PlanetData

Network of Excellence

FP7 – 257641

D2.2 Methods for Quality Repair

Coordinator: Giorgos Flouris

With contributions from:

Yannis Roussakis, Vassilis Christophides

1st Quality Reviewer: Zoltan Miklos
2nd Quality Reviewer: Pablo Mendes

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M18
Actual delivery date:	M18
Version:	1.0
Total number of pages:	38
Keywords:	Validity, Diagnosis, Repair, RDF(S) DB, RDF(S), Linked Data, DED

Abstract

This deliverable studies the problem of *quality repair* in RDF(S) Linked Data with particular emphasis on the validity dimension. Validity is a very generic notion that refers to the set of assumptions or requirements regarding the data that are made by the applications that are using the data. Validity is a crucial requirement because data that does not satisfy the application-imposed requirements could cause the related applications to function sub-optimally, or fail altogether, i.e., rendering the data useless. Due to the importance of validity as a quality dimension, we consider techniques both for identifying and imposing validity. The latter refers to repair approaches that modify invalid datasets in order to render them valid, causing minimal changes on the original data (according to some custom, user-defined metric of minimality). We provide technical solutions for both diagnosing invalidities and repairing invalid datasets, and show some experiments.

Executive Summary

During the last years we have witnessed a tremendous increase in the amount of possibly interlinked RDF(S) datasets published on the Web in the context of Linked Data. We will refer to these datasets as *RDF(S) DBs*. In such a setting, quality issues arise, as interlinked data do not necessarily satisfy any quality standards. The issue of quality in general, with emphasis on defining a conceptual model for quality assessment, is studied in Deliverable D2.1; the present deliverable complements D2.1, in the sense that it focuses on the problem of *quality repair*, i.e., the problem of imposing quality. We focus on a specific, and important, quality dimension, namely *validity*, but our approach could be adapted for other quality dimensions as well.

Validity is a very generic notion that refers to the set of assumptions or requirements regarding the data that are made by the applications that are using the data. Validity is a crucial requirement because more and more RDF(S) DBs are coupled with various domain or application specific integrity constraints to ensure the quality of the Linked Datasets published on the web. Data that does not satisfy the application-imposed requirements could cause the related applications to function sub-optimally, or fail altogether, i.e., rendering the data useless. As such, validity is necessary for data-centric applications to work.

In this deliverable we focus on RDF(S) data. Our approach will be based on a mapping of RDF(S) datasets to relational data and can be applied on a multitude of different contexts with minor modifications. We study and formalize the notion of validity, and propose ways to identify invalidities (*diagnosis*). However, the main focus of this deliverable is on techniques for imposing validity, i.e., *repair* approaches that modify invalid datasets in order to render them valid, causing minimal changes on the original data (according to some custom, user-defined metric of minimality).

Validity will be formalized using rules of the form of Disjunctive Embedded Dependencies (DEDs), which have been shown to be powerful enough to describe a wide range of conditions and requirements (including primary and foreign key constraints, acyclicity, transitivity and other properties, cardinality constraints etc). Determining invalidities using this formalization is reduced to searching the RDF(S) DB for specific tuples.

As already mentioned, the main focus of this deliverable is on *quality repair*. In this respect, we describe a declarative framework for assisting curators in the arduous task of repairing constraint violations and restoring validity in RDF(S) DBs; note that this task is performed nowadays manually. A major challenge is that there are usually many ways to resolve each single invalidity, and it is usually difficult for a system to determine the optimal one automatically without user feedback. To address this problem while allowing automated repairs, we use an intermediate approach, where the users can personalize the repair approach by providing a set of "specifications" for the ideal solution; the system will then automatically search for the solution (repair) that is closest to the ideal one. More specifically, personalization is performed by allowing curators to specify complex preferences over interesting features of potential repairs (e.g., their size and type) that can capture diverse notions of minimality of repair. The aforementioned specifications may be related either to the repair itself (i.e., the resulting RDF(S) DB), or the steps required to get from the invalid RDF(S) DB to the repair, and could use features like the number of changes required, the number of additions required, or the quality of the resulting RDF(S) DB in terms of the quality metrics presented in Deliverable D2.1.

Another challenge to repairing is related to the expressive power of the used constraints (DEDs), which gives rise to interdependencies between violated constraints and their possible resolutions. For example, resolving one violation in a certain way could cause another violation; another case is that a given resolution could resolve more than one violated constraints at the same time. Repairing in the presence of interdependencies forces us to consider all possible resolution options, as it is not possible to predict in advance the positive or negative ramifications of each possible resolution option.

To address these problems, we propose both an exhaustive and a greedy repair finding algorithm, and prove that only the former is immune to the resolution order and syntax of violated constraints and can thus correctly compute globally optimal repairs for different kinds of constraints and preferences. Since both are of exponential nature, we have considered a series of optimizations that reduce the number of validity checks that need to be performed, as well as the search space explored. We experimentally demonstrate that, despite the discouraging complexity, in practice both algorithms scale linearly to the size of the RDF(S) DB. Moreover, in most practical cases (large RDF(S) DBs with a small number of changes) the time to find the preferred repairs is dominated by

the time required to check for constraint violations (quality assessment). To show the feasibility of our approach to real-world settings, we also perform experiments with large, real-world datasets, and explore issues related to user interface and interaction.

Document Information

IST Project Number	FP7 – 257641	Acronym	PlanetData
Full Title	PlanetData		
Project URL	http://www.planet-data.eu/		
Document URL	http://planet-data-wiki.sti2.at/uploads/a/a6/D2.2.pdf		
EU Project Officer	Leonhard Maqua		

Deliverable	Number	D2.2	Title	Methods for Quality Repair
Work Package	Number	WP2	Title	Quality Assessment and Context

Date of Delivery	Contractual	M18	Actual	M18
Status	version 1.0		final ☒	
Nature	Report (R) ☒ Prototype (P) ☐ Demonstrator (D) ☐ Other (O) ☐			
Dissemination Level	Public (PU) ☒ Restricted to group (RE) ☐ Restricted to programme (PP) ☐ Consortium (CO) ☐			

Authors (Partner)	Giorgos Flouris (FORTH)			
Responsible Author	Name	Giorgos Flouris	E-mail	fgeo@ics.forth.gr
	Partner	FORTH	Phone	+302810391674

Abstract (for dissemination)	This deliverable studies the problem of *quality repair* in RDF(S) Linked Data with particular emphasis on the validity dimension. Validity is a very generic notion that refers to the set of assumptions or requirements regarding the data that are made by the applications that are using the data. Validity is a crucial requirement because data that does not satisfy the application-imposed requirements could cause the related applications to function sub-optimally, or fail altogether, i.e., rendering the data useless. Due to the importance of validity as a quality dimension, we consider techniques both for identifying and imposing validity. The latter refers to repair approaches that modify invalid datasets in order to render them valid, causing minimal changes on the original data (according to some custom, user-defined metric of minimality). We provide technical solutions for both diagnosing invalidities and repairing invalid datasets, and show some experiments.
Keywords	Validity, Diagnosis, Repair, RDF(S) DB, RDF(S), Linked Data, DED

Version Log			
Issue Date	Rev. No.	Author	Change
20/02/2012	0.5	Giorgos Flouris	First full draft
24/02/2012	0.9	Giorgos Flouris	Ready for internal review process
16/03/2012	1.0	Giorgos Flouris	Final version, to submit to EU

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 General Context and Importance of the Problem

During the last years we have witnessed a tremendous increase in the amount of RDF(S) data published on the Web in almost every field of human activity. For example, billions of RDF(S) triples from Wikipedia, U.S. Census, CIA World Factbook, open government sites in the US and the UK, memory organizations like the British Museum and Europeana, as well as news and entertainment sources, such as BBC, are published nowadays on the so-called *Web of Data*, along with numerous vocabularies and conceptual schemas from e-science aiming to facilitate annotation and interlinking of scientific and scholarly data.

As a matter of fact, more and more data break the walls of private management within their production site, and become available as Linked Data enabling a new generation of decision making and business intelligence applications across domains. In this global data space, data takes the form of RDF(S) DBs whose semantics can be explicitly given by related ontologies while they can be freely interlinked with others from remote hosts [15]. In this context, various issues related to the quality of the data emerge; a detailed analysis of these quality issues, with emphasis on presenting a conceptual model for *quality assessment*, appears in Deliverable D2.1 [26]. The present deliverable focuses on *quality repair*, i.e., how to impose quality, with emphasis on one of the most important dimensions of data quality, namely *validity*. We only consider validity on RDF(S) datasets (RDF(S) DBs), i.e., we don't consider the extra features sometimes included in Linked Data (e.g., triples using OWL properties).

Validity is a very generic notion that refers to the set of assumptions or requirements regarding the data that are made by the applications that are using the data. Validity is relevant in many RDF(S) DBs, which often have to satisfy various domain or application specific constraints [7, 20, 28, 32]. For example, some works related to query optimization, in order to improve the algorithms' performance, make the assumption that class and property hierarchies should be acyclic [32]. Other applications may require the introduction of cardinality constraints, functional properties or other context-specific constraints (such as e.g., the requirement that a person cannot be his own father). Note that validity only refers to logical assumptions regarding the data; other types of requirements, such as, e.g., the semantically correct use of properties, or the requirement for an RDF(S) DB to be up-to-date, are covered by other quality dimensions and are described in detail in Deliverable D2.1 [26].

Validity is a crucial requirement because data that does not satisfy the application-imposed requirements could cause the related applications to function sub-optimally, or fail altogether, i.e., rendering the data useless. This is in contrast with some of the other quality metrics, e.g., timeliness, which are important but cannot cause applications to fail or render data useless for a particular purpose. Due to the importance of validity as a quality dimension, it is often desirable to develop techniques for imposing validity; the present deliverable addresses this problem, i.e., it considers repair approaches that modify invalid datasets in order to render them valid, causing minimal changes on the original data (according to some custom, user-defined metric of minimality).

1.2 Contributions of this Deliverable

In this deliverable we focus on RDF(S) datasets. Our approach will be based on a mapping of RDF(S) datasets to relational data (RDF(S) DBs) and can be applied on a multitude of different contexts with minor modifications.

Invalidities in RDF(S) DBs may arise in a multitude of ways. For example, an invalidity may appear when an RDF(S) DB evolves due to newly acquired information, revisions to the intended usage or simply errors; also invalidities may occur when changes of interlinked RDF(S) DBs are propagated across the Web, or even when the employed constraints themselves get revised. Clearly, the value of Linked Data lies in the quality of the published RDF(S) DBs and thus for successful data analytics we need to cope with the involved invalidities. This problem has been addressed in the literature [3] either by providing valid query answers over invalid data [4] or by actually *repairing* DBs [11]. Since curators update their RDF(S) DBs to constantly ensure data quality, we are interested in *declarative repairing* frameworks for assisting them in the arduous task of *identifying* and *resolving* constraint violations (i.e., invalidities) which is performed nowadays mostly manually.

Disjunctive Embedded Dependencies (DEDs) [10] have been proved expressive enough to capture a *variety of integrity constraints* related to frequent invalidities of RDF(S) DBs such as acyclicity of certain properties (e.g., subsumption) [32], primary/foreign key constraints [8], or cardinality constraints [28]. Note that most of the existing repairing algorithms focus on specific sub-forms of DEDs (e.g., consider only functional, conditional functional, and inclusion dependencies [5, 9]). It is important to note that DEDs are powerful enough to capture not only constraints related to validity, but also other quality dimensions, such as incompleteness or inconsistency (see Deliverable D2.1 [26] for details on these quality dimensions), as long as these dimensions can be expressed in a logical form.

There are two major challenges as far as repairing invalid RDF(S) DBs is concerned. The first is to identify the invalidities associated with an RDF(S) DB, also referred to as the *diagnosis* problem. Using DEDs, diagnosis is reduced to a simple reasoning problem, and we will show how it can be addressed in Section 3; diagnosis corresponds to the *quality assessment* problem discussed in Deliverable D2.1 [26] and is a prerequisite to resolving (repairing) these invalidities. The second, and more difficult problem is how to resolve these invalidities, commonly known as the *repair* problem. The major challenge regarding repair is related to the possible *interdependence of DED constraints*, i.e., the fact that the resolution of a constraint violation may cause the violation of other constraints and/or the simultaneous resolution of other violations. This deliverable focuses mainly on the (harder) problem of repair.

It has been argued that the optimal resolution is the one that causes *minimal* DB effects (updates) [1, 8]. Note that the requirement of minimality has also been coined in other, related contexts and research fields, e.g., in ontology evolution [19] and belief revision (through the Principle of Minimal Change [12]).

Thus, in the context of repairing, the purpose of a *declarative repairing framework* is the automatic identification of a valid DB that would have a minimal delta compared to the original DB [11]. Minimality is a context-dependent notion, and can be either defined in relation to the number and/or type of changes that must be applied upon the original DB to repair it, or in relation to the quality of the final repair result. In both cases, its exact definition depends on a number of underlying assumptions (e.g., when the curator considers the RDF(S) DB to be complete, he may favor repairs performing removals [1], whereas in the opposite case, he may prefer repairs performing additions [22]). In existing algorithms (e.g., [5, 6, 9]) the minimality preference is fixed at design time and curators cannot intervene. Furthermore, they usually consider the resolution of each constraint violation in isolation from the others; as we will show in Section 2, this often makes the result of repairing sensitive to the evaluation order of the constraints, as well as their syntactic form. In a nutshell, this deliverable makes the following contributions:

- In Section 3 we propose a *generic* and *personalized* repairing framework which supports a *variety of useful integrity constraints*, and show how invalidities can be diagnosed in this setting. In addition, we introduce complex curator *preferences* over interesting features of the resulting repairs [13, 17] (e.g., the number and type of repairing updates), which can be set at *run-time* and used in the repair process to determine the preferred repair. The proposed framework is expressive enough to capture different notions of minimality that have been proposed in the literature (e.g., [1, 11]). This way, existing repair algorithms can be revisited, compared or reviewed, under the light of our framework (see also Propositions 2, 3 and Section 8).

- In Section 4 we describe two *repair resolution strategies*: a *globally optimal* (GO) strategy that exhaustively considers all resolutions in order to identify globally optimal repairs (per the minimality preference) and a *locally optimal* (LO) strategy which verifies minimality during the resolution of each invalidity in isolation (greedily). Despite the fact that most of the existing algorithms [5, 6, 9] rely on the latter "short-sighted" strategy, we show that the two strategies do not yield, in general, the same resulting repairs since only GO is immune to changes in the constraints' syntax and evaluation order.

- In Section 5 we introduce two repair finding algorithms according to the above strategies and study their analytical complexity bounds. Since both are of exponential nature, we have considered a series of optimizations that reduce the number of validity checks that need to be performed, as well as the search space (resolution options) that need to be explored.

- In Section 6 we experimentally evaluate the effectiveness and efficiency of GO and LO algorithms using both actual and synthetic RDF(S) DBs. The synthetic datasets were created according to various distributions exhibited in reality [35], and were used to show that both algorithms scale linearly with respect to the size of the RDF(S) DBs. In our prototype implementation the time required for validity checking (diagnosis) scales linearly to the RDF(S) DBs size, whereas the actual repairing process is determined by the size of the search space of potential repair resolutions that the algorithms have to explore. Thus, in practical curation cases (i.e., large RDF(S) DBs with a small number of violations), the overhead imposed by the repair process itself is small and performance is dominated by the diagnosis time. Finally, the pruning effect of the employed repair preference is crucial for LO behaviour: when little pruning is performed, LO performance is similar to GO, whereas drastic pruning increases the possibility that LO returns less-than-optimal repairs (see Section 6.5 for an example). The experiments using real-world datasets showed that our approach is feasible also in real-world data.
- In Section 7 we explore technical implementation issues, in particular we explain how one should develop an adequate interface for using our framework with large real-world curation cases.

Finally, Section 8 reviews related work while Section 9 concludes and sketches contributions for future research. Note that an earlier version of this work appears in [31].

2 Problem Statement

Throughout this deliverable we consider a relational representation of RDF(S) ontology and instance data that is suitable for dynamic DBs [33] on which DED integrity constraints will also be expressed. More precisely, $CS(B)$, denotes that B is a class, $C\_IsA(B, A)$, denotes a (direct or transitive) subsumption relationship between B, A, $PS(P)$, denotes that P is a property, and $Domain(P, B)$, $Range(P, B)$, denote that the domain and range (respectively) of P is B. Unlike the widely used representation in a single triple table with three columns, we rely on a typed relational encoding of RDF(S) data that can still afford ontology changes [19]. This is not the case when encoding each ontology class and property by a different table, which has employed by unifying frameworks for query answering under constraints expressed by an ontological schema [7].

Various kinds of constraints related to the ontology schema and data and recommended by ontology curation best practices [16] can be expressed using DEDs [10]; here, we consider the constraints that have been proposed in [32], which are a superset of those used in minimal RDF(S) [29], and are required by a wide range of curated KBs. They concern the type uniqueness, i.e., that each resource has a unique type, the acyclicity of subClassOf and subPropertyOf relations, the uniqueness of properties' domains and ranges, that the domain and range of a property should be subclasses of the corresponding domain/range of the superproperty, and that the subject and object of some property instance should be correctly classified under the domain and range of the property respectively. These constraints have been introduced to allow efficient query answering in large RDF(S) DBs.

For simplicity, we will initially formalize only some of the above constraints using the DED constraints below, which state that all subsumption relationships should be between defined classes (IC_1), that properties should have a defined domain and range (IC_2), which in turn should be a defined class (IC_3, IC_4); more constraints related to the above validity model, concerning both the ontology schema and data, are given in Section 6.

$IC_1 : \forall x_1, x_2 C\_IsA(x_1, x_2) \rightarrow CS(x_1) \wedge CS(x_2)$

$IC_2 : \forall x PS(x) \rightarrow \exists y_1, y_2 Domain(x, y_1) \wedge Range(x, y_2)$

$IC_3 : \forall x_1, x_2 Range(x_1, x_2) \rightarrow PS(x_1) \wedge CS(x_2)$

$IC_4 : \forall x_1, x_2 Domain(x_1, x_2) \rightarrow PS(x_1) \wedge CS(x_2)$

Note that this is only one possible definition of the notion of validity; a different set of requirements (formally expressed using a different set of DEDs) could be used to define validity in a different context or application.

For illustration purposes, we will consider a DB $D = \{CS(B), C\_IsA(B, A), PS(P), Range(P, B)\}$. D violates IC_1 as we have the subsumption relation $C\_IsA(B, A)$ but the class A is not declared (i.e., $CS(A) \notin D$). Moreover, it violates IC_2, because the DB contains a property (P) with no domain. The widely-used repairing strategy [6, 9] consists in selecting one violated constraint, repairing it, then finding the next violated constraint etc, until no more violations exist. The resolution options for a constraint can be deduced from its syntax: for example, to resolve IC_1 we must either remove $C\_IsA(B, A)$ or add $CS(A)$; note however that many popular approaches (e.g., [9]) consider only some of the options, rather than all of them. Similarly, to resolve IC_2, we can remove $PS(P)$, or add $Domain(P, x)$ for any x. This process can be modeled in a *resolution tree*, at each node of which one constraint violation is resolved (Figure 2.1).

Observe that our repairing choices are not independent, but may have unforeseen consequences. For example, if we remove $PS(P)$ to resolve IC_2, then IC_3 is subsequently violated, and this violation is caused by the removal of $PS(P)$. Similarly, if we add $Domain(P, A)$ to resolve IC_2, then IC_4 is subsequently violated. The latter violation is prevented if we resolve IC_1 by adding $CS(A)$; in this case, the addition of $CS(A)$ resolves two constraint violations at the same time.

2.1 Repair Strategies

In popular frameworks (e.g., [5, 6, 9]), the repair finding strategy consists in selecting a violated constraint, identifying the optimal resolution options for said constraint and applying them, discarding the non-optimal

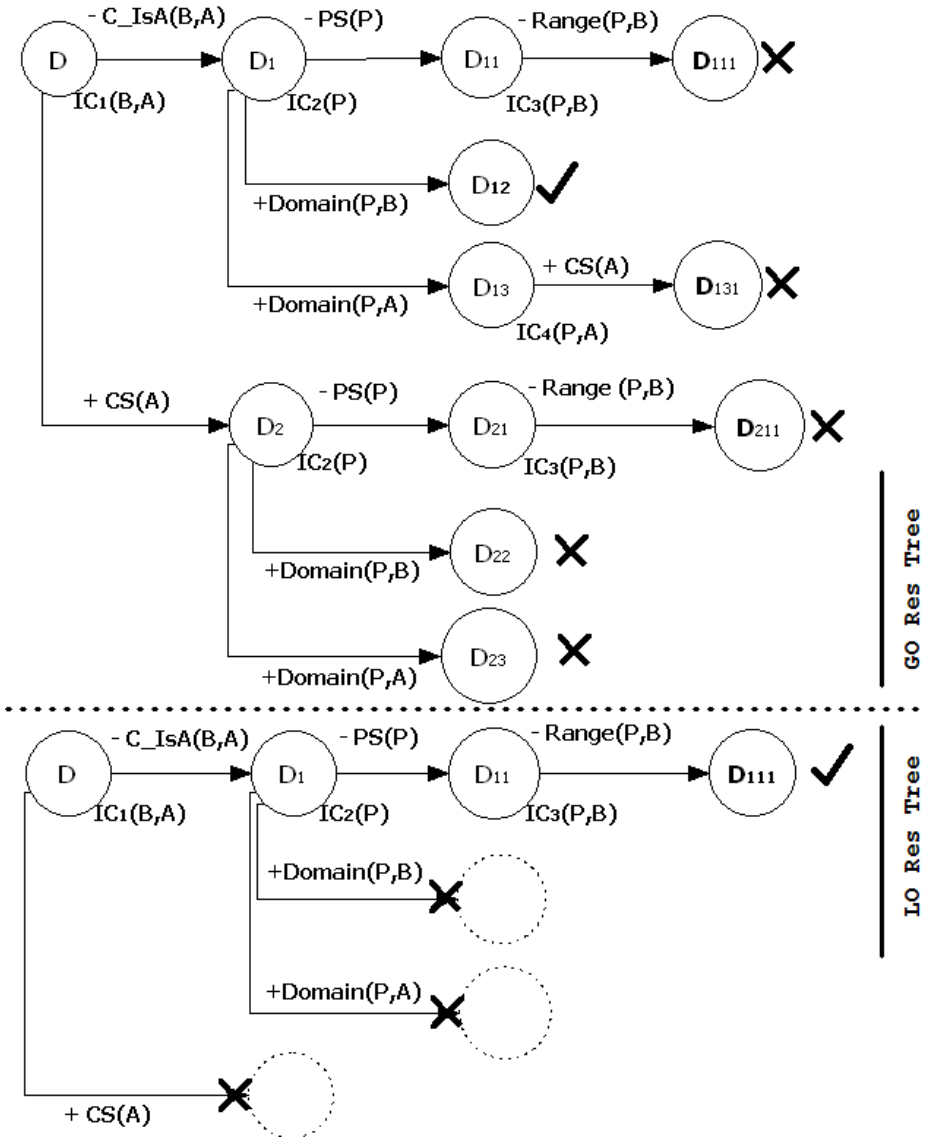

Figure 2.1: Resolution Trees (Globally-optimal and Locally-optimal)

ones; then the process is repeated for the next violation. This is also similar to the chase process [23], which is used to identify data implications when integrity constraints are seen as data dependencies (i.e., inference rules). This process can be modeled using a resolution tree whose leaves *correspond to preferred repairs*. We will call this strategy *locally optimal (LO)* because optimality is determined "locally" for each violation. The lower part of Figure 2.1 shows one application of this strategy, in which the adopted policy is: "I want to make minimal update steps during repair; in case of ties, I prefer removals". The dotted circles in Figure 2.1 indicate rejected resolution options. Under the above policy, the algorithm would first choose to remove $C\_IsA(B, A)$ to restore IC_1 (this option is preferable than the addition of $CS(A)$). The new RDF(S) DB (D_1) violates IC_2, so we remove $PS(P)$ (to resolve IC_2) and finally remove $Range(P, B)$ (for IC_3); thus, the (only) returned repair would be $D_{111} = \{CS(B)\}$. The resolutions under this strategy are selected in a greedy manner, in the sense that only the current constraint is considered, and the potential consequences of a selected/rejected option are neglected. Thus, we may miss more preferred repairs, as in the repair $D' = \{PS(P), CS(B), Range(P, B), Domain(P, B)\}$, which occurs from the original RDF(S) DB D with only two update steps, so it is preferred over D_{111} under our policy and should have been returned instead.

An additional shortcoming of the LO strategy is that it is often sensitive to the constraint evaluation order and syntactic form. To see this, consider the same example with the policy: "I prefer deletion of class-related

information (CS, C_IsA) over additions; but I prefer both over the addition of property-related information ($PS, Range, Domain$), which, in turn, is preferable than deletion of property-related information". Let us consider the evaluation order: $IC_2 \rightsquigarrow IC_1 \rightsquigarrow IC_3 \rightsquigarrow IC_4$. First, IC_2 is resolved using two alternative options: add $Domain(P, A)$ or add $Domain(P, B)$. Let us concentrate on the first option, i.e., add $Domain(P, A)$. Then, the resolution of IC_1 would delete $C\_IsA(B, A)$, and the resolution of IC_4 would add $CS(A)$. The end result is the repair: $D_{order} = \{PS(P), CS(B), CS(A), Range(P, B), Domain(P, A)\}$ (note that this is not the only repair that will be returned). The reader can verify that for the evaluation order $IC_2 \rightsquigarrow IC_3 \rightsquigarrow IC_4 \rightsquigarrow IC_1$, D_{order} will not be returned. The reason is that the branch that adds $Domain(P, A)$, would then resolve IC_4 through the addition of $CS(A)$, which in turn resolves IC_1 implicitly and $C\_IsA(B, A)$ will not be removed.

To see the effect of the constraint syntax, let us slightly change our example and consider the RDF(S) DB $D_{syntax} = \{ PS(P), CS(B)\})$. Also, change the repairing policy into: "I want to make minimal update steps during repair; in case of ties, I prefer additions". Then, the resolution of IC_2 (the only violated constraint) would only accept the branch where $PS(P)$ is removed. If we replace IC_2 with the equivalent set of constraints $IC_{2a} : \forall u PS(u) \rightarrow \exists v_1 Domain(u, v_1)$ $IC_{2b} : \forall u PS(u) \rightarrow \exists v_2 Range(u, v_2)$, then the removal of $PS(P)$ is no longer a selected resolution; instead, the algorithm would add $Domain(P, x)$ and $Range(P, y)$ for some constants x, y. Thus, the returned repair depends on the constraints' syntax, rather than the constraints' semantics.

To address these shortcomings of the LO strategy, we propose the *globally optimal (GO) resolution strategy*, under which none of the resolution options of a violation is rejected; instead, all resolution branches are considered (see upper part of Figure 2.1). Hence, each leaf of the resolution tree is a *potential* repair, but not necessarily a *preferred* one; preferred repairs are determined by comparing all potential ones against the repairing policy. Figure 2.1 shows one application of this strategy: no branches are rejected, but most of the leaves (e.g., D_{111}, D_{131} etc.) are rejected as non-preferred. Going back to our original policy, the RDF(S) DB $D_{12} = \{PS(P), CS(B), Range(P, B), Domain(P, B)\}$ would be returned, as expected. Note that this is a different repair from the one returned by the LO strategy for the same DB and policy, and that the GO resolution tree could not have been created using, e.g., tableau transformation rules (as in [23]), because such rules consider only one potential resolution per violation, discarding the rest.

Given that GO considers complete resolutions, the returned repairs are always the most preferred ones, i.e., globally optimal (unlike LO); in addition, as we will show later (Proposition 1), GO strategies are immune to changes in the constraints' evaluation order or syntax. However, GO strategies have to compute a larger resolution tree, so they are, on average, less efficient than LO strategies, even though the worst-case complexity is the same in both cases (see Section 5). In Section 5, we also show that several optimizations can be considered to reduce the size of the resolution tree in the average case.

2.2 Repair Preferences

To allow the curator to apply (and experiment with) different repair policies, we will use a *formal preference model* over interesting *features* of repairs. We rely on qualitative preference models proposed for relational databases supporting the *declarative* specification of preferences using *atomic and composite preference expressions* over relational attributes [13, 18, 17]. For example, when the curator considers the RDF(S) DB to be complete, he may want to obtain minimum additions, so the interesting feature is the number of additions, whereas the atomic preference is the *minimization function* (Min). The application of this preference under a GO strategy over the possible repairs of our example RDF(S) DB would return the repair $D_{111} = \{CS(B)\}$. To express composite preferences spanning several attributes, a curator may employ *constructors* between atomic preferences, such as $\otimes$ (pareto) and $\&$ (prioritized). For example, a declarative repairing policy could be to equally prefer (i.e., pareto) repairs featuring both a minimum number of updates and a minimum number of additions [17].

As we will see later, there are two ways to formally define these preferences. Under the first, preferences (and features) are defined upon the *changes* that should be applied in the original, invalid RDF(S) DB to result to the repaired, valid RDF(S) DB. The examples used above (e.g., minimum number of updates, or minimum number

of additions) correspond to this method of defining preferences. An alternative is to define preferences (and features) as related to the final repair result. In this respect, quality assessment metrics such as those proposed in Deliverable D2.1 [26] can be used to assess the "value" of each possible repair, and select the best one.

As we will explain later, the two methods are formally equivalent, so their only difference lies in the intuitiveness of defining features and preferences with respect to repairs, or changes that lead to repairs. Additionally, preferences will be shown to be powerful enough to model several diverse resolution policies that have appeared in the literature. Moreover, they are more flexible than other approaches, such as tableau rules [23], because they allow keeping more than one possible options per resolution.

2.3 Labeled Nulls

One problem with GO or LO strategies is that the resolution options for certain types of constraints involve the introduction of new constraint values; as a result, the number of resolution options in these cases is very large. In our motivating example, IC_2 can be resolved by removing $PS(P)$, or by adding $Domain(P, x)$ for any constant x. The problem of *value invention* has been addressed by introducing *labeled nulls* [7], denoted by ε_i, which essentially constitute skolem constants representing in a compact way several alternative constants (also useful when curators inspect repairs). For example, $Domain(P, \varepsilon_1)$ means "$Domain(P, x)$ for some constant x" and can be used as a resolution option for IC_2, thereby reducing the size of the resolution tree by combining several branches into one. More than one labeled nulls are actually needed: if two different properties violated constraint IC_2 (i.e., they had no domains), then a different labeled null would be necessary for each resolution.

However, labeled nulls alone are not enough in our setting. To see this, consider our previous example, and suppose that we resolve IC_2 by adding $Domain(P, \varepsilon_1)$. Then, we note that IC_4 may, or may not be violated, depending on the value of ε_1; e.g., if we replace ε_1 with B, then IC_4 is not violated, but if we replace it with A then IC_4 is violated (because $CS(A) \notin D$).

This problem raises the need to constrain the values that a labeled null could potentially take. In particular, we would like to follow different resolution paths depending on whether IC_4 needs to be resolved (i.e., is violated) or not. Thus, we introduce the *labeled null range*, Ω, which is a set containing the values that the used labeled null(s) can take. In our example, if $\Omega = \{B\}$ then IC_4 is not violated.

3 Preliminaries

We consider standard relational semantics and some arbitrary schema. A *DB* D is any finite set of relational atoms of the form $R(\vec{a})$, where R is a relation in said schema and $\vec{a}$ a tuple of constants. We denote by $\mathcal{D}$ the set of all DBs. We adopt the relational semantics which make the *Closed World Assumption*, so: $D \vdash R(\vec{a})$ iff $R(\vec{a}) \in D$, and $D \vdash \neg R(\vec{a})$ iff $R(\vec{a}) \notin D$. We equip our framework with *labeled nulls* ($\varepsilon_1, \varepsilon_2, \ldots$). The *range* of the labeled nulls, Ω, is a set that determines the constants that each labeled null can be replaced with.

Labeled nulls can be used to compactly represent a set of different RDF(S) DBs. We use the notation $[[D]]^{\Omega}$ to denote the set of DBs that occur from D by replacing the labeled nulls in D with constants, as specified by Ω (e.g., $[[R(\varepsilon)]]^{\{A,B\}}$ corresponds to $\{\{R(A)\}, \{R(B)\}\}$). Note that if we had more than one labeled null to deal with, the range would contain tuples of constants (each element of the tuple representing one labeled null).

Changes performed during repairs are represented using *deltas* $\delta = \langle \delta_a, \delta_d \rangle$, which contain the sets of relational atoms to be added (δ_a) or removed (δ_d) from the RDF(S) DB. As with DBs, deltas may contain labeled nulls in order to compactly represent several deltas, and the same notation will be used ($[[\delta]]^{\Omega}$). We denote by Δ the set of all deltas.

The *application* of a delta $\delta = \langle \delta_a, \delta_d \rangle$ upon D is simply defined as $D \bullet \delta = (D \cup \delta_a) \setminus \delta_d$. Deltas can be also *composed* to produce a delta with a cumulative effect; given two deltas $\delta_1 = \langle \delta_{a1}, \delta_{d1} \rangle$, $\delta_2 = \langle \delta_{a2}, \delta_{d2} \rangle$, their composition is defined as: $\delta_1 \uplus \delta_2 = \langle \delta_{a1} \cup \delta_{a2}, \delta_{d1} \cup \delta_{d2} \rangle$. The above definitions can be easily generalized for deltas with labeled nulls.

3.1 Integrity Constraints

The *integrity constraints (ICs)* that we consider are first-order logical formulas over relational atoms without labeled nulls. We consider ICs under the form of DEDs (Disjunctive Embedded Dependencies), in particular the class $DED^{\neq}$, which has the general form $\forall \vec{x} p(\vec{x}) \rightarrow \vee_{i=1,\ldots,n} \exists \vec{y_i} q_i(\vec{x}, \vec{y_i})$, where $\vec{x}$,$\vec{y_i}$ are tuples of variables and p, q_i are conjunctions of relational atoms and/or (in)equality atoms of the form $(w = w')$, $(w \neq w')$, where w, w' are variables or constants (p may be empty).

A *constraint set* is a finite set of constraints $\mathcal{I} = \{IC_1, \ldots, IC_n\}$. Given a constraint IC and an assignment of the universally quantified variables ($\vec{x}$) to constants ($\vec{a}$), the *instance of* IC with respect to $\vec{a}$ (denoted by $IC(\vec{a})$), is the formula that is produced by replacing all variables from $\vec{x}$ in IC by their corresponding assignment in $\vec{a}$.

We employ the *unique name assumption*, thus different constants correspond (by default) to different real-world entities. Thus, given two different constants a_1, a_2, the (in)equality axiom $a_1 = a_2$ ($a_1 \neq a_2$) always evaluates to $false$ ($true$). Thanks to this assumption, equality axioms can always be eliminated from constraint instances. For example, for $IC_{eq1}\ (x_1, x_2) = \forall x_1, x_2 p(x_1, x_2) \rightarrow (x_1 = x_2)$, it holds that $IC_{eq1}\ (a_1, a_2) = \neg p(a_1, a_2)$ and $IC_{eq1}(a, a) = true$. Similarly, we can eliminate inequality axioms that involve constants and/or universally (but not existentially) quantified variables.

We say that a DB D *satisfies* the constraint instance $IC(\vec{a})$, iff $D \vdash IC(\vec{a})$. A DB satisfies a constraint IC iff it satisfies all its instances. Similarly, an RDF(S) DB satisfies a constraint set $\mathcal{I}$ iff it satisfies all constraints in the set. An RDF(S) DB *violates* a constraint instance iff it does not satisfy it (similarly for constraints and constraint sets). An RDF(S) DB is *valid* with respect to $\mathcal{I}$ iff it satisfies $\mathcal{I}$.

3.2 Detecting and Resolving invalidities

The form of DED constraints allows both the easy detection of a violation (diagnosis) as well as the determination of all possible repairing options for this violation. This can be easily seen with an example: let us take the constraint instance $IC_1(a_1, a_2)$ from the motivating example. Then, according to our semantics, for a given RDF(S) DB D, $D \vdash IC_1(a_1, a_2)$ iff $C\_IsA(a_1, a_2) \notin D$ or $CS(a_1), CS(a_2) \in D$. Therefore, if $D \nvdash IC_1(a_1, a_2)$, then none of the above conditions holds, so we can resolve this invalidity by making any of those conditions true, i.e., either by removing $C\_IsA(a_1, a_2)$ from D or by adding $CS(a_1), CS(a_2)$ to D.

The idea can be easily extended to DEDs with existential quantifiers. For IC_2 for example, we get that $IC_2(a)$ is satisfied by D iff $PS(a) \notin D$ or there are constants b, c such that $Domain(a, b), Range(a, c) \in D$. If $D \nvdash IC_2(a)$, then the invalidity can be resolved either by removing $PS(a)$ from D, or by adding $Domain(a, b)$ and $Range(a, c)$ to D for some constants b, c; using labeled nulls, the latter option can be compactly written as: $[[\delta]]^{\Omega}$ where $\delta = \langle \{Domain(a, \varepsilon_1), Range(a, \varepsilon_2)\}, \emptyset \rangle$ and $\Omega = \mathcal{U}_D \times \mathcal{U}_D$ (where $\mathcal{U}_D$ contains all the constants appearing in D).

Finally, note that equality axioms in a constraint need not be considered, because they are eliminated in each of its instances (see Section 3.1). Inequality axioms can appear inside existential quantifiers, in which case they simply overrule some of the options (by restricting Ω).

An RDF(S) DB featuring labeled nulls corresponds to a set of RDF(S) DBs ($[[D]]^{\Omega}$); thus, some of the RDF(S) DBs in $[[D]]^{\Omega}$ may satisfy a given constraint instance, while others may not. Given that it does not make sense to resolve a violation that does not exist in the first place, detection and resolution in this case involves, first, discriminating between *violating* (say $[[D]]^{\Omega_V}$) and *non-violating* (say $[[D]]^{\Omega_{NV}}$) members of $[[D]]^{\Omega}$, and, second, the resolution of the violation for the violating members of $[[D]]^{\Omega}$ (i.e., $[[D]]^{\Omega_V}$) in the standard manner.

As an example of applying this idea, consider $IC_4 = \forall x_1, x_2 Domain(x_1, x_2) \rightarrow PS(x_1) \wedge CS(x_2)$ and the RDF(S) DB $D = \{CS(B), PS(P), Range(P, B), Domain(P, \varepsilon)\}$ for $\Omega = \{A, B\}$. There is only one fact of the form $Domain(a, b)$ in D, namely $Domain(P, \varepsilon)$; moreover, $PS(P) \in D$. Per our semantics, IC_4 can be violated only for those values of ε for which $CS(\varepsilon) \notin D$. Thus, D violates IC_4 (in particular, $IC_4(P, A)$) for $\Omega_V = \{A\}$, and is satisfied for the remaining elements of $[[D]]^{\Omega}$, i.e., for $[[D]]^{\Omega_{NV}}$, where $\Omega_{NV} = \{B\}$. Thus, the first step towards the resolution of this invalidity is the creation of two separate recursive branches, one for $[[D]]^{\Omega_{NV}}$ and one for $[[D]]^{\Omega_V}$. For the former case, the invalidity no longer exists; for the latter, it can be resolved as usual, i.e., by removing $Domain(P, \varepsilon)$ for Ω_V, or by adding $PS(P), CS(\varepsilon)$ for Ω_V (since $PS(P) \in D$, the latter is reduced to the addition of $CS(\varepsilon)$ only).

The set of minimal deltas that resolve the violation of a constraint instance $IC(\vec{a})$ in an RDF(S) DB D will be denoted by $Res(IC(\vec{a}), D)$ and called its *resolution set*. In our example, for the second step of resolution, $Res(IC_4(P, A), [[D]]^{\Omega}) = \{ [[\delta_{V_1}]]^{\Omega_V}, [[\delta_{V_2}]]^{\Omega_V} \}$.

4 Declarative Repairing Policies

As explained above, a *repairing policy* is expressed through a set of "specifications" that are provided by the curator and determine his preferences regarding the *preferred repair* (e.g., "I want to make minimal update steps during repair; in case of ties, I prefer removals"). In practice, such specifications should be provided by the curator through an adequate *preference elicitation* interface [27], that would allow him to adequately specify his intentions. Preferences are expressed by determining the interesting *features* of a repair (e.g., number of update steps) along with atomic and composite preferences upon these features.

Features capture functional properties of potential repairs, and they could be defined over the repairs themselves or over the repairing deltas; the two approaches are formally equivalent, because repairs and repairing deltas are interdefinable. To see this, take some D and a repair D_r, and set $\delta_r = \langle D_r \setminus D, D \setminus D_r \rangle$; it follows that $D \bullet \delta_r = D_r$; for the opposite, note that for any given delta δ_r, $D_r = D \bullet \delta_r$.

Defining features over repairs is more "result-oriented", focusing on the repair, rather than the changes that led to the repair. This option has the advantage that it allows the direct use of the quality assessment metrics (presented in Deliverable D2.1 [26]) as features for the repair process. On the other hand, defining features (and, thus, preferences) over repairing deltas gives a more "update-centric" view on the preferences, and are based on the idea that specifications are more related to the update steps that led from one RDF(S) DB to the other, rather than the RDF(S) DBs themselves. In the rest of this deliverable, we follow for simplicity the latter approach, as we believe it's more intuitive to define preferences over deltas; nevertheless, as explained above, the two approaches are formally equivalent and our descriptions and framework can be easily recast for the alternative approach if required.

Thus, a feature is a functional attribute over deltas, whose value (usually a number) represents some interesting property of the delta. For example, given a delta $\delta = \langle \delta_a, \delta_d \rangle$, the "number of updates" is a feature defined as follows: $f_{size}(\delta) = f_{additions}(\delta) + f_{deletions}(\delta)$ where $f_{additions}(\delta) = |\delta_a|$, $f_{deletions}(\delta) = |\delta_d|$.

Features are used in *atomic preferences* of the form $P = (f_A, >_P)$ [13, 18, 17], where $>_P$ is a binary relation among the possible values of f_A; intuitively, if $f_A(\delta_1) >_P f_A(\delta_2)$ then δ_1 is preferred over δ_2. For numerical values, $>_P$ can often be stated compactly using aggregate functions, e.g., the expression $Min(f_A)$ states that "the lowest available values for f_A are more preferred than others" and thus δ_1 is more preferred than δ_2 iff $f_A(\delta_1)$ is minimal, but $f_A(\delta_2)$ is not [13].

Atomic preferences can be further composed using operators such as & (prioritized) and $\otimes$ (pareto) [13, 18, 17]. For example, $P_1 \& P_2$ states that P_1 is more important than P_2, so P_2 should be considered only for values which are equally preferred with respect to P_1. Thus, the preference stated in the beginning of this section can be formalized as $P = P_1 \& P_2$, where $P_1 = Min(f_{size})$ and $P_2 = Max(f_{deletions})$.

Given an atomic or composite preference expression, we can trivially induce an order $(\Delta, >)$ over deltas [13, 18, 17]. Recall that depending on the employed strategy (LO/GO), a curator's preferences are applied either after each individual resolution (in each node of the resolution tree) or at the end of the repairing process (in each leaf of the resolution tree), so a repairing policy must also specify the strategy (GO/LO) to follow. To denote this fact, we will use the symbol $>^G$ ($>^L$) to denote a GO (LO) policy defined by an order $(\Delta, >)$. Features and preferences can be easily generalized to support deltas with labeled nulls.

For the LO strategy, an RDF(S) DB D' is a *preferred repair* for D iff it is valid and there is some sequence of *locally* optimal deltas, whose sequential application upon D leads to D'. Given that the constraint resolution order matters, we consider some arbitrary, but fixed evaluation order for constraints, encoded as a *violation selection function* ($NextV$), which determines the next violation to consider.

Definition 1 *Consider a LO repairing policy $>^L$, a set of integrity constraints $\mathcal{I}$, an RDF(S) DB D, and a violation selection function $NextV$. A sequence of RDF(S) DBs $SEQ = \langle D_0, D_1, \ldots \rangle$ is called a* preferred repairing sequence *of D for $>^L$ iff:*

1. $D_0 = D$.
2. *If $D_i \vdash \mathcal{I}$ then $D_{i+1} = D_i$, else $D_{i+1} = D_i \bullet \delta$, where $\delta \in Res(NextV(D), D)$ and there is no $\delta' \in Res(NextV(D), D)$ such that $\delta' > \delta$.*

We say that SEQ terminates after n steps *iff $D_n = D_{n+1}$ and either $n = 0$ or $D_{n-1} \neq D_n$. An RDF(S) DB D' is a* preferred repair *of D for $>^L$ iff there is some preferred repairing sequence which terminates after n steps and $D' = D_n$.*

The analysis in Section 2.1 can be repeated in formal terms using preferred repairing sequences; the end result (D_{111} in Section 2.1 and Figure 2.1) corresponds to the third element of the (only) preferred repairing sequence for the given preference, $P = Min(f_{size})\&Max(f_{deletions})$, and $NextV$. It is easy to generalize Definition 1 to support RDF(S) DBs/deltas with labeled nulls.

For the GO strategy, an RDF(S) DB D' is a *preferred repair* for D iff it is valid and there is some delta which is *globally* optimal per $>^G$, and whose application upon D leads to D'. In GO, the cumulative effect of all resolutions is considered.

When considering policies adopting the GO strategy, there is a subtle issue deserving further clarification. If we restrict ourselves only to optimality defined by the policy ($>^G$) we may not always obtain *useful repairs*. For instance, consider the GO resolution tree of Figure 2.1 and the preference $Min(f_{deletions})$. Under this preference, the optimal repairs occur from the branches D_{22}, D_{23}, each of which causes no deletions. Consider now the RDF(S) DB: $D_I = D_{22} \cup \{CS(C)\}$. Note that D_I also causes no deletions, and is valid, so it should be equally preferred to D_{22}, D_{23}. However, D_I contains one extra fact ($CS(C)$), which did not exist in the original RDF(S) DB and is totally irrelevant to the resolution process (i.e., it was arbitrarily added, without being dictated by the resolution of some violation). Thus, it would be irrational to return D_I as a preferred repair. To avoid such cases, we put an additional requirement, namely that the delta leading to the preferred repair should be *useful*, i.e., minimal with respect to the *subset relation*. This requirement is in accordance to the minimality notion established in [1, 8]. This issue does not arise in LO strategies, because the sequence of deltas that leads to a preferred repair contains (by definition) only updates that are actually dictated by the violation resolution.

Definition 2 *Consider a GO repairing policy $>^G$, a set of integrity constraints $\mathcal{I}$ and some RDF(S) DB D. A delta $\delta = \langle \delta_a, \delta_d \rangle$ is a* preferred repairing delta *of D for $>^G$ iff:*

1. *$D \bullet \delta \vdash \mathcal{I}$.*
2. *The δ is useful, i.e., there is no $\delta' = \langle \delta'_a, \delta'_d \rangle$ such that $D \bullet \delta' \vdash \mathcal{I}$, $\delta'_a \subseteq \delta_a$, $\delta'_d \subseteq \delta_d$ and $\delta \neq \delta'$.*
3. *There is no δ' satisfying the above two requirements for which $\delta' >^G \delta$.*

An RDF(S) DB D' is called a preferred repair *of D for $>^G$ iff there is some preferred repairing delta δ of D for $>^G$ such that $D' = D \bullet \delta$.*

The GO part of Figure 2.1 in Section 2.1 shows the various potential repairs for D (as leaves in the resolution tree); it is easy to see that, for the preference $P = Min(f_{size})\ \&\ Max(f_{deletions})$, the only preferred one is D_{12}. Again, Definition 2 can be easily generalized to take into account RDF(S) DBs/deltas with labeled nulls.

In Section 2 it was shown that the syntax of the constraints affects the preferred repairs in LO. This is not true for the GO strategy:

Proposition 1 *Consider two sets of integrity constraints $\mathcal{I}$,$\mathcal{I}'$ such that $\mathcal{I} \equiv \mathcal{I}'$ and a repairing policy $>^G$. Then D' is a preferred repair of D per $>^G$ for the constraints $\mathcal{I}$ iff it is a preferred repair of D per $>^G$ for the constraints $\mathcal{I}'$.*

To compare existing repair approaches with our framework, we will generically model a *repair finding algorithm* as a function RF that, given an RDF(S) DB, returns a non-empty set of valid RDF(S) DBs; formally, $RF : \mathcal{D} \mapsto 2^{\mathcal{D}} \setminus \emptyset$, such that for all D and all $D' \in RF(D)$ it holds that $D' \vdash \mathcal{I}$. Our objective is to characterize exactly the properties that a repair finding algorithm must satisfy in order to be *expressible* by some policy $>^G$ or $>^L$. Formally, a repair finding algorithm RF will be called *GO-expressible* (*LO-expressible*) iff there is some repairing policy $>^G$ ($>^L$) such that for all RDF(S) DBs D it holds that $D' \in RF(D)$ iff D' is a preferred repair for D per $>^G$ ($>^L$). The following propositions prove the generality of our framework:

Proposition 2 *A repair finding algorithm RF is GO-expressible iff $D_r \in RF(D)$, $D'_r \vdash \mathcal{I}$, $D'_r \setminus D \subseteq D_r \setminus D$ and $D \setminus D'_r \subseteq D \setminus D_r$, implies $D'_r = D_r$.*

Proposition 3 *A repair finding algorithm RF is LO-expressible iff $RF(D) = \{D\}$ when $D \vdash \mathcal{I}$ and there is a family of repair finding algorithms $\{RF_0^{IC(\vec{a})}\}$ such that $RF_0^{IC(\vec{a})}$ is GO- expressible and considers only one integrity constraint, namely $\{IC(\vec{a})\}$, and $RF(D) = \bigcup_{D_0 \in RF_0^{IC(\vec{a})}(D)} RF(D_0)$ where $IC(\vec{a}) = NextV(D)$, when $D \nvdash \mathcal{I}$.*

Proposition 2 is quite general and implies that a repair finding algorithm is GO-expressible iff it returns useful repairs. Similarly, Proposition 3 captures the recursive and "memory-less" character of LO strategy: we select a violated constraint ($IC(\vec{a}) = NextV(D)$), repair it in an optimal manner ($RF_0^{IC(\vec{a})}(D)$), then start over. The discrimination between valid and invalid RDF(S) DBs in Proposition 3 is necessary, because for a valid D, $NextV(D)$ is not defined. These loose requirements suggest that our framework can capture most existing (and future) repairing policies; in Section 8, we review some existing algorithms that are reducible to our framework.

5 Repair Finding Algorithms

Our framework is implemented using the $Repair$ function, which gets the repairing policy $>^P$ (i.e., $>^G$ or $>^L$) and an RDF(S) DB D in the input, and returns the preferred repairs. $Repair$ is simple and omitted; its only use is to initialize some variables and call the corresponding function (GO or LO, see Algorithms 1, 2) depending on the used strategy; for the case of GO, it also filters potential repairs to get the preferred ones.

As labeled nulls may be inserted during the recursive repairing process, the input to GO (Algorithm 1) will be the current RDF(S) DB $[[D_c]]$ and the delta that has been computed so far $[[\delta_{tot}]]$.

The computation takes 3 different paths depending on the diagnosis. If there exists some constraint which is violated for all possible replacements of labeled nulls with constants in $[[D]]$ (lines 1-8), then each delta in $Res(IC(\vec{a}), [[D_c]])$ constitutes a resolution option and is applied, in different resolution branches, upon D_c (using $\bullet$, see line 6), but also upon $[[\delta_{tot}]]$ (using $\uplus$, see lines 4,5) to get the cumulative delta that has been applied so far. If for some delta $\delta = \langle \delta_a, \delta_d \rangle$, it holds that $\delta_a \cap \delta_d \neq \emptyset$, then for $\delta' = \langle \delta_a \setminus \delta_d, \delta_d \rangle$ it holds that $D \bullet \delta = D \bullet \delta'$, so δ cannot be a useful delta (cf. Definition 2). The same idea applies for deltas with labeled nulls, the only difference being that for such deltas, the problem may exist only partly (i.e., for a subset of the labeled nulls' range). The function $clean\_delta$ "cleans" $[[\delta_{tot}]] \uplus [[\delta]]$ from such deltas; if all deltas are filtered out, $null$ is returned (this check also protects from recursive loops). If $clean\_delta$ does not return $null$, GO is called again recursively for the new RDF(S) DB and delta in order to explore the branch further, but if $null$ is returned, the current delta (in the FOR loop) cannot lead to any useful deltas and is ignored.

The second case appears when $[[D_c]]$ only "partly violates" a certain constraint, i.e., the constraint is violated only for some assignments of labeled nulls to constants (lines 10-12). In this case, the discrimination between violating and non-violating members of $[[D_c]]$ is performed, and a new instance of GO is recursively called.

Finally, if $[[D_c]]$ is valid to begin with (lines 14-15), then one potential solution has been found, so $[[\delta_{tot}]]$ is added to RD and non-useful deltas are filtered out (non-preferred deltas will be filtered out later, in $Repair$).

Note that, for the GO algorithm, the resolution order of the violated constraint instances does not affect the result, as we will show later; however, we have found that selecting first constraints with small resolution sets gives, on average, smaller recursion trees and better performance.

Algorithm 1: $GO([[D_c]], [[\delta_{tot}]])$

1: **if** there exists $IC(\vec{a})$ such that $D \nvdash IC(\vec{a})$ for all $D \in [[D_c]]$ **then**
2: **for all** $[[\delta]] \in Res(IC(\vec{a}), [[D_c]])$ **do**
3: **if** $clean\_delta([[\delta_{tot}]] \uplus [[\delta]]) \neq null$ **then**
4: $[[\delta_{tot}]] = clean\_delta([[\delta_{tot}]] \uplus [[\delta]])$
5: $GO([[D_c]] \bullet [[\delta]], [[\delta_{tot}]])$
6: **end if**
7: **end for**
8: **else**
9: **if** there exists $IC(\vec{a})$ and $D \in [[D_c]]$ such that $D \nvdash IC(\vec{a})$ **then**
10: $GO([[D_c]]^{\Omega_V}, [[\delta_{tot}]])$
11: $GO([[D_c]]^{\Omega_{NV}}, [[\delta_{tot}]])$
12: **else**
13: $RD = RD \cup \{[[\delta_{tot}]]\}$
14: $RD = RD \setminus \{[[\delta]] \mid [[\delta]]$ is not useful $\}$
15: **end if**
16: **end if**

The locally-optimal strategy, LO (Algorithm 2), is very similar. Note that, for LO, the next constraint to consider is determined using $NextV$ (line 2), rather than arbitrarily; this requires the check of line 1 as $NextV$ is undefined for valid DBs. From that point, the computation takes three alternative paths, as in GO.

If there exists some constraint which is violated for all possible replacements of labeled nulls with constants in $[[D]]$ (lines 4-11), we filter the deltas that resolve said constraint using $>^L$ and apply them upon $[[D_c]]$ using

•. Line 7 checks whether the current RDF(S) DB has been considered before; if so, then, due to the "oblivious" character of LO, the same sub-tree will be reproduced, so the processing of this branch need not continue. The function $clean\_DB$ (details omitted) takes in its input the current RDF(S) DB and all the RDF(S) DBs that have been considered in previous recursion steps and detects such loops. For RDF(S) DBs with labeled nulls, only some of the RDF(S) DBs in $[[D_c]]$ may have been considered before; in this case, $clean\_DB$ adjusts the range accordingly, returning $null$ if all RDF(S) DBs have been considered. If $null$ is returned, then this resolution option need not be considered. Otherwise, $prev\_DB$ is updated, and LO is recursively called for the new $[[D_c]]$.

As in GO, the second case appears when $IC(\vec{a})$ is violated only for some assignments of labeled nulls to constants in $[[D_c]]$ (lines 14-15). In this case, the discrimination between violating and non-violating members of $[[D_c]]$ is performed, and a new instance of LO is recursively called.

In case $[[D_c]]$ is valid to begin with (third case), then it is added to the list of preferred repairs (PR) in line 18.

Algorithm 2: $LO([[D_c]])$

```
1:  if [[D_c]] ⊮ I then
2:     Set IC(a⃗) = NextV([[D_c]])
3:     if D ⊮ IC(a⃗) for all D ∈ [[D_c]] then
4:        RD = {preferred (per >^L) deltas in Res(IC(a⃗), [[D_c]])}
5:        for all [[δ]] in RD do
6:           [[D_c]] = [[D_c]] • [[δ]]
7:           if clean_DB([[D_c]], prev_DB) ≠ null then
8:              [[D_c]] = clean_DB([[D_c]], prev_DB)
9:              prev_DB = prev_DB ∪ {[[D_c]]}
10:             LO([[D_c]])
11:          end if
12:       end for
13:    else
14:       LO([[D_c]]^{Ω_V})
15:       LO([[D_c]]^{Ω_NV})
16:    end if
17: else
18:    PR = PR ∪ {[[D_c]]}
19: end if
```

The above algorithms can be shown to return the correct result according to the policy $>^G$ or $>^L$. This result, combined with the fact that the order in Algorithm 1 is not specified, implies that the constraints' evaluation order is not relevant for the GO strategy. Formally:

Proposition 4 *Consider a set of constraints $\mathcal{I}$, a GO or LO repairing strategy $>^P$ and an RDF(S) DB D, and suppose that the call $Repair(D, >^P)$ terminates with output PR. Then the preferred repairs of D, for the policy $>^P$ are exactly the RDF(S) DBs in PR.*

5.1 Complexity Analysis

The complexity of our algorithm is basically determined by the size of the corresponding resolution tree (say NOD) that needs to be constructed and the cost of each individual recursive call. In our subsequent analysis we assume a fixed number of constraints in $\mathcal{I}$, the largest of which has size N_r; the number of universally (existentially) quantified variables in a constraint are at most N_x (N_y); the original RDF(S) DB has size N_D and contains c different constants.

We examined various types of DED constraints that have been used in the literature (see Table 5.1); the reader is referred to [1, 10] for more details. Table 5.1 illustrates the complexity bounds for the height (H) and

Table 5.1: Height and Width of Resolution Tree

Type of Constraints	**Height (H)**		**Width (W)**	
	Acyclic	**Cyclic**	**With Labeled Nulls**	**Without Labeled Nulls**
FD, CFD	O(N_D)	–	O(1)	O(1)
EGD, Denial	O(N_D)	–	O(N_r)	O(N_r)
Full TGD, Full DED	O($N_D{}^{N_r}$)	O(c^{N_x})	O(N_r)	O(N_r)
IND, CIND	O(N_D)	O(c^{N_x})	O(c^{N_y})	O(N_r)
LAV TGD	O(N_D)	O(c^{N_x})	O(c^{N_y})	O(N_r)
TGD, DED	O($N_D{}^{N_r}$)	O(c^{N_x})	O(c^{N_y})	O(N_r)

Table 5.2: Worst-case Complexities

$T_>(n,k)$	$O(k \cdot n \cdot T_f) + O(min\{n \cdot log^{(k-2)}n, k \cdot n^2\})$
T_{GO}	$O((N_D + H)^{N_r} + W^H \cdot H \cdot N_r)$
T_{LO}	$O((N_D + H)^{N_r} + W \cdot (N_r + H \cdot (N_D + H))) + T_\otimes(W,k)$
NOD	$O(W^H)$
T_{Repair}	$max\{NOD \cdot T_{GO} + T_\otimes(W^H,k), NOD \cdot T_{LO}\}$

width (W) of the resolution tree for these types. H is equal to the maximum number of invalidities in a branch, and includes invalidities caused by the resolution process. W is equal to the maximum size of the resolution set for each type of constraint; for constraints with existential quantifiers, labeled nulls significantly reduce the width, as expected. The worst-case size of the resolution tree (NOD) is $O(W^H)$ (Table 5.2).

Since the maximum number of tree nodes is finite, the algorithm will terminate. Note that the same worst-case results hold for both GO and LO. However, LO may have, on average, smaller recursion trees, depending on the aggressiveness of the pruning imposed by the preference (see also Section 6), which may even reduce the tree to a chain.

Table 5.2 shows the complexity of comparing n deltas using a preference $>$ composed of k atomic preferences ($T_>(n,k)$), where T_f is the cost of computing one feature for one delta. $O(k \cdot n \cdot T_f)$ is the cost of computing all features, and dominates the cost of comparing feature values in each atomic preference, under the reasonable assumption that atomic preferences use a total order (e.g., Min). The second additive in $T_>(n,k)$ is the cost of composing preferences using $\otimes$ [14]; if & is used, the cost is lower.

Table 5.2 also shows the computational cost for algorithms 1 (T_{GO}) and 2 (T_{LO}). These include the cost to determine validity ($O((N_D + H)^{N_r})$). The total repairing cost (T_{Repair}) is computed by multiplying NOD with T_{GO} or T_{LO}. Note that for the GO case we have an additional cost $T_\otimes(W^H,k)$ (in $Repair$) to compare all the potential repairing deltas ($O(W^H)$ in total).

As the formulas show, the complexity is analogous to the tree size, which, in turn, is exponential to the tree height H (number of violations). In the worst-case scenario, this is analogous to the DB size N_D (i.e., it is $O(N_D)$, but, in practice, the number of violations is not directly related to N_D, because the quality of an RDF(S) DB is unrelated to its size. Thus, the above worst-case complexity is misleading as to the complexity exhibited in practice by the algorithm; this will also be demonstrated in our experimental evaluation (Section 6).

5.2 Optimizations

We can exploit the requirement for useful repairs in Definition 2 to prune certain branches. To grasp the intuition, consider the motivating example of Section 2 and the GO part of Figure 2.1. Consider the branch that led to D_{131}: in the last step, we resolve $IC_4(P,A)$ by adding $CS(A)$. The same operation (adding $CS(A)$) would have resolved the first violated constraint in the resolution tree, $IC_1(B,A)$, but the branch under consideration

chose an alternative resolution (removal of $C\_IsA(B, A)$). Since the branch of D_{131} eventually led us to add $CS(A)$ anyway, we could as well had chosen to do so to resolve $IC_1(B, A)$ in the first place; this would have led to a "smaller" delta. Thus, D_{131} could not have been a preferred repair, because its corresponding delta is not useful. More generally:

Proposition 5 *Consider an RDF(S) DB D, a GO policy $>^G$ and a leaf node L in the resolution tree, whose input was $D_L, \delta_L = \langle \delta_{aL}, \delta_{dL} \rangle$. Then the following are equivalent:*

1. *There is a node D (with input D_D, δ_D) in the same branch as L, such that $IC(\vec{a})$ was resolved in D, and $\delta_1 = \langle \delta_{1a}, \delta_{1d} \rangle$, $\delta_2 = \langle \delta_{2a}, \delta_{2d} \rangle$ such that $\delta_{1a}, \delta_{2a} \subseteq D_L$, $\delta_1, \delta_2 \in Res(IC(\vec{a}), D_D)$ and $\delta_{1d} \cap D_L = \delta_{2d} \cap D_L = \emptyset$.*
2. *There is another leaf node, say L' (with input $D_{L'}, \delta_{L'} = \langle \delta_{aL'}, \delta_{dL'} \rangle$) for which $\delta_{aL} \subseteq \delta_{aL'}$, $\delta_{dL} \subseteq \delta_{dL'}$.*

Proposition 5 implies that, if, for some node (leaf or other), it holds that the current delta (δ_{tot}) contains the ground facts from two deltas (δ_1, δ_2) that could resolve a constraint that was previously resolved, then we know that all the leaf nodes that will result from this node will satisfy the first condition of Proposition 5. This gives us the option to prune entire branches as depicted in Figure 5.1. In case 1, we follow the resolution option δ_1 in the first step of the resolution process; then, in the second step, we follow δ_2. Both δ_1, δ_2 are now "included" in δ_{tot}, and both are resolution options for a previously considered constraint. Thus, this branch must be rejected. In case 2, the same situation is shown, but then, in another branch, the same pair (δ_1, δ_2) appears in a different order. As shown in Figure 5.1, the second branch is processed normally, because it may correspond to the case where $\delta_L = \delta_{L'}$ (see above). This optimization also allows us to avoid the checks for useful deltas (line 11 of GO algorithm), because, if the first bullet of Proposition 5 does not hold, then the delta is certainly useful.

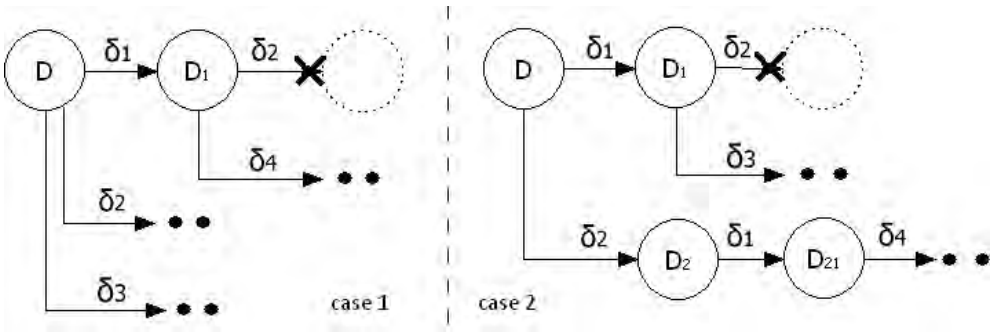

Figure 5.1: Examples of Resolution Tree Pruning

Another optimization exploits the fact that one validity check can identify several violations, whereas only one violation is resolved in each recursive node. Thus, validity checks need not be performed in every node. Moreover, in subsequent checks, any detected violations did not exist in the original RDF(S) DB, so they were introduced by the resolution of previously considered violations; thus, we can reduce the number of integrity constraints that need to be checked to those that previous resolutions could violate. This optimization applies to both GO and LO.

These optimizations do not jeopardize the algorithms' correctness. Also, they can be combined with heuristics that affect, e.g., the constraint resolution order for the GO algorithm (constraints with few resolution options take priority), or preference-specific constraints (for fixed preferences).

6 Experimental Evaluation

6.1 Experimental Setting

To evaluate the performance of our algorithms, we relied on both real and synthetically generated DBs and violations. The experiments using synthetic RDF(S) DBs studied the impact of critical parameters (e.g., number of violated constraints, preference expressions) on the execution time, as well as on the quality of the obtained repairs. The experiments using real datasets focused on performance under specific run-time parameters and showed that repairing is feasible even for large datasets containing several invalidities. However, for certain datasets, feasibility of repairing is achieved only using the LO strategy.

To cover a more representative variety of DEDs, we extended constraints $IC_1 - IC_4$ (see Section 2) with the following:

$IC_5 : \forall x_1, x_2 C\_IsA(x_1, x_2) \wedge C\_IsA(x_2, x_1) \rightarrow \perp$

$IC_6 : \forall x_1, x_2, x_3 Domain(x_1, x_2) \wedge Domain(x_1, x_3) \rightarrow (x_2 = x_3)$

$IC_7 : \forall x_1, x_2, x_3 Range(x_1, x_2) \wedge Range(x_1, x_3) \rightarrow (x_2 = x_3)$

$IC_8 : \forall x_1, x_2, x_3, x_4 PI(x_1, x_2, x_3) \wedge Domain(x_3, x_4) \rightarrow CIns(x_1, x_4)$

$IC_9 : \forall x_1, x_2, x_3, x_4 PI(x_1, x_2, x_3) \wedge Range(x_3, x_4) \rightarrow CIns(x_2, x_4)$

In the above constraints, $PI(x_1, x_2, x_3)$ denotes that the pair (x_1, x_2) is a direct or transitive instance of property x_3, and $CIns(x_1, x_2)$ denotes that x_1 is a direct or transitive instance of class x_2. Constraint IC_5 requires that all subsumption relationships are acyclic while IC_6, IC_7 ensure that properties have a unique domain and range. Constraints IC_8, IC_9 require that the subject/object of a property instance must respect the corresponding property's domain/range.

We made five sets of experiments in total, the first four of which were based on synthetic datasets, whereas the last used two real-world ontologies, namely Gene Ontology and DBpedia. In the first set of experiments, we show that the size of the DB has a linear effect on the execution time, other things being equal. In the second set, we study the effect of the size of the resolution tree (linear) and the number of violations (exponential) on the execution time. The third experiment using synthetic datasets shows the catalytic effect that the preference function may have on the execution time of the LO algorithm. The fourth experiment shows examples of synthetic data where the LO and GO algorithms give different repair results. The last experiment (that used Gene Ontology and DBpedia) showed that large real-world ontologies can be repaired using our technique.

Our synthetic inputs were created using PowerGen [34], which allows us to create synthetic RDF(S) DBs featuring realistic, but different structural characteristics. Given that PowerGen generates valid RDF(S) DBs, we rely on a *violation insertion algorithm* (VIA) to create invalidities. In particular, VIA randomly adds a specific number of invalidities related to selected constraints, as specified by its parameters. For the first three experiments, we created six RDF(S) DBs with $100K$, $200K$, $1M$, $2M$, $10M$, $20M$ of triples, balancing between the number of classes, properties and their instances. Then we applied VIA to violate constraints $IC_5 - IC_7$ which do not exhibit interdependencies from others and thus allow us to control the size of the resolution tree, which seriously impacts the performance of our algorithms. We scaled from 1 to 20 the number of initially introduced invalidities. For the fourth experiment, we generated an RDF(S) DB with approximately 1500 triples, whose emphasis is more on classes and class/property instances. We employed VIA to violate the constraints ($IC_8 - IC_9$) which interfere with $IC_1 - IC_7$, so our analysis in Section 6.2 takes into account the actual violations produced, not just the initial ones introduced by the VIA. Note that this indirect introduction of violations is not experimentally studied in related work [5, 9], so this is a novel contribution.

The last experiment used Gene Ontology and DBpedia as inputs. The Gene Ontology is an RDF(S) DB dealing with biological data, whose size and update rate make it one of the largest and most representative datasets for studying data management problems in ontologies. This ontology describes gene products in terms of their associated biological processes, cellular components and molecular functions in a species-independent

manner. It is composed of circa 28000 classes, that are all instances of one meta-class and are organized in a complicated hierarchy, and 1350 property instances of a single property ($obsolete$) which is, sometimes, used by the editors to mark classes as obsolete as an alternative to deleting them; the total number of triples is around $200K$. Although Gene Ontology is provided in RDF/XML format, the subsumption relationships between classes are represented by user-defined properties instead of the standard $rdfs : subClassOf$ property, so we used the versions released by the UniProt Consortium [2], which use RDFS semantics. Gene Ontology is updated on a daily basis, but UniProt releases a new version every month. For our experiments we retrieved 3 versions from UniProt (dated 16.12.08, 22.09.09 and 20.04.10). DBpedia is a community effort to extract structured information from Wikipedia and to make this information available on the Web, and is a central ontology in the Linked Data cloud. The DBpedia project describes $3,5$ million resources based on 843.000 articles from Wikipedia leading to 29 billion RDF triples. For our experiments, we loaded the largest fragment of DBpedia that could fit in memory, which contained approximately $10M$ triples. Note that Gene Ontology and DBpedia were already violating some of the constraints, so VIA was not used in this experiment. More details on the above experiments will appear in the sections below.

All our experiments were performed on an Intel Xeon machine at 2.33GHz with 16GB of memory, running Open Suse. To smoothen the effects of the randomized VIA, the first three experiments were run 40 times, and the averages were taken.

6.2 Effect of DB Size

To perform this experiment, we used the 6 large synthetic RDF(S) DBs, introduced 10 and 20 violations in each and ran both the GO and the LO algorithm, resulting in a total of 4 experiments, each of which had a fixed number of violations and resulting tree size. In each experiment, we measured separately the diagnosis time (finding existing violations), and repair time (finding and applying potential resolutions). Figure 6.1 shows how the diagnosis and repair time (in logscale) correlate with the RDF(S) DB size (in logscale).

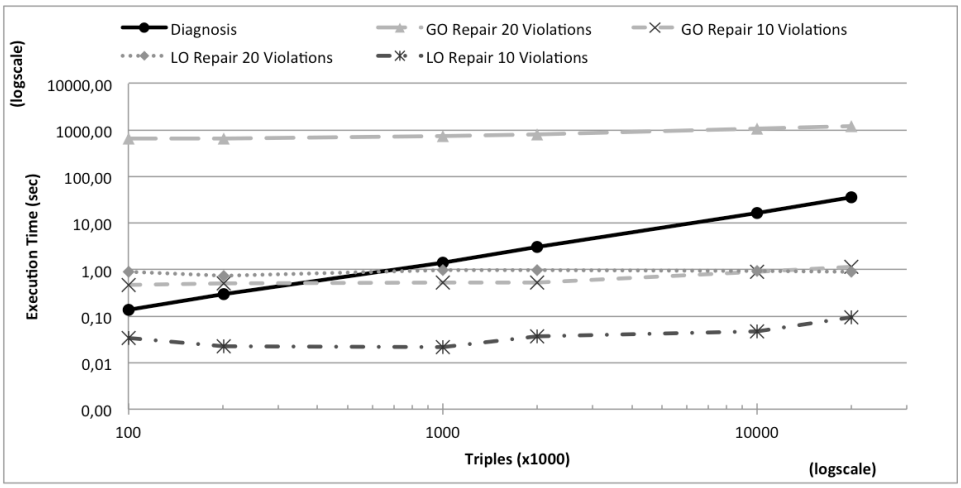

Figure 6.1: Size of the DBs and Execution Time

Due to the optimization described in Section 5.2, diagnosis was run a fixed number of times in all 4 cases; as a result, it had a fixed execution time, regardless of the number of violations, and appears once in the graph. Diagnosis exhibits a linear correlation with the RDF(S) DB size. On the other hand, the repair time is mainly affected by the algorithm used (GO/LO) and the number of violations (10 or 20). The RDF(S) DB size has a small, linear effect on the execution time.

These results seem to contradict the complexity analysis of Table 5.2, but this is not the case. The linear time of diagnosis is due to a series of implementation-specific optimizations. For repair, a closer look at Table 5.2 shows that the complexity is exponential with respect to the number of violations (i.e., the height of the tree, H),

which in the worst-case scenario can be analogous to the RDF(S) DB size (see Table 5.1); in practice however, the size of the RDF(S) DB is not directly related to the number of violations.

Another interesting observation is that the diagnosis time often dominates the repair time, depending on the tree size (which determines the repair time) and the size of the RDF(S) DB (which determines the diagnosis time). Thus, for large RDF(S) DBs, the diagnosis, rather than the repair, is the dominant factor in performance.

Regarding memory requirements, our experiments showed that the repair process itself requires around 350-400MB of memory for the aforementioned runs (with a maximum of 420MB), whereas the rest is devoted to the storage of the RDF(S) DB.

6.3 Effect of Violations

Figure 6.2 shows how the execution time is affected by the size of the resolution tree (linearly) and the number of violations (exponentially), confirming our complexity analysis in Table 5.2. This observation holds for all 6 RDF(S) DBs, but different DB sizes cause different inclinations in the curves, as larger DBs cause larger execution time per node; however, the effect of the DB size is negligible compared to the effect of the number of violations (e.g., the cost of repairing 20 violations in the 100K-triples DB is larger than the cost of repairing 18 violations in the 20M-triples DB). The reported performance figures do not consider any repairing preference, because the computational time for determining the preferred deltas is negligible using efficient algorithms [13].

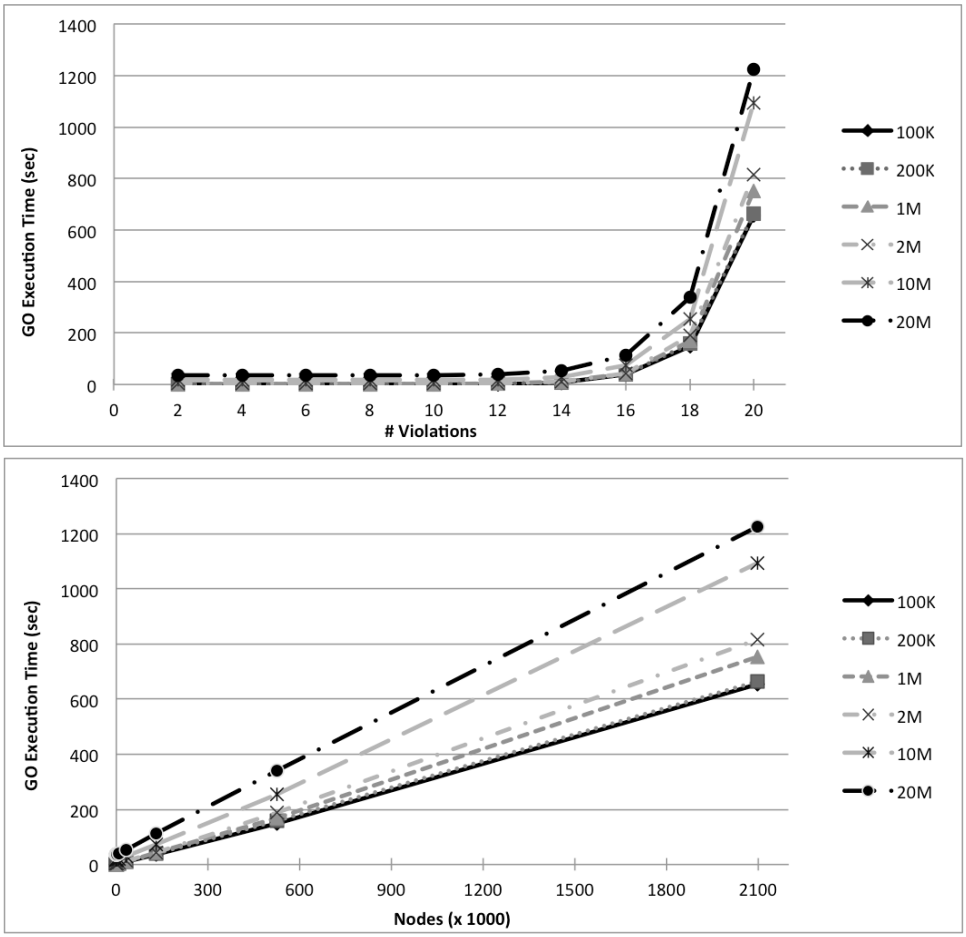

Figure 6.2: Globally-optimal Strategy (GO): Tree Size and Execution Time

For the evaluation of LO algorithm we used a preference function P_2 which filters out one of the two possible resolution options for IC_6, IC_7 and keeps both options for IC_5. Figure 6.3 shows the execution times reported, per tree size and number of violations, for each of the 6 RDF(S) DBs. As explained before (Figure 6.1), the

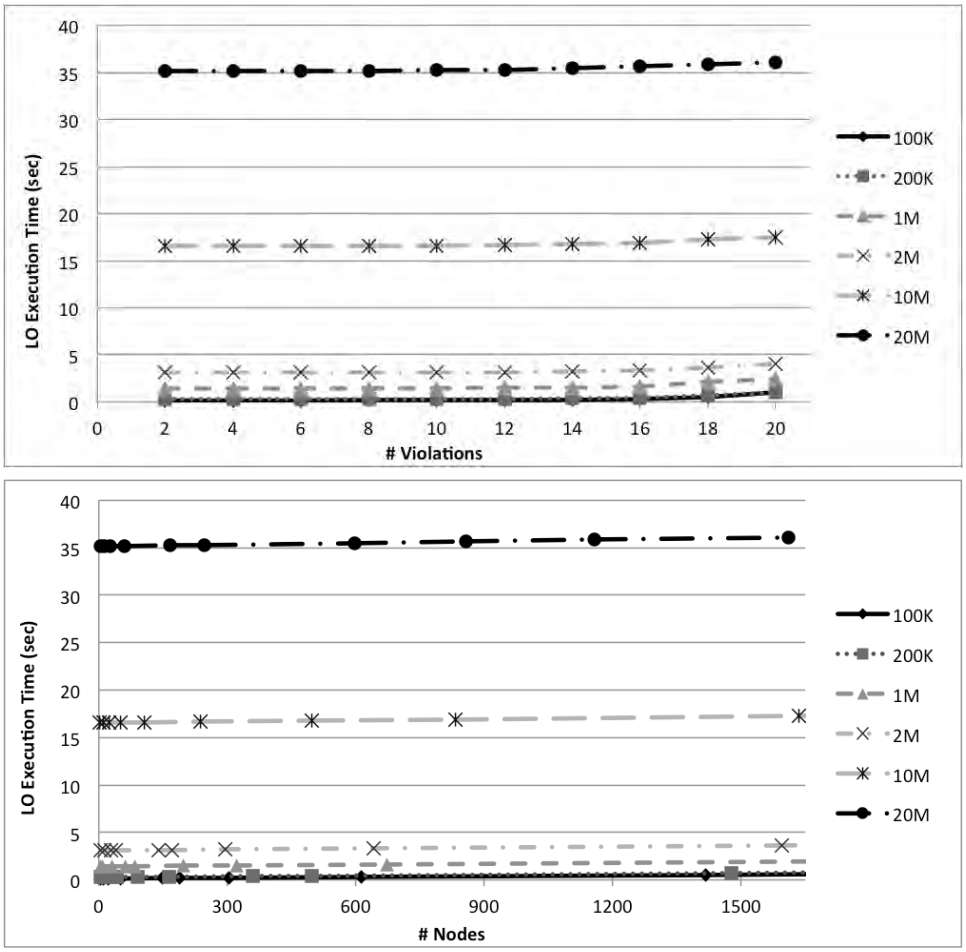

Figure 6.3: Locally-optimal Strategy (LO): Tree Size and Execution Time

diagnosis time dominates performance in LO, and it is fixed per DB size; this explains the very small inclination of the curves in Figure 6.3. In addition, the reported times (and the corresponding tree size) in LO are about two orders of magnitude smaller than in GO, due to the pruning performed.

6.4 Effect of Preference Function (in Locally-optimal Strategy)

Figure 6.4 shows the effect of the preference function (and the related pruning) on LO execution time for the 20M-triples RDF(S) DB. The considered preference functions are: P_0, which considers all resolution options (no pruning); P_1, which is designed to prune one of the two possible solutions when IC_5 is resolved, and performs no pruning in other cases; P_2 which prunes one of the two solutions in IC_6, IC_7, but keeps both in IC_5 (used also in Section 6.3); and P_3, which filters out one of the two possible solutions for constraints $IC_5 - IC_7$. Figure 6.4 shows that P_0, which does no pruning, achieves similar performance as the GO algorithm, whereas more "aggressive" pruning policies (preference functions) give significantly better performance (up to an order of magnitude, depending on the number of violations).

6.5 Quality of Locally-optimal (LO) Repairs

To evaluate the quality of LO repairs, we used a small DB (see Section 6.1 above) and measured the number of globally optimal repairs (returned by GO) which are not returned by the LO (*false negatives*) and the number of preferred repairs returned by LO which are not globally optimal (*false positives*), denoted by $GO \setminus LO$, $LO \setminus GO$

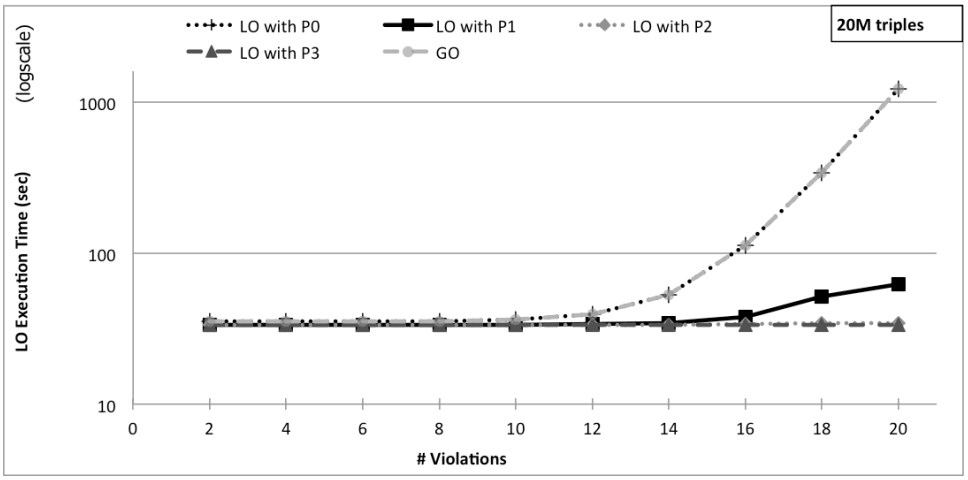

Figure 6.4: Effect of Preference Function (in Locally-optimal Strategy)

respectively in Figure 6.5; common repairs are denoted by $GO \cap LO$. Note that we consider repairs with labeled nulls, i.e., a repair $[[D]]$ is counted as a single repair.

As shown in Figure 6.5, for the preference $Min(f_{additions})$, LO returns several non-globally optimal results (false positives), whereas for $Max(f_{additions})$, LO returns both false positives and false negatives. It should be mentioned that such a behaviour (large number of false positives or negatives) is not an exceptional case and can be easily reproduced whenever constraint interdependencies exist.

The reason for this behaviour is related to the violations of IC_8, IC_9 that were imposed on CCD. For $Min(f_{additions})$, such violations can be optimally resolved by deleting a property instance, but LO will also consider the option to delete a domain or range (causing false positives). For $Max(f_{additions})$, LO will only consider the addition of class instantiations, whereas some of the deletions could also lead to optimal repairs.

6.6 Experiments with Real Datasets

The purpose of the experiments with real datasets was to show the feasibility of applying our repair approach to real-world ontologies. As described above, two different ontologies were used, namely Gene Ontology and DBpedia.

Our evaluation of the Gene Ontology showed that its invalidity problems were related to the non-explicit definition of a domain and/or range in certain properties, which, according to the integrity constraints employed in this work is an invalidity (see IC_2). Repairing these violations using the two proposed strategies (GO and LO) is shown in Table 6.1. A glance at the table verifies some of the results already established for the synthetic ontologies, e.g., the fact that LO is about two orders of magnitude faster than GO.

Table 6.1: Gene Ontology: Results

Gene Ontology Version (date)	**Number of Triples**	**Algorithm and Preference Used**	**Duration for Repair (msec)**	**Number of Violations**
16.12.2008	185.813	GO: Min(Additions)	201.488	24
16.12.2008	185.813	LO: Min(Additions)	3.584	24
22.09.2009	195.124	GO: Min(Additions)	213.314	24
22.09.2009	195.124	LO: Min(Additions)	3.842	24
20.04.2010	210.075	GO: Min(Additions)	238.807	24
20.04.2010	210.075	LO: Min(Additions)	4.173	24

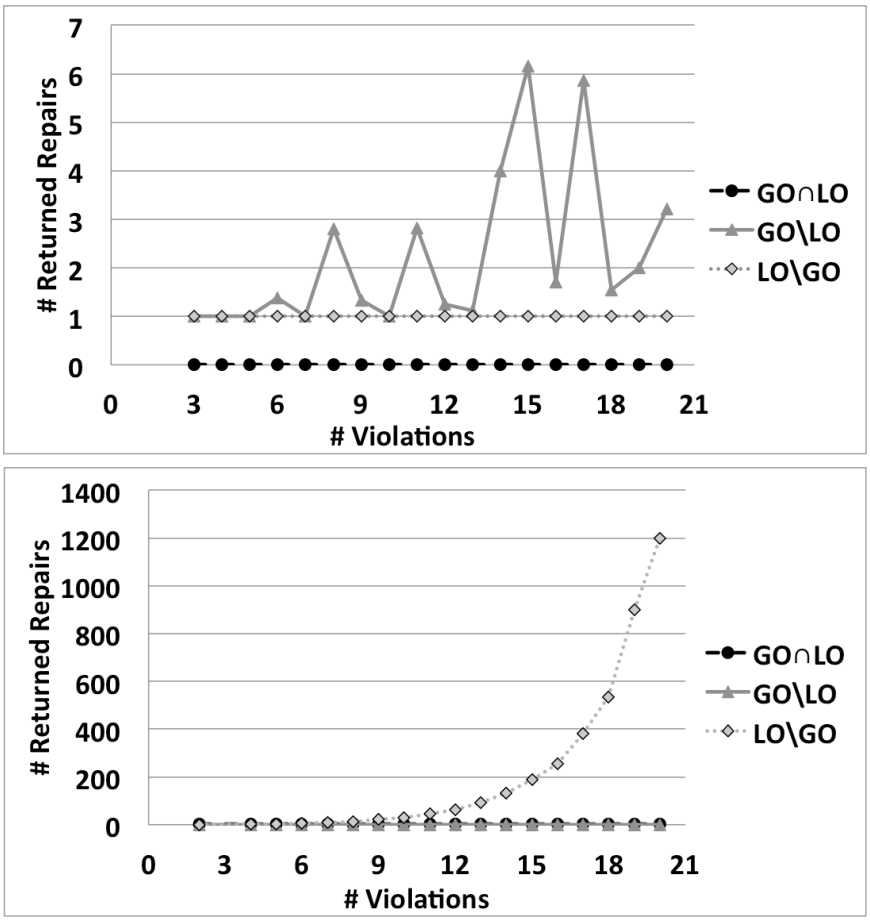

Figure 6.5: Quality of Repairs: Locally-optimal vs Globally-optimal

Most of the violations in DBpedia were also related to the absence of explicitly defined domain/range for certain properties. Due to the large size of DBpedia (9.768.234 triples), and, most importantly, due to the large number of errors that appear in it (644), the GO algorithm failed to produce a result, as the resolution tree grew to a very large size. However, the LO algorithm was able to generate a tree of decent size and compute a repair in less than 40 seconds, as Table 6.2 shows. For the LO algorithm, we used a preference that chooses to repair the missing domains/ranges by assigning as a default domain/range one of the special resources: $rdfs:Resource, rdfs:Class, rdf:Property, rdfs:Literal$ (this is represented by the preference $Min(Deletions)\&Max(Val\_Additions)$ shown in Table 6.2).

Table 6.2: DBpedia Ontology: Results

Number of Triples	**Algorithm and Preference Used**	**Duration for Repair (msec)**	**Number of Violations**
9.768.234	LO: Min(Deletions) & Max(Val_Additions)	39.498	644

7 Interface Issues

An important issue related to the repair process is the interface with the user (curator) who performs the repair. In this respect, there are two major issues that need to be addressed: the first is how to make it easy for the user to provide the input, including the preference to be used in the repair process, while the second is helping the user understand the ramifications of the chosen preference and verifying that the performed repair coincides with the intended repair.

In essence, the former is related to enabling the user to *provide the input* in a user-friendly manner, whereas the latter amounts to helping the user *understand and examine the output*. Understanding the output is very important, because it allows the user to fine-tune the preference and be able to make the automated process as close as possible to a manual one; this is similar to how debuggers help the programmers resolve issues with their code, i.e., identify and correct unintended output.

Figures 7.1, 7.2 show two mockup screen shots for the input and output dialogues respectively. The two sections below provide details on the intended functionality of the interface. It should be noted that the development of the user interface is still incomplete, and further improvements are expected in the next few months (not in the scope of the PlanetData project).

7.1 Providing the Input

In the input dialogue (Figure 7.1), the user should be able to provide the input RDF(S) DB that should be repaired, and, most importantly, the repair methodology to be used, i.e., the strategy (GO or LO) and the corresponding preference. The preference is the most important, so we focus on that. Recall that a preference is composed of (a) features, which are arbitrary functions over deltas (or repairs), (b) aggregate functions that are used to define the atomic preference $>^P$, and, (c) operators ($\&, \otimes$) that are used to compose these atomic preferences.

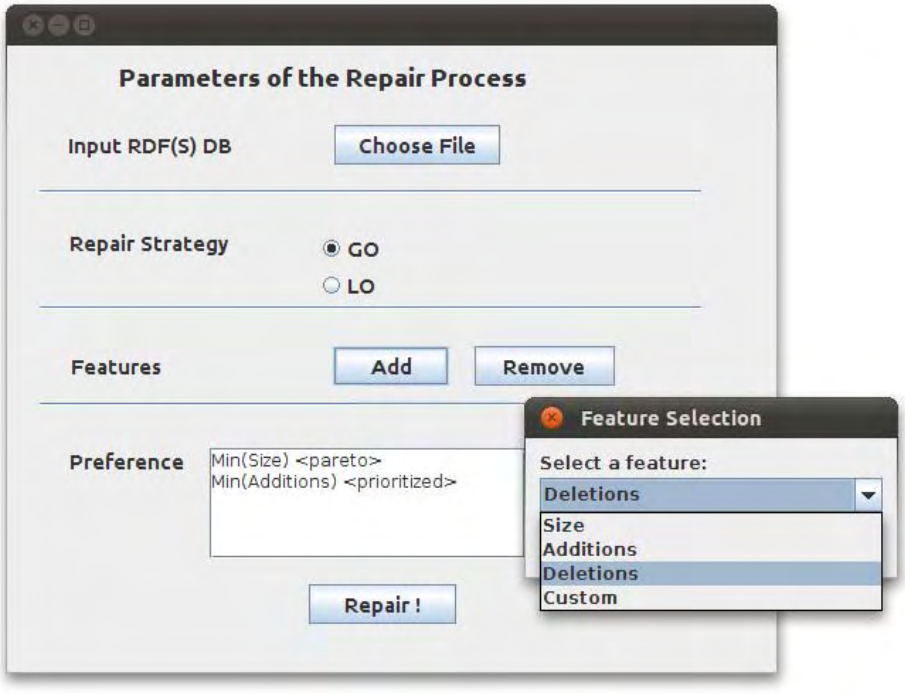

Figure 7.1: Input Dialogue

For the features, the system needs to provide a set of default, useful features that the user can employ in his preferences, like f_{size} or $f_{additions}$ ("Size" and "Additions" respectively in Figure 7.1). However, these will normally not be enough for all purposes, so the user should be allowed to provide his own customized features

as well (option "Custom" in the drop-down list of the pop-up window). These custom features in practice would be external functions, described using user-provided programmatic code and dynamically loaded at run-time.

As explained in Section 4, features are used in atomic preferences of the form $P = (f_A, >_P)$, where $>_P$ is a binary relation among the possible values of f_A. We argue that, for most practical purposes, features will be numerical functions whose values (results) can be ordered using some aggregate function such as those proposed in [13, 17] (Min, Max, $Around$, etc). Under this assumption, the selection of the aggregate function to be used could be made using a drop-down list.

Once the atomic preference (i.e., feature and aggregate function) has been selected, it needs to be composed with other atomic preferences in order to form composite preferences, using the "prioritized" ($\&$) or "pareto" ($\otimes$) operators. The user should be able to use either of the two operators to do so, and also parentheses, in order to specify the order of applying the operators (in the case where more than two atomic preferences are used).

In practice, as shown in Figure 7.1, this is done using an "Add" button, that successively opens sub-dialogues (pop-up windows) allowing the user to select the feature, the aggregate function and the operator to connect the resulting atomic preferences (as well as parentheses, if applicable). The final preference is shown in a text box at the bottom of the dialogue window.

7.2 Examining the Output

Once the repair process is completed, the system should not only return the preferred repair(s) to the user, but also allow him to understand how these preferred repair(s) were generated. To do so, the system should present a detailed view of the preferred repairs, by showing the delta(s) that lead to (each of) the preferred repair(s), the violations that were resolved in the process of repairing, and an association of these violations with the change in each delta that was applied in order to resolve said violation (see Figure 7.2).

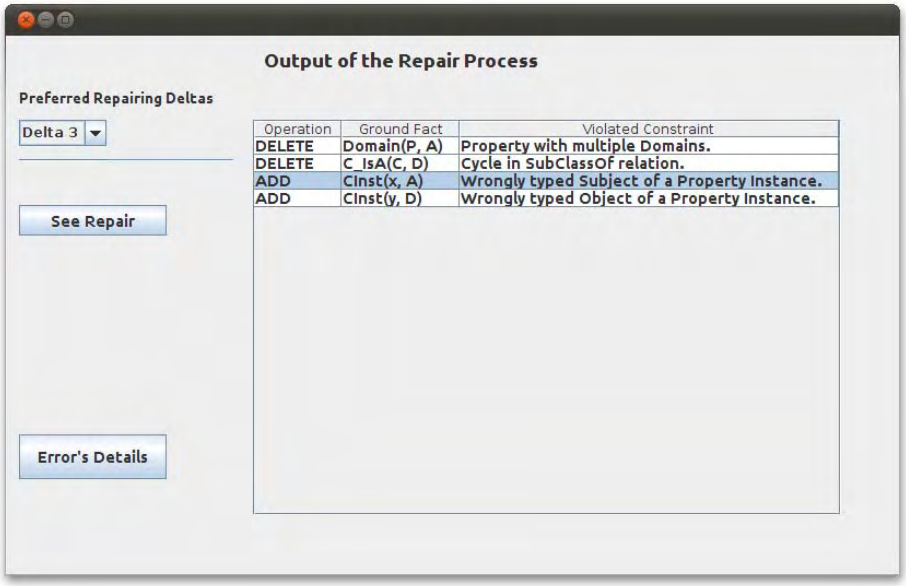

Figure 7.2: Output Dialogue

More specifically, the output dialogue should allow the user to examine each repairing delta in isolation. For each such delta, the list of added/deleted ground facts in the delta should appear, along with the violated constraint whose resolution triggered their inclusion in the delta.

The user should be allowed to select each of those ground facts to find more information (e.g., which were the alternative resolution options, and whether these resolution options were used in some other preferred repairing delta). These details would appear upon clicking on the button "Error's Details" shown in Figure 7.2. Note that, in the case of LO, the selected resolution option would be obvious by the preference, whereas in the case of GO

it might happen that the system chose a resolution that seems less preferred than others; this is a consequence of the definition of the two strategies, as GO will select the deltas that are preferred as a whole, not deltas that contain the most preferred resolutions for each violation in isolation.

In case the user does not approve the resolution choices made by the system, he has the option to rerun the repair with a different preference, or "undo" one (or more) of the resolution options, i.e., manually forcing the system to choose one particular resolution option for a particular violation, overriding the behaviour dictated by the preference itself. This functionality is available again through the detailed checking of each single resolution (button "Error's Details" in Figure 7.2).

8 Related Work

Few declarative approaches for repairing data invalidities have been proposed for standard relational settings. Compared to our work, they address only specific forms of data invalidities which correspond to subtypes of DEDs [10] studied in our framework, while they rely on fixed policies to filter-out potential repairs during violation resolution which cannot be customized at run-time.

One stream of works focuses on deciding whether a given DB is a preferred repair under various policies (*repair checking*); this is a different problem from actually computing a repair (*repair finding*). The various approaches can be further classified depending on whether they are automatic or require user interaction at run-time. The computational complexity of repair checking under various settings was studied in [1, 8]. Four variations of minimality were proposed in [1], all of which can be captured by our framework:

- *Subset repairs* require that the repaired RDF(S) DB should be a maximal (per $\subseteq$) sub-instance of the initial RDF(S) DB, and are captured using a GO preference requiring minimal additions.
- *Symmetric difference repairs* require that the repaired RDF(S) DB should contain the least possible (per $\subseteq$) updates (additions/deletions). Useful repairs are actually symmetric difference repairs, so this notion is captured by GO if no preference is provided.
- *Cardinality repairs* require that the repairing RDF(S) DB should contain the least possible (in terms of cardinality) updates (additions and deletions) and can be captured using a GO preference requiring the delta to have minimum size.
- *Component cardinality repairs* require that the repairing RDF(S) DB should contain the least possible updates (additions and deletions) per relation in terms of cardinality. They can be captured by using one feature per relation, which counts the appearances of said relation in delta; requiring the minimization of all such features, and connecting said preferences with the pareto connective, we get the desired GO preference.

An automated repair finding algorithm has been proposed in [5, 9], where only functional, conditional functional, and inclusion dependencies (FDs, CFDs, INDs) are considered, and each violation is resolved independently depending on the constraint type (a LO strategy). FDs and CFDs are resolved using replacements, whereas INDs are resolved by adding tuples of the form $R(x, null)$, where $null$ is used for the existentially quantified variables in an IND, to avoid considering all possible assignments for such variables. The latter can be easily captured in our framework using a LO preference that favors additions which have $null$ in the existential variables of INDs. We plan to capture the proposed FD and CFD resolution by extending our framework to support replacements.

A more general algorithm which considers all DED types, is found at [6]. The resolution of each violation is made independently (LO policy), by taking all possible resolution options (except replacements), but only the $null$ value is considered for the existentially quantified variables (as in [9]). This policy can be expressed in our framework by a LO preference that filters out additions of tuples without $null$ in said variables.

An interactive approach is proposed in [36], where repairing of CFD violations is performed in independent batches consisting of one automatic and one manual filtering of undesired resolution options, which repeat until no more violations exist. During the process, a learning mechanism records choices as an aid for future repairs.

In the context of ontology debugging, repairs are manually determined by the curator (see e.g., ORE [21], PROMPT [30], Chimaera [24]). Our framework is the first declarative framework that allows curators to intervene in the repairing process by specifying complex preferences over interesting repair features, thus offering richer repairing policies while sparing them from having to make any resolution choices at run-time.

Rondo [25] is actually a generic framework for relational and XML model management that employs a repair finding algorithm for FD constraints. To determine the preferred repair, Rondo uses 9 different "importance classes", and classifies each tuple in one of them based on user-defined information. Again, this policy can be captured by defining 9 features, which count the number of deletions per "importance class"; the final LO preference is a prioritized composition of the minimization of these 9 features.

9 Conclusions and Future Work

This deliverable complements Deliverable D2.1 [26], by studying the problem of *quality repair*, with emphasis on one specific quality dimension, *validity*. Validity is a very important quality dimension, as it represents the application-specific requirements that are imposed on the data, and is therefore critical for the seamless operation of data-centric applications. Due to its importance, imposing validity, via quality repair, is an important problem. Validity is a very generic notion, as the application requirements may take different forms, and it is formalized using constraints in the form of DEDs, which allows the notion of validity to be formally expressed in logical terms.

It should be emphasized that this deliverable deals only with validity repair, but it can be also applied to other quality dimensions, as long as they are expressible using DEDs. Other repair dimensions that cannot be expressed using DEDs (e.g., accuracy) should be handled using specialized approaches that are out of the scope of this deliverable. For example, a specific repair method for identifying (and resolving) errors and outliers in sensor data appears in Deliverable D2.1 [26].

In the present deliverable, we proposed a declarative framework for assisting curators in repairing invalidities that often arise in RDF(S) DBs. Our framework is expressive enough to capture a wide class of integrity constraints (i.e., validity models) using DEDs, and various *preferences* over interesting features of the resulting repairs (i.e., updates' minimality). Such preferences are used to filter out non-minimal repairs, either during the resolution of each individual invalidity (greedy strategy, LO) or at the end, after having resolved all invalidities and acquired all potential repairs (exhaustive strategy, GO); preferences are provided by the curator at the input, through an adequate preference elicitation interface, and essentially determine the results of the repair.

The two strategies (GO/LO) yield different results and exhibit a different behaviour. We implemented the corresponding algorithms and studied their complexity for various types of integrity constraints. We devised adequate optimizations that considerably improve the practical complexity of GO and render it scalable with respect to the size of the repaired RDF(S) DB. In particular, our experimental evaluation showed that for large enough RDF(S) DBs with few invalidities the dominant performance factor is the diagnosis cost, rather than the repair. Moreover, for the LO algorithm, the pruning capacity of the employed preference is crucial for the performance of the repair process as well as for the quality of the repair results. Finally, experiments showed that our approach is feasible also for real-world datasets.

Although we have focused on a specific relational encoding and set of integrity constraints for RDF(S) DBs, the expressiveness of our framework allows us to also exploit the proposed repair finding algorithms in standard relational settings. To this end, we plan to allow *replacements* [9, 5] as resolution options. This extension requires dropping the unique name assumption which is inherent in objects (like URIs of classes or properties) but not in values. As a matter of fact it is not clear whether the replacement of a value in a tuple should be local (i.e., affecting said tuple only) or global (i.e., affecting all appearances of said value). Even though it is easy to introduce replacements in our framework, it turns out that this extension causes an undesirable dependency of the repair result on the constraint evaluation order and on the constraint syntax (currently exhibited only by the LO strategy). Further investigation on this issue is a subject of future work.

GO gains over LO in terms of repairing quality are maximized in cases where several interdependent constraints should be simultaneously enforced. On the other hand, LO is faster, and, given that each branch is totally independent from the others, it can also be easily parallelized for further efficiency improvements. Thus, an interesting future research path would be to formulate conditions (preferences/constraints) under which the GO and LO strategies give the same results, or conditions under which LO gives "nearly optimal" results under some notion of "nearly optimal".

We also plan to devise an anytime repair finding algorithm, whose performance properties would be set by the curator at the expense of repairing quality. Depending on whether we explore the resolution tree in a breadth-first or depth-first manner, this would result in optimal but incomplete repairs, or in non-optimal but complete ones. A similar idea would be to perform tree pruning (i.e., optimality checking) periodically at certain tree depths; such a strategy would strike a compromise between LO (where pruning is done at every node) and GO (where no pruning is done). Such techniques could allow scaling the approach to larger RDF(S) DBs with more constraint violations, a necessary requirement for the approach to be used in the Linked Data cloud.

Moreover, we plan to explore further the connection between the quality metrics described in Deliverable D2.1 [26] and the preference expressions used in this work, and propose realistic preference expressions for use in specific contexts; towards this end, it makes sense also to study other repair strategies, not necessarily based on any notion of "minimality".

Along the same lines, completing the aforementioned user-friendly implementation of our framework (i.e., using an adequate Graphical User Interface) would allow evaluating our approach with real users. Towards this end, it is crucial to devise and use an adequate preference elicitation interface, to ensure that the preferences provided by the user will correspond to the actual preference intended, and will resemble as much as possible the output that would be provided by a manual repairing process.

REFERENCES

[1] F.N. Afrati and P.G. Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In *ICDT-09*, 2009.

[2] A. Bairoch, R. Apweiler, C.H. Wu, W.C. Barker, B. Boeckmann, S. Ferro, et al. The universal protein resource (uniprot). *Nucleic Acids Research*, 2005.

[3] L. Bertossi. *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers, 2011.

[4] L.E. Bertossi. Consistent query answering in databases. *SIGMOD Record*, 2006.

[5] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.

[6] L. Bravo and L. Bertossi. Semantically correct query answers in the presence of null values. In *EDBT*. 2006.

[7] A. Calì, G. Gottlob, and A. Pieris. Advanced processing for ontological queries. *Proceedings of VLDB Endowment*, 3:554–565, 2010.

[8] J. Chomicki and J. Marcinkowski. On the computational complexity of minimal-change integrity maintenance in relational databases. *Inconsistency Tolerance*, 2005.

[9] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB-07*, 2007.

[10] A. Deutsch. Fol modeling of integrity constraints (dependencies). In *Encyclopedia of Database Systems*. 2009.

[11] W. Fan. Dependencies revisited for improving data quality. In *PODS-08, invited talk*, 2008.

[12] P. Gärdenfors. *Belief Revision*, chapter Belief Revision: An Introduction, pages 1–28. Cambridge University Press, 1992.

[13] P. Georgiadis, I. Kapantaidakis, V. Christophides, E. Mamadou Nguer, and N. Spyratos. Efficient rewriting algorithms for preference queries. In *ICDE-08*, 2008.

[14] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *VLDB Journal*, 2007.

[15] T. Heath and C. Bizer. Linked data: Evolving the web into a global data space. In *Synthesis Lectures on the Semantic Web: Theory and Technology*, volume 1, pages 1–136. Morgan & Claypool, 2011.

[16] A. Hogan, A. Harth, A. Passant, S. Decker, and A. Polleres. Weaving the pedantic web. In *LDOW*, 2010.

[17] W. Kießling. Foundations of preferences in database systems. In *VLDB-02*, 2002.

[18] W. Kießling and G. Köstler. Preference sql - design, implementation, experiences. In *VLDB-02*, 2002.

[19] G. Konstantinidis, G. Flouris, G. Antoniou, and V. Christophides. A formal approach for rdf(s) ontology evolution. In *ECAI*, 2008.

[20] G. Lausen, M. Meier, and M. Schmidt. Sparqling constraints for rdf. In *EDBT-08*, 2008.

[21] Jens Lehmann and Lorenz Buhmann. Ore - a tool for repairing and enriching knowledge bases. In *ISWC*, 2010.

[22] M. Lenzerini. Data integration: A theoretical perspective. In *PODS-02*, 2002.

[23] D. Maier, A.O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *TODS*, 4, 1979.

[24] D.L. McGuinness, R. Fikes, J. Rice, and S. Wilder. An environment for merging and testing large ontologies. In *KR*, 2000.

[25] S. Melnik. *Generic Model Management: Concepts and Algorithms*. Springer, 2004.

[26] P. Mendes, C. Bizer, J.H. Young, Z. Miklos, J-P. Calbimonte, A. Moraru, and G. Flouris. Conceptual model and best practices for high-quality metadata publishing. Deliverable D2.1, PlanetData, 2012.

[27] D. Mindolin and J. Chomicki. Preference elicitation in prioritized skyline queries. *The VLDB Journal*, 20, 2011.

[28] B. Motik, I. Horrocks, and U. Sattler. Bridging the gap between owl and relational databases. In *WWW-07*, 2007.

[29] S. Munoz, J. Perez, and C. Gutierrez. Simple and efficient minimal rdfs. *Journal of Web Semantics*, 7:220–234, 2009.

[30] N.F. Noy and M.A. Musen. Prompt: Algorithm and tool for automated ontology merging and alignment. In *AAAI*, 2000.

[31] Yannis Roussakis, Giorgos Flouris, and Vassilis Christophides. Declarative repairing policies for curated kbs. In *Proceedings of the 10^{th} Hellenic Data Management Symposium (HDMS-11)*, 2011.

[32] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of rdf(s) query patterns. In *ISWC-05*, 2005.

[33] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of rdf(s) stores. In *ISWC-05*, 2005.

[34] Y. Theoharis, Y. Tzitzikas, D. Kotzinos, and V. Christophides. On graph features of semantic web schemas. *IEEE TKDE*, 20(5):692–702, 2008.

[35] Yannis Theoharis, George Georgakopoulos, and Vassilis Christophides. Powergen: A power-law based generator of rdfs schemas. *Information Systems*, 2011.

[36] M. Yakout, A.K. Elmagarmid, J. Neville, M. Ouzzani, and I.F. Ilyas. Guided data repair. *PVLDB*, 4(5):279–289, 2011.