Garey, M., Johnson, D.: Computers and intractability. Vol. 174. Freeman. (1979)
12. Greco, G., Greco, S., Zumpano, E.: A logical framework for querying and repairing inconsistent databases. IEEE TKDE. 15(6). (2003)
13. Grieco, L., Lembo, D., Rosati, R., Ruzzi, M.: Consistent query answering under key and exclusion dependencies: Algorithms and experiments. In: CIKM. (2005)
14. Karakostas, G.: A better approximation ratio for the vertex cover problem. In: Automata, Languages and Programming. (2005)
15. Kontokostas, D., Westphal, P., Auer, S., Hellmann, S., Lehmann, J., Cornelissen, R.: Databugger: a test-driven framework for debugging the web of data. In: WWW (Companion Volume). (2014)
16. Lausen, G., Meier, M., Schmidt, M.: SPARQLing constraints for RDF. In: EDBT. (2008)
17. Lembo, D., Lenzerini, M., Rosati, R., Ruzzi, M., Savo, D.: Query rewriting for inconsistent DL-Lite ontologies. In: Web Reasoning and Rule Systems. (2011)
18. Masotti, G., Rosati, R., Ruzzi, M.: Practical ABox cleaning in DL-Lite (progress report). In: Description Logics. (2011)
19. Motik, B., Horrocks, I., Sattler, U.: Bridging the gap between OWL and relational databases. In: WWW. (2007)
20. Papadimitriou, C., Steiglitz, K.: Combinatorial optimization: algorithms and complexity. Courier Dover Publications. (1998)
21. Qi, G., Wang, Z., Wang, K., Fu, X., Zhuang, Z.: Approximating model-based ABox revision in DL-Lite: Theory and practice. In: AAAI. (2015)
22. Tao, J., Sirin, E., Bao, J., McGuinness, D.: Integrity constraints in OWL. In: AAAI. (2010)
23. Wijsen, J.: Consistent query answering under primary keys: a characterization of tractable queries. In: ICDT. (2009)