RDF Updates with Constraints

Mirian Halfeld-Ferrari[1], Carmem S. Hara[2], and Flávio Uber[2,3] ✉

[1] Université d'Orléans, INSA CVL, LIFO EA 4022 FR-45067, Orléans, France
[2] Universidade Federal do Paraná, Curitiba-PR, Brazil
[3] Universidade Estadual de Maringá, Maringá-PR, Brazil
mirian@univ-orleans.fr, carmem@inf.ufpr.br, flavio.uber@gmail.com

Abstract. This paper deals with the problem of updating an RDF database, expected to satisfy user-defined constraints as well as RDF intrinsic semantic constraints. As updates may violate these constraints, side-effects are generated in order to preserve consistency. We investigate the use of nulls (blank nodes) as placeholders for unknown required data as a technique to provide this consistency and to reduce the number of side-effects. Experimental results validate our goals.

Keywords: RDF · RDFS · constraints · updates

1 Introduction

Due to the increasing number of distributed RDF datasets and their dynamic nature, the development of techniques for ensuring their consistency becomes a fundamental data quality issue. However, when analyzing the database and the web semantics worlds, a dichotomy on the notion of consistency can be observed. The web semantics world adopts the open world assumption (OWA) and ontological constraints are, in fact, inference rules. The database world usually adopts the closed world assumption (CWA) and constraints impose data restrictions. Let us consider the rule $r : Researcher(X) \rightarrow Professor(X)$ and a database storing the fact that Bob is a researcher ($D = \{Reseacher(Bob)\}$). When r is an inference rule, D is consistent because $Professor(Bob)$ is inferred from D and r. However, if r is a constraint, D is inconsistent because the fact $Professor(Bob)$ is not true in D (here, facts which are not stored in the database are considered false). Although inference rules and constraints can co-exist ([11,17]) their mechanisms are usually defined separately.

This paper adopts the database point of view and deals with the problem of updating an RDF database. Traditionally, whenever a database is updated, if constraint violations are detected, either the update is refused or compensation actions, which we call side-effects, must be executed in order to guarantee their satisfaction. Our work tackles the problem of "active rules" for RDF and computes the side-effects required by update operations. The originality of our approach is in the use of blank nodes as free nulls in the computation of side-effects. Although blank nodes have different capabilities [4], in this paper, we are only interested in their standard interpretation as existential variables (which can be replaced by free labeled nulls). To avoid confusion, we will refer to them

as null nodes (or just nulls), used as placeholders for unknown required data. Notice however that in our approach user's update requirements have no nulls: nulls can only be generated automatically during side-effect computation.

We work with a logical formalism using special predicates to describe RDF data. For instance, we write: (i) $CI(Bob, Professor)$ to express that Bob is an instance of the class $Professor$ and (ii) $PI(Bob, DB, teaches)$ to indicate an instance of property $teaches$, assuming that $Professor$ and $Courses$ are, respectively, the property's domain and range. In this context, the following example illustrates our challenges and gives an overview of our approach.

CI(Bob, Researcher) CI(Bob, Professor) PI(Bob, Jupiter, isMember) PI(Bob, DB, teaches) PI(Bob, CNPq,grantFrom) CI(Ann, Student) PI(Tom, Java, teaches)	r_1:PI(X_1,X_2,coordinates)→CI(X_1,Researcher) r_2:CI(X_1,Researcher)→PI(X_1,X_2,isMember) r_3:PI(X_1,X_2,coordinates)→PI(X_1,X_2,isMember) r_4:CI(X_1,Researcher)→ CI(X_1,Professor) r_5:CI(X_1,Professor)→ ¬CI(X_1,Student) r_6:CI(X_1,Professor)→ PI(X_1,X_2,teaches) r_7:PI(X_1,X_2,grantFrom)→CI(X_1,Researcher)
(a)	(b)

Fig. 1. Database Instance D (a) and Constraints $\mathcal{C}$ (b) for Example 1

Example 1. Let $\mathcal{C}$ (Figure 1) be a set of constraints defined on an academic application. Constraints state that only a researcher may coordinate a project (r_1) and that he must also be a member of this project (r_3). All researchers are required to be a member of at least one project (r_2). Researchers are professors (r_4), professors cannot be students (r_5) and are required to teach at least one course (r_6). Finally, people receiving research grants should be researchers (r_7). Constraints are defined as rules, where the left-hand side is called the *body* of the rule, and the right-hand side is its *head*. In this example we consider this set of constraints and analyze how a database instance is updated according to successive update requirements which may generate side-effects *w.r.t.* $\mathcal{C}$. Consider the database instance in Figure 1, which is consistent *w.r.t.* $\mathcal{C}$.

The insertion of the fact $CI(Ann, Professor)$ cannot be executed by simply adding this new fact in D because it provokes the violation of r_5 and r_6. The following side effects should be considered: (i) rule r_5 generates $\neg CI(Ann, Student)$, which corresponds to the deletion of $CI(Ann, Student)$, and (ii) rule r_6 produces $PI(Ann, N_1, teaches)$ where N_1 a new fresh null, indicating that Ann teaches a course, although it is not yet known which one. The new updated database is: $D_1 = (D \cup \{CI(Ann, Professor), PI(Ann, N_1, teaches)\}) \setminus \{CI(Ann, Student)\}$. Notice that D_1 contains a null value produced during the side effect computation.

Now, consider the deletion of $f = PI(Bob, Jupiter, isMember)$ from D_1. To avoid the violation of r_2 we cannot just eliminate f from D_1. The usual solution (also proposed by [8]) is to delete all facts generating f, in order to obtain $D_{trad} = D_1 \setminus \{PI(Bob, Jupiter, isMember), CI(Bob, Researcher), PI(Bob, CNPq, grantFrom)\}$. Such a solution seems too radical. Rule r_2 says that if someone is a researcher, there should *exist* a project having this person as a member. Deleting f only indicates that Bob is not a member of project $Jupiter$ any more (he is perhaps a member of an another project which we do not know yet). Therefore, in this situation, our proposal is to replace $PI(Bob, Jupiter, isMember)$ by $PI(Bob,$

N_3, $isMember$) where N_3 is a fresh null, a placeholder indicating that, for the moment, we do not know on which project Bob is working. The new updated database is $D_2 = (D_1 \setminus \{PI(Bob, Jupiter, isMember)\}) \cup PI(Bob, N_3, isMember)$. Notice that in this way we limit cascading deletions.

The latter reasoning is not appropriate for every situation. As a last example, consider the deletion of $CI(Bob, Professor)$. Rule r_4 states that *all* researchers must be professors. Clearly, if Bob is not a professor, he cannot be accepted as a researcher. Also if he is not a researcher, he cannot receive a research grant (r_7). In this case, replacing Bob by a null in $CI(Bob, Professor)$ is meaningless. To perform this update, cascading deletes are necessary. The new database is: $D_3 = D_2 \setminus \{CI(Bob, Professor), CI(Bob, Researcher), PI(Bob, CNPq, grantFrom)\}$. Notice that $PI(Bob, N_3, isMember)$ is still in D_3. Indeed, a rule such as r_2 does not impose members (of a project) to be researchers. □

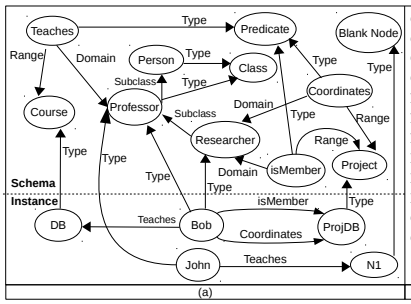
An important aspect of our update proposal is to use nulls when a deletion concerns instantiations of existential variables in the head of a constraint. To the best of our knowledge this is the first work that proposes an automated mechanism to introduce null nodes in RDF datasets to limit cascade updates.

We deal with two kinds of constraints separately: application constraints ($\mathcal{C}$), imposed by a user to personalize his context or to establish the particularity of his application and the RDF intrinsic semantic constraints ($\mathcal{A}$). Algorithms to compute side-effects from $\mathcal{A}$ and $\mathcal{C}$ are developed as distinct steps. Our goal is to compute $\mathcal{C}$'s side effects without consulting the database. The computation of $\mathcal{A}$'s side-effects, however, requires knowledge of the underlying data.

The rest of the paper is organized as follows. Section 2 defines our data model, constraints and update operations. Section 3 considers the computation of side-effects. Section 4 reports our experimental results. Related work and final remarks (Sections 5 and 6) conclude the paper.

2 RDF Constraints and Updates

Let $\mathbf{A}_C = \{a, b, \ldots, a_1, a_2, \ldots\}$, be a countably infinite set of constants, $\mathbf{A}_N = \{N_1, N_2, \ldots\}$, a countably infinite set of nulls and $var = \{X_1, X_2, \ldots, Y_1, \ldots\}$ be an infinite set of variables ranging over elements in $\mathbf{A}_C \cup \mathbf{A}_N$. We use $\mathbf{X}$ as an abbreviation for $X_1 \ldots X_k$ where $k \geq 0$. A *term* is a constant, a null or a variable. Extending the logical formalism of [8] to deal with nulls, we classify predicates into two sets: (i) SchPred = $\{Cl, Pr, CSub, Psub, Dom, Rng\}$, used to define the database schema, standing respectively for classes, properties, sub-classes, sub-properties, property domain and range, and (ii) InstPred = $\{CI, PI, Ind, BN\}$, used to define the database instance, standing respectively for class and property instances, individuals and blank nodes (or nulls). An *atom* has the form $P(u)$, where P is a predicate, and u is a list of terms. When all the terms of an atom are in $\mathbf{A}_C \cup \mathbf{A}_N$, we have an instantiated atom. When they are all in $\mathbf{A}_C$, we have a fact. Figure 2 illustrates our model. Figure 2(a) shows an RDF instance and schema as a graph, and Figure 2(b) a subset of its representation as positive atoms. Figure 2(c), reproduced from [8], shows the correspondence between triples in RDF/S and facts.



(a)

(b)
Csub(Professor, Person)
Cl(DB, Course)
Cl(John, Professor)
Cl(Bob, Professor)
Pl(Bob, DB, Teaches)
Pl(John, N1, Teaches)
Pr(Teaches)
Dom(Teaches, Professor)
Rng(Teaches, Course)
BN(N1)
Cl(Professor)
Cl(Person)

RDFS Triple	Ground Facts
(A,rdf:type,rdfs:Class)	Cl(A)
(A,rdf:type,rdf:Property)	Pr(A)
(A,rdf:type,rdfs:Resource)	Ind(A)
(A,rdf:type,B)	Cl(A,B)
(A,rdfs:subClassOf,rdfs:Resource)	Csub(A, rdfs:Resource)
(A,rdfs:subClassOf,B)	Csub(A,B)
(A,rdfs:subPropertyOf,B)	Psub(A,B)
(A,rdfs:domain,rdfs:Resource)	Dom(A,rdfs:Resource)
(A,rdfs:domain,B)	Dom(A,B)
(A,rdfs:range,rdfs:Resource)	Rng(A,rdfs:Resource)
(A,rdfs:range,rdfs:Literal)	Rng(A,rdfs:Literal)
(A,rdfs:range,B)	Rng(A,B)
(A,B,C)	PI(A,C,B)

(c)

Fig. 2. (a) RDF instance and schema. (b) Dataset defined as a set of facts. (c) Correspondence of RDF/S triples and facts [8]

Definition 1 (Database). An RDF database is a triple $\Delta = (D, D_{Sch}, \Sigma)$ where D is the database instance (a set of instantiated atoms with predicates in InstPred), D_{Sch} is the database schema (a set of facts with predicates in SchPred) and $\Sigma = (\mathcal{A}, \mathcal{C})$ is a set of constraints, where $\mathcal{A}$ is a set of RDF semantic constraints and $\mathcal{C}$ is a set of application constraints. □

2.1 Constraints

A constraint is a logical rule r whose left-hand side is denoted as *body(r)*, while the right-hand side is denoted as *head(r)*. Application constraints personalize the context on which an RDF database is treated while RDF constraints ensures the intrinsic RDF/S semantics.

Definition 2 (Application Constraints). Let c_1, c_2 be class labels, and p_1, p_2 be property labels in $\mathbf{A}_C$. Application rules in $\mathcal{C}$ have the forms presented in Table 1. Moreover, the following restrictions are imposed on $\mathcal{C}$: (1) for constraints r_1 of Type 2 there exists no constraint $r_2 \in \mathcal{C}$ such that $head(r_1)$ and $body(r_2)$ are unifiable; (2) for constraints r_1 of Type 3 there exists no constraint $r_2 \in \mathcal{C}$ such that $body(r_1)$ and $head(r_2)$ are unifiable. □

Our constraints are special cases of tuple generating dependencies (TGDs). We refer to [14] to recall that TGDs are database dependencies represented by the logical formula $\forall \mathbf{X}\ \phi(\mathbf{X}) \rightarrow \exists \mathbf{Y}\ \psi(\mathbf{X}, \mathbf{Y})$; where ϕ and ψ are conjunctions of atoms, all with variables among those in $\mathbf{X}$– every variable in $\mathbf{X}$ appears in $\phi(\mathbf{X})$ but not necessarily in $\psi(\mathbf{X}, \mathbf{Y})$. Although we restrict ourselves to the so-called linear LAV (local-as-a-view) TGDs [1] *i.e.*, to rule's body and head with a single atom, our constraints allow a negative atom in their heads. Restrictions imposed to our constraint rules aim at avoiding null propagation and at guaranteeing deterministic updates. In this context, side-effects generation does not deal with the well-known chase problems (considered, for instance, in [6]). For instance, consider r_2 of Example 1 and $r_2' : PI(X_1, X_2, isMember) \rightarrow PI(X_2, X_3, postulate4Grants)$. In such a context, the insertion of $CI(Bob, Researcher)$ would generate $PI(Bob, N_1, isMember)$ and $PI(N_1, N_2, postulate4Grants)$. Our constraints bypass controversial aspects related to the generation

of nulls from nulls in an update context by avoiding their propagation (*i.e.*, refusing a set $\mathcal{C}$ where both rules, such as r_2 and r_2', exist).

Our choice in separating application constraints from RDF/S semantic constraints allows us to impose the above restrictions to $\mathcal{C}$ without interfering with the well-known RDF/S constraints. With such restrictions we are able to build a simple and efficient algorithm to compute $\mathcal{C}$'s side-effects without dealing with some tricky aspects of the chase and without consulting the database instance.

Type 1:	$CI(X_1, c_1) \rightarrow CI(X_1, c_2)$ or
	$CI(X_1, c_1) \rightarrow \neg CI(X_1, c_2)$ or
	$PI(X_1, X_2, p_1) \rightarrow PI(X_1, X_2, p_2)$ or
	$PI(X_1, X_2, p_1) \rightarrow \neg PI(X_1, X_2, p_2)$
Type 2:	$CI(X_1, c_1) \rightarrow PI(X_1, X_2, p_1)$ or
	$CI(X_1, c_1) \rightarrow \neg PI(X_1, X_2, p_1)$ or
	$CI(X_2, c_1) \rightarrow PI(X_1, X_2, p_1)$ or
	$CI(X_2, c_1) \rightarrow \neg PI(X_1, X_2, p_1)$
Type 3:	$PI(X_1, X_2, p_1) \rightarrow CI(X_1, c_1)$ or
	$PI(X_1, X_2, p_1) \rightarrow \neg CI(X_1, c_1)$
	$PI(X_1, X_2, p_1) \rightarrow CI(X_2, c_1)$ or
	$PI(X_1, X_2, p_1) \rightarrow \neg CI(X_2, c_1)$

Table 1. Types of application constraints.

2.2 Updates

An *update set* is a set of operations where each operation is a positive or negative instantiated atom corresponding, respectively, to insertions and deletions which are performed in just *one* transaction. Only instance-level updates (*i.e.*, involving predicates in InstPred) are treated by our approach.

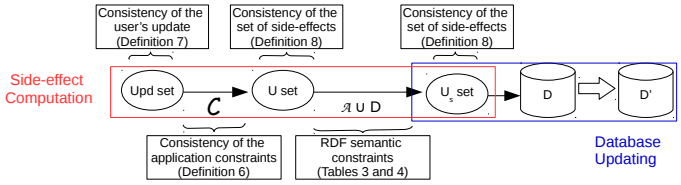


Fig. 3. Computing update side effects: consistency requirements at each step.

We distinguish three update steps, summarized in Figure 3. Each step refers to a distinct update set respecting specific consistency requirements. To update a database instance D, a user gives as input his update requests in the set *upd* where only *facts* are allowed. Facts in *upd* may trigger constraints in $\mathcal{C}$ allowing the computation of the set U. Finally, by using the RDF constraints in $\mathcal{A}$ and information in D, the set U_S is generated. The updated database D' is obtained by applying updates in U_S on D.

Before considering the computation of U and U_S, we establish our update semantics by showing how the updates in U_S are applied to D.

Given an update set, a positive atom denotes an insertion while a negative one denotes a deletion. When dealing with a non-null database instance, the semantics of these basic operations is straightforward. Given update sets $\{CI(a, c)\}$ and $\{\neg CI(a, c)\}$ on D, the resulting databases are, respectively, $(D \cup \{CI(a, c)\})$ and $(D \setminus \{CI(a)\})$. The possibility of having nulls in the database, introduced during side-effect computation, imposes a new update semantics, defined in Table 2. Operations not listed in the table have no direct effect on the database. For

Table 2. Semantics of update operations on a database instance D.

$[\![CI(a,c)]\!]_D =$	$\{CI(a,c)\}$	$[\![\neg CI(a,c)]\!]_D =$	$\{\neg CI(a,c)\}$
$[\![\neg Ind(a)]\!]_D =$	$\{\neg Ind(a)\} \cup$ $\{\neg PI(X,a,p) \mid PI(X,a,p) \in D\} \cup$ $\{\neg PI(a,X,p) \mid PI(a,X,p) \in D\}$	$[\![Ind(a)]\!]_D =$	$\{Ind(a)\}$
$[\![BN(N_1)]\!]_D =$	$\{BN(N_1)\}$	$[\![\neg BN(N_1)]\!]_D =$	$\{\neg BN(N_1)\}$
$[\![PI(a,b,p)]\!]_D =$	$\{PI(a,b,p)\} \cup$ $\{\neg PI(N_1,b,p), \neg BN(N_1) \mid PI(N_1,b,p) \in D\} \cup$ $\{\neg PI(a,N_1,p), \neg BN(N_1) \mid PI(a,N_1,p) \in D\}$	$[\![\neg PI(a,b,p)]\!]_D =$	$\{\neg PI(a,b,p)\}$
$[\![PI(N_1,b,p)]\!]_D =$	if there exists X such that $PI(X,b,p) \in D$ then $\{\}$ else $\{PI(N_1,b,p), BN(N_1)\}$	$[\![\neg PI(N_1,b,p)]\!]_D =$	$\{\neg PI(X,b,p) \mid$ $PI(X,b,p) \in D\}$
$[\![PI(a,N_1,p)]\!]_D =$	if there exists X such that $PI(a,X,p) \in D$ then $\{\}$ else $\{PI(a,N_1,p), BN(N_1)\}$	$[\![\neg PI(a,N_1,p)]\!]_D =$	$\{\neg PI(a,X,p) \mid$ $PI(a,X,p) \in D\}$

each operation, we show what should be added (positive atoms) and removed (negative atoms) from a database D. Intuitively, nulls are inserted in D only if there exists no other resource in D that plays the same role. Insertion and removal of *property instances* are particularly impacted. An insertion of a property p without nulls ($[\![PI(a,b,p)]\!]_D$) *replaces* one that involves nulls that may already exist in the database. This is in accordance to our semantics for nulls as placeholders of unknown required data. Thus, an insertion of a property involving a resource a and a null N_1 ($[\![PI(a,N_1,p)]\!]_D$ or $[\![PI(N_1,a,p)]\!]_D$) only affects the database if a is not involved with any other resource or null by property p. Since inserted nulls are interpreted as existential variables, nulls in removal operations (negative atoms) are interpreted as universally quantified variables. Thus, $[\![\neg PI(a,N_1,p)]\!]_D$ results in deleting from the database all properties p of resource a. We denote by $D \biguplus U_S$ the result of applying an update set U_S on database D. Based on the semantics of these operations, we can define the notion of subsumption between update operations.

Definition 3 (Update Subsumption). Operation op_1 is subsumed by op_2, denoted as $op_1 \preceq op_2$ if for any database D, $([\![op_1]\!]_D \cup [\![op_2]\!]_D) = [\![op_2]\!]_D$. That is, the effect on the database of executing op_1 and op_2 is the same as executing only op_2. Given a set of update operations upd, we denote by $clean(upd)$ a subset of upd with no subsumed operations. □

As examples, $PI(Bob, N_1, coordinates) \preceq PI(Bob,\ Proj\ DB,\ coordinates)$ and $\neg PI(Bob, ProjDB, coordinates) \preceq \neg PI(Bob, N_1, coordinates)$. Thenceforth, we only consider update sets containing non-subsumed operations.

3 Computing Side Effects

We develop algorithms for computing side-effects which only affect the database instance. They have the following guidelines: (*i*) from an input set of updates *upd* we use constraints to generate a new list of updates that should be performed to guarantee constraint validation; (*ii*) null nodes are generated during the inference process that computes side effects *i.e.*, the inference can be stopped by the use of nulls; (*iii*) side effects from application constraints are computed without consulting the database; and (*iv*) side effects from RDF semantic constraints are computed in a subsequent step and may require inspection of the database.

3.1 Side Effects based on Application Constraints

Firstly, let us introduce some notations. Let I be an update set, then: (1) $I = I^+ \cup I^-$ where I^+ is the set of positive atoms and I^- is the set of negative atoms and (2) $I = \bullet(I) \cup \circ(I)$ where $\bullet(I)$ is a set of ground atoms (without nulls) and $\circ(I)$ is a set of atoms having nulls. Clearly we can write $I^+ = \bullet(I^+) \cup \circ(I^+)$ and $I^- = \bullet(I^-) \cup \circ(I^-)$. We recall that a *homomorphism* from the set of atoms A_1 to the set of atoms A_2, both over the same predicate P, is a a function (or a *substitution*) h from the terms of A_1 to the terms of A_2 such that: (i) if $t \in \mathbf{A}_C$, then $h(t) = t$, and (ii) if $P(t_1, ..., t_n) \in A_1$, then $P(h(t_1), ..., h(t_n)) \in A_2$. If h is a homomorphism, $P(h(t_1), ..., h(t_n))$ is simply denoted by $h(P(t_1, ..., t_n))$. The notion of homomorphism naturally extends to conjunctions of atoms.

Now, given I and $\mathcal{C}$ we introduce three operators used for computing side-effects. The first operator (T) computes the side-effects traversing the rule forward, from the body to the head while the second (ϑ) moves backwards, from the head to the body.

Definition 4 (Operators T and ϑ). Let $T_{\mathcal{C}}$ and $\vartheta_{\mathcal{C}}$ be operators over a set of application constraints $\mathcal{C}$ and an update set I. When the set $\mathcal{C}$ is understood, we write T (respectively, ϑ) instead of $T_{\mathcal{C}}$ (respectively, $\vartheta_{\mathcal{C}}$) .

$T(I) = I \cup \{l \mid \exists c \in \mathcal{C} \wedge$ there is a homomorphism h such that $h(body(c)) \in I \wedge l = \widehat{h}(head(c))$ where $\widehat{h} \supseteq h$ is an extension of h such that when c is of Type 2 and Y is the set of terms in $head(c)$ and not in $body(c)$, then for every $y_i \in Y$, if $y_i \in var$, then $\widehat{h}(y_i) = N_i$ where $N_i \in \mathbf{A}_N$ is a fresh null. $\}$

$\vartheta(I) = I \cup \{l \mid \exists c \in \mathcal{C} \ \wedge$ there is a homomorphism h such that $h(head(c)) \in I \wedge l = \widehat{h}(body(c)$ where $\widehat{h} \supseteq h$ is an extension of h such that when c is of Type 3 and Y is the set of terms in $body(c)$ and not in $head(c)$, then for every $y_i \in Y$, if $y_i \in var$, then $\widehat{h}(y_i) = N_i$ where $N_i \in \mathbf{A}_N$ is a fresh null node. $\}$ □

Operators T and ϑ are monotonic and have a *least fixed point*. We denote by $T^*(I)$ and by $\vartheta^*(I)$, the least fixed point of T and ϑ, respectively, with respect to a set of ground atoms I. Since constraints in $\mathcal{C}$ have only positive bodies, we have $T^*(I^-) = I^-$ and $T^*(I^-)^- = \vartheta^*(I^-)^- = I^-$.

Example 2. Consider $\mathcal{C}$ of Example 1. Let $I = \{ PI(John, projDB, coordinates)\}$. Applying Definition 4, we obtain $T^*(I) = \{ PI(John, projDB, coordinates)$, $CI(John, Researcher)$, $PI(John, N_1, isMember)$, $PI(John, projDB, isMember)$, $CI\ (John, Professor)$, $PI(John, N_2, teaches)$, $\neg CI(John, Student)\}$. Now let $I = \{ PI(John, projDB, isMember)\}$. Applying Definition 4, we obtain $\vartheta^*(I) = \{ PI(John, projDB, isMember)$, $CI(John, Researcher)$, $PI(John, projDB, coordinates)$, $PI(John, N_1, coordinates)$, $PI(John, N_2, grantFor)\}$. □

We denote by $\vartheta_{|ty_{1,3}}$ the application of the operator ϑ restricted to constraints of Types 1 and 3 and we introduce a third operator, η, that only applies to rules of Type 2. As seen in Example 1 (instance D_2), a special treatment is proposed when dealing with instantiations of existential variables on a constraint's head. As defined below, $\eta(PI(Bob, Jupiter, isMember))$ contains $PI(Bob, N_1, isMember)$.

Definition 5 (Operator η). Operator η (or $\eta_{\mathcal{C}}$) is applied only on Type 2 rules. For each rule c of Type 2, let us denote by (i) X the set of terms appearing in both $body(c)$ and $head(c)$ and (ii) Y the set of terms appearing in $head(c)$ but not in $body(c)$. Let I be a set of facts.
$\eta(I) = \{l \mid \exists c \in \mathcal{C}$ such that c is of Type 2 $\wedge$ there is an homomorphism h such that $h(head(c)) \in I \wedge$ there is an homomorphism h_2 such that $l = h_2(head(c))$ with $h_2(X) = h(X) \wedge$ for every $y_i \in Y$ if $y_i \in var$ then $h_2(y_i) = N_i$, where N_i is a fresh null node. $\}$ □

Consistency. We now turn to the problem of defining different notions of consistency employed on each step of our method (as illustrated in Figure 3). Firstly, since rules in $\mathcal{C}$ may have positive or negative literals in their heads, it is important to determine when a set $\mathcal{C}$ is a consistent set of update rules (algorithms of this kind can be found in [11]).

Definition 6 (Consistent set of application constraints). *A set $\mathcal{C}$ of application constraints is* consistent *if for every fact f, $(T^*(f))^+ \cap \neg.(T^*(f))^- = \emptyset$.* □

Now let us establish the consistency definition for the user's update requests.

Definition 7 (Consistency of the user's update requests). *Given an update set I, we say that I is a consistent set of user's update requests if there are no two atoms $l_1 \in I$ and $l_2 \in I$ for which there exists a homomorphism from $\boldsymbol{A}_N \rightarrow (\boldsymbol{A}_N \cup \boldsymbol{A}_C)$ such that $h(l_1) = \neg h(l_2)$.* □

The consistency introduced in Definition 7 is imposed to the set *upd* (Figure 3) and also to its immediate consequences obtained by traversing rules forward (operator T). However, the notion of a *consistent set of side-effects*, such as U, which considers traversing backwards (operator ϑ and η) is more relaxed than the notion stated in Definition 7. Indeed, the consistency of sets U (obtained by Algorithm 1, which computes the side-effects imposed by application constraints) and U_S (obtained by Algorithm 2, which computes the side-effects imposed by RDF semantic constraints) follows the definition below.

Definition 8 (Consistency of the set of side-effects). *Given an update set I, we say that I is a consistent set of side-effects if for each positive atom $l_0 \in I$ there is no negative atom $\neg.l_1 \in I$ such that $l_0 = h(l_1)$ where h is a homomorphism from $\boldsymbol{A}_N \rightarrow (\boldsymbol{A}_N \cup \boldsymbol{A}_C)$.* □

As an example, let us consider the following sets: $I_1 = \{\neg.PI(Bob, DB, teaches)$, $PI(Bob, N_1, teaches)\}$; $I_2 = \{PI(Bob, DB,\ teaches)$, $\neg.PI(Bob, N_1, teaches)\}$; $I_3 = \{PI(Bob, N_1,\ teaches)$, $\neg.PI(Bob, N_1, teaches)\}$ and $I_4 = \{PI(Bob,\ DB, teaches)$, $\neg.PI(Bob, DB, teaches)\}$. According to Definition 7 all of them are examples of inconsistent update sets. However, according to Definition 8 set I_1 is consistent, while sets I_2, I_3 and I_4 are inconsistent. Indeed, I_2, I_3, and I_4 are direct consequences from the semantics of the operations. The discussion of I_1 consistency is more subtle. First, consider Definition 7, which must be satisfied after traversing rules forward. Observe that if $PI(Bob,\ N_1,\ teaches)$ is in I_1 it must be the case that there exists a rule imposing the insertion, such as rule r_6 in Example 1. Thus, the update set must contain an operation that triggered the insertion of $PI(Bob,\ N_1,\ teaches)$, such as $CI(Bob,\ Professor)$. On the

other hand, $\neg.PI(Bob, DB, teaches)$ is also in I_1. In the traditional (cascading) semantics, this update would required the deletion of $CI(Bob, Professor)$, according to r_6. But this is inconsistent with the fact that triggered the insertion of $PI(Bob, N_1, teaches)$. Intuitively, it is not clear what the user's intentions are. Since Definition 7 concerns the consistency of the *user* update requests, we consider this demand inconsistent. Inconsistent sets of user's update requests are rejected. The process of computing side-effects continues for consistent update sets, traversing rules backwards (operators ϑ and η). Definition 8 applies to the result of this process. Consider again rule r_6 of Example 1 and a single user update request $\neg.PI(Bob, DB, teaches)$. The application of operator η generates as side-effect $PI(Bob, N_1, teaches)$ as a *technique* to stop cascading updates. Thus, according to Definition 8, set I_1 is consistent because it started with a consistent set of user's update requests (and thus without the aforementioned ambiguity) and the existence of both $\neg.PI(Bob, DB, teaches)$ and $PI(Bob, N_1, teaches)$ must be the result of applying the η operator.

Algorithm. We now introduce Algorithm 1 to compute side effects imposed by *application* constraints. In this algorithm, Line 1 applies operator T over an update set *upd* and obtains a set of positive and negative atoms which are the side effects imposed by $\mathcal{C}$ moving forward. Once update requests are cleaned, consistency is verified (Line 2). Notice that at this step we still require consistency *w.r.t.* Definition 7. However, for the result set U consistency requirements is relaxed (Definition 8). We handle positive and negative atoms separately. When rules of Type 2 (Definition 2) are used by T, fresh null nodes appear and thus the result in $T^*(upd)$ is a set of atoms, not necessarily grounded.

Algorithm 1: Side effects due to $\mathcal{C}$

Input: A set of (positive or negative) facts *upd* and a consistent set of constraints $\mathcal{C}$
Output: The set U of side effects of *upd w.r.t.* $\mathcal{C}$.

1: Compute $T^*(upd) := clean(T^*(upd))$, where $T^*(upd) = T^*(upd)^+ \cup T^*(upd)^-$
2: **if** $T^*(upd)$ is consistent *w.r.t.* Definition 7 **then**
3: $Inv := \emptyset$; $U := \emptyset$;
4: **for all** $f \in \circ(T^*(upd)^-)$ or $f \in T^*(upd)^+$ **do**
5: $Inv := Inv \cup \vartheta^*(\neg f)$
6: **for all** $f \in \bullet(T^*(upd)^-)$ **do**
7: $Inv := Inv \cup \vartheta^*_{\restriction_{ty1,3}}(\neg f)$; $\circ(U^+) := \eta(\neg f)$;
8: $U^+ = U^+ \cup T^*(upd)^+$; $U^- = T^*(upd)^- \cup \neg.Inv^+$;
9: **return** U
10: **else**
11: Error Exception: update requests are not consistent

Example 3. Analyzing $T^*(upd)$ of Example 2 we notice that: (i) cleaning $T^*(upd)$ implies annulling the update request $PI(John, N_1, isMember)$, since it is subsumed by $PI(John, projDB, isMember)$ and (ii) if $CI(John, Student)$ is in the database, it must be removed. Now, each positive atom obtained in Line 1 of Algorithm 1 represents a required insertion and thus, cannot be false in the database. Operator ϑ is used to find atoms which generate $\neg f$ (the inverse of f).

Given $f_1 : \neg PI(John, projDB, isMember)$, the result of $\vartheta^*(\neg f_1)$ is depicted in Example 2 and contains $f_2 : PI(John, projDB, coordinates)$, obtained when moving backwards on r_3. Observe that f_1 is in $T^*(upd)^-$ and f_2 is inserted in Inv (Line 7). Subsequently, in Line 8, $\neg f_2$ is inserted in the resulting update set U^-. The same treatment is given for positive atoms in $T^*(upd)^+$, and negative atoms with null nodes ($\circ(T^*(upd)^-)$) (Lines 4 and 5). Notice that $f_3 : CI(John, Researcher)$ is also in $\vartheta(\neg f_1)$ by rule r_2. However, r_2 is of Type 2. In this case, the algorithm does not include f_3 in Inv (since the ϑ operator is restricted to rules of Types 1 and 3), but applies the η operator (Line 7) on f_1 and includes $f_4 : PI(John, N_1, isMember)$ in Inv. As consistency requirements at this step are different from those in Definition 7, both f_1 and f_4 are accepted in the resulting set U and we stop the backward chaining. □

Proposition 1. *Let upd be a consistent set of user's update requests (Definition 7) and $\mathcal{C}$ a consistent set of constraints (Definition 6). The update set U obtained by Algorithm 1 satisfies the following properties:* (1) *$upd \subseteq U$;* (2) *U is consistent w.r.t. Definition 8;* (3) *U satisfies $\mathcal{C}$ and* (4) *for each atom $l \in (U \backslash upd)$ the set $U \setminus \{l\}$ does not satisfy $\mathcal{C}$.* □

3.2 Side Effects based on RDF Semantic Constraints

An RDF database should respect the intrinsic RDF semantic constraints in $\mathcal{A}$. Thus, following the same reasoning used for constraints in $\mathcal{C}$, our approach proposes to generate additional updates in order to maintain consistency *w.r.t.* $\mathcal{A}$. Constraints in $\mathcal{A}$ are those presented in Tables 3 and 4. Table 3 borrows from [8] a subset of the RDF semantic constraints, and Table 4 presents the rules that have been modified and added in order to consider the existence of nulls.

In Table 4, we consider that null nodes are distinct from predicates, entities and classes (rules b_1-b_3) and that they can be the subject or object of a property (rules b_4-b_5). They are also used to stop the propagation of properties required by particular classes. This is done by allowing properties to connect to null nodes even when the type of the subject and the object are defined (rules b_6 and b_7). The following example illustrates this point.

Example 4. In Example 1 we assume the existence of a class $Person$ which is the domain of property $isMember$. The insertion of $PI(N_1, projDB, isMember)$ satisfies the RDF semantic constraints without any additional side-effects. This is because rule b_6 allows a null node to be the subject of $isMember$ even when the schema defines that $Dom(isMember, Person)$. However, in [8], b_6 is defined as $PI(x, y, z) \wedge Dom(z, w) \rightarrow CI(x, w)$. In order to satisfy this constraint, $CI(N_1, Person)$ should be inserted as side-effect. As a consequence, the null node becomes "typed" and all properties for $Person$ would be required for N_1, possibly generating several additional null nodes. □

The above example shows that, similarly to the restrictions introduced to constraints in $\mathcal{C}$, the goal of the adapted rules b_1-b_7 is to avoid null propagation.

Table 3. Subset of rules from [8]

m_1: Cl(x) ∧ Pr(y) → (x ≠ y)
m_2: Cl(x) → Csub(x, rdfs:Resource)
m_3: Ind(x) → CI(x, rdfs:Resource)
m_4: Csub(x,y) ∧ Csub(y,z) → Csub(x,z)
m_5: Pr(x) → Dom(x,y) ∧ Rng(x,z)
m_6: CI(x,y) → Ind(x)
m_7: CI(x,y) → Cl(y) ∨ (y=rdfs:Resource)
m_8: PI(x,y,z) → Pr(z)
m_9: CI(x,y) ∧ Csub(y,z) → CI(x,z)

Table 4. Rules that involve blank nodes

b_1: Pr(x) ∧ BN(y) → (x ≠ y)
b_2: Ind(x) ∧ BN(y) → (x ≠ y)
b_3: Cl(x) ∧ BN(y) → (x ≠ y)
b_4: PI(x,y,z) → Ind(x) ∨ BN(x)
b_5: PI(x,y,z) → Ind(y) ∨ Lit(y) ∨ BN(y)
b_6: PI (x,y,z) ∧ Dom(z,w) → CI(x,w) ∨ BN(x)
b_7: PI(x,y,z) ∧ Rng(z,w) → CI(y,w) ∨ (Lit(y) ∧ (w=rdfs:Literal)) ∨ BN(y)

Algorithm 2: Side effects due to $\mathcal{A}$

Input: A set of updates U, a database instance D, application constraints $\mathcal{C}$
Output: The set U_S of side effects of U *w.r.t.* $(\mathcal{A} \cup \mathcal{C})$.

1: $U_S := \emptyset$;
2: **repeat**
3: **if** $ChangeSchema(U, D)$ or U is not consistent *w.r.t.* Definition 8 **then**
4: Error Exception;
5: $U_0 := U$;
6: $U_S := \{[\![op]\!]_{D \cup U_S} \mid op \in U \wedge op$ has a predicate in InstPred$\}$;
7: $U_{BN} := \{op \mid op \in U_S \wedge op = \neg PI(u) \wedge$ u contains null $N_1 \wedge \neg BN(N_1) \in U_S\}$;
8: $U_S := U_S - \{[\![op]\!]_{D \cup U_S} \mid op \in U_{BN}\}$;
9: $U := U \cup \{l \mid l = ResultTrigRule(r, op, D \uplus U_S)$ such that $r \in \mathcal{A} \wedge op \in U_0$ is an insertion or a deletion without nulls $\}$;
10: **for all** deletions of the form $\neg CI(a, c)$ in U_0 **do**
11: $U := U \cup \{l \mid l$ is of the form $PI(N_1, b, p)$ (or $PI(b, N1, p)$), s.t. there exists $PI(a, b, p)$ ($PI(b, a, p)$, respect.) in $(D \uplus U_S)$ whose deletion violates $r \in \mathcal{C}\}$;
12: **until** $U = U_0$;
13: **return** U_S;

Algorithm 2 computes U_S, an extension of U which includes: (1) the interpretation of each update on D, according to Table 2 and (2) side-effects obtained by applying $\mathcal{A}$ on U, on the basis of D_{Sch}. The complete algorithm is in [18].

Line 3 rejects update sets that contain schema changes or that are inconsistent *w.r.t.* Definition 8. As an example, the insertion of $CI(CNPq, RInst)$, where $RInst$ is a non existent class, triggers m_7 and produces the insertion of this class in the schema ($Cl(RInst)$)). Since we do not support schema changes, the entire update set is rejected by Algorithm 2. Consider now a consistent update set $U = \{CI(Bob, Student), \neg PI(Bob, N_1, hasSalary), \neg CI(Bob, Person)\}$. Assuming that $Student$ is a subclass of $Person$, rule m_9 imposes the insertion of $CI(Bob, Person)$ in U, which becomes inconsistent, and thus rejected by the algorithm.

As in Algorithm 1, insertions (positive atoms) activate rules forward and their instantiated heads are added to U while deletions (negative atoms) trigger them backwards, inserting the inverse of their instantiated bodies to U. On line 9, the set U is completed in this way by function *ResultTrigRule*. Consider $\mathcal{C}$ of Example 1 and database instance $\{CI(Bob, Professor), CI(DB, Course), PI(Bob, DB, teaches)\}$. The update $upd = \{\neg CI(DB, course)\}$ does not produce any application-level side-effects. However, b_7 imposes the deletion of $PI(Bob, DB, teaches)$. Such a deletion violates r_6 (line 11). Thus, our algorithm adds $PI(Bob, N_1,$

teaches) to U. Note that this operation will *effectively* insert a null if DB was the only course taught by Bob, according the the semantics of $[\![PI(Bob, N_1, teaches)]\!]$ as defined in Table 2.

Proposition 2. *Let $\Delta = (D, D_{Sch}, \Sigma)$ be a consistent database w.r.t. $\Sigma = (\mathcal{C}, \mathcal{A})$. Given an update set upd, let U_S be the set of side-effects computed according to Algorithms 1 and 2. Let U_S^{Sch} be the subset of U_S with facts in* Sch-Pred. *If U_S is consistent (Definition 8) and $U_S^{Sch} \subseteq D_{Sch}$ then the result of $D \biguplus U_S$ is a new database instance D' which satisfies the following properties: (i) $upd^+ \subseteq D'$ and $upd^- \not\subseteq D'$ and (ii) D' is satisfies $\mathcal{C}$ and $\mathcal{A}$.* □

Complexity. In Algorithm 1 the computation of the fix-point of T corresponds to the immediate consequence operator used in Datalog, since only positive rules may iterate. It is known that the number of iterations is bounded by the number of rules plus one ($O|C|$). As in our approach instantiation of application constraints is bounded by $O(|C| \times |upd|)$, the size of U is $O(|C|^2 \times |upd|)$. Cleaning U is $O(|C|^2 \times |upd|)^2$. Algorithm 2 nowadays works on a simple file, a non-optimized version where the most expensive task is limited by $O(|U|^2 \times |D|^2)$ (where $|D|$ is the database instance size). Thus, its complexity is $O(|C|^4 \times |upd|^2 \times |D|^2)$.

4 Experimental Study

We have implemented the BNS system, based on Algorithms 1 and 2, using the Standard ML of New Jersey compiler. In this section, BNS is compared with the FKAC system. FKAC is an implementation of the approach proposed in [8]. Among the related work, it is the most similar to our proposal. To provide a fair comparison, the FKAC approach has been modified in the following aspects: (*i*) We do not allow it to compute schema changes. As the original system selects the smallest side-effect set among all possible ones, this modification reduces considerably the search space and thus the execution time for side-effects computation. (*ii*) We do not allow it to compute deletions as insertion side-effects. Deleting atoms with a null value is much more expensive than inserting them, since the removal requires a database traversal to determine all possible null instantiations while the insertion needs just one instantiation. Thus, these implementation choices are advantageous for the FKAC system.

An important difference between FKAC and BNS concerns the capability of storing null values. As the FKAC system *does not accept nulls*, when an insertion imposes the existence of an unknown required data, an arbitrary instantiation is performed. This instantiation is *not* a user's choice, and thus cannot be considered as semantically meaningful. For instance, consider the insertion of $CI(Bob, Researcher)$ in the context of Example 1. Rule $r2$ requires Bob to be a member of some project. If the range of $isMember$ is the class $Project$, one of the possible side-effects proposed by the FKAC system is to *choose* an arbitrary instance of $Project$, say $Jupiter$, and insert $PI(Bob, Jupiter, isMember)$.

Worst still, when in a later time, the user associates Bob to a *real* project, the previous arbitrary fact is not removed. Contrary to that, BNS stores null values, provided that it is not subsumed by an existing fact in the database. An

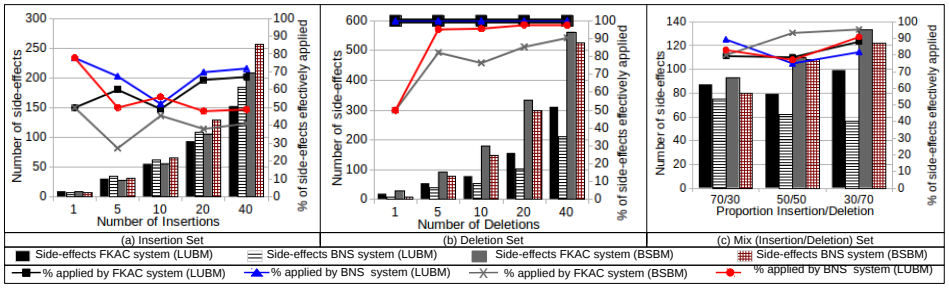


Fig. 4. Results for BSBM and LUBM Benchmarks

atom with a null value can then be replaced by facts introduced by the user in later time. Since BNS never introduces arbitrary facts in the database we can say that its strategy is semantically more meaningful.

In our experiment, we first generate U using Algorithm 1 with $\mathcal{C}$. Then, before forwarding U to the FKAC system, we replace or expand atoms involving nulls.

We use Berlin [5](BSBM) and LUBM [10], two benchmarks frequently adopted in RDF experiments. A dataset with 500 products were generated using Berlin (220,000 triples, 900,000 facts in our model). Based on the benchmark specification, we have identified 19 application constraints, which were also translated to our model ([18]). For LUBM, a dataset with 198,668 triples (1,784,296 facts) were generated. We have identified 10 application constraints in the benchmark specification. As the motivation for introducing nulls in side-effect operations is to limit cascade updates, we compare BNS and FKAC systems *w.r.t.* the size of the update set. That is, given an input update set *upd*, randomly generated, we compare the final size of U_S. We consider sets *upd* of increasing sizes, and with three different compositions: only insertion operations, only deletion operations, and mixed sets with insertion/deletion percentage of 70/30, 50/50, and 30/70.

Figure 4 presents the results for the LUBM and Berlin benchmarks. The bars show the final update size (which in Algorithm 2 are denoted as U_S), while the lines present the percentage of operations in this set that *effectively* modify the database, by inserting or removing facts (following Definition 3). Figure 4(a) show that when considering only insertion operations, the number of side-effects in the BNS system is larger than in the FKAC system. This is caused by the way we handle insertions with nulls in both systems. For instance, in FKAC, the insertion $PI(a, N_1, p)$ is a single operation which replaces N_1 by an existing instance in the database. However, in BNS, besides the insertion with N_1, we also create as side-effects: $CI(N_1, rdfs : resource)$ and $BN(N_1)$. In the graph, for 40 insertions in LUBM, 16 operations with nulls have been created as side-effects, resulting in a set U_S with 32 more updates for BNS than for FKAC. This number corresponds to the two additional atoms for each null created by BNS.

With respect to the percentage of side-effects that affect the database, the lines for both systems follow the same pattern. Indeed, the number of SchPred predicates (which have no effect on the database) in the resulting U_S is similar for both systems. Additionally, as Berlin datasets are more densely populate the percentage of effective operations in the BNS system is smaller. To see why, consider again the operation $PI(a, N_1, p)$. According to Table 2, this operation

has no effect if a is already linked to some other instance (or null node) through p. Thus, more densely populated databases tend to create fewer nulls.

BNS system avoids deletion propagation, as shown in Figure 4(b). As the FKAC system generates only facts to be deleted by finding the nulls' instantiations, its percentage of *effectively* executed updates is always close to 100%. For BNS, on the other hand, a deletion may require the insertion of a null value and this null value may require a not allowed schema change. Thus updates may be rejected more often. Consider for example the LUBM benchmark with 20 deletions. The number of updates with side-effects is 155 for FKAC system and 103 for the BNS system. While FKAC performs 100% of the operations, BNS performs only 83%. The results with sets of mixed operations show that even when 70% of the operations are insertions, the reduction on the number of side-effects resulting from the deletions, by limiting cascading updates, overcomes the overhead of creating null nodes. Thus, our approach is effective on reducing the size of the update set, while generating semantically meaningful operations as side-effects. Making an analogy with the semantics of deletion operations in the relational model, the FKAC approach is similar to the 'on delete cascade' while we adopt the 'on delete set null' semantics.

5 Related Work

The co-existence of constraints, as in a CWA, with inferences, as in a OWA, has recently inspired some works on RDF data management. In [7] and in Stardog [17], a new knowledge graph platform, coincide in their capability of considering both types of rules, reliving the proposal in [11]. Technologies such as ShEx [16] and SHACL [13], deal with the validation of the shape of an RDF graph. Although their focus in on schema and ours is on integrity constraints, a study on their interaction with our work deserves further investigation. However, none of them deals with consistency maintenance due to updates. Mechanisms to control frequent updates on RDF are desirable [15], but the update and consistency maintenance approaches in [8,15] do not consider nulls. As stated in [4], the standard semantics for blank nodes comes from first order logic and interprets them as existential variables. However, blank nodes are now treated in different ways, implying different semantics (such as [9]). This paper focus on its standard semantics and to avoid confusion, we denote them as *nulls*. Our update approach falls into the category Sem_2^{mat} of [2]. Adapting the exception viewpoint in [11] to our current work is a future perspective which approaches our work to [3]. Even if we consider updates as changes in the world; and not as a revision in our world's knowledge ([12]), it is possible to relate results of our method to the core principles of belief revision.

6 Conclusion

We present algorithms to determine the set of side-effects, consisting of additional updates required to keep the database consistent in the presence of application and RDF semantic constraints. Our approach differs from previous works because side-effects may introduce nulls in order to reduce cascade updates. Our experimental study shows that, although insertions tend to generate a larger

set of updates (justified by null nodes definitions), deletions tend to generate a smaller set of updates, since null nodes interrupt cascade deletions. Future work perspectives include a more expressive class of application constraints, while keeping the ability to deterministically compute the set of side-effects and experiments with a larger number of updates and datasets.

Acknowledgements: This work is partially funded by APR-IA Girafon and PEPS-INS2I Multipoint.

References

1. Afrati, F.N., Kolaitis, P.G.: Repair checking in inconsistent databases: algorithms and complexity. In: Proc of the Int Conf on Database Theory. pp. 31–41 (2009)
2. Ahmeti, A., Calvanese, D., Polleres, A., Savenkov, V.: Dealing with inconsistencies due to class disjointness in SPARQL update. In: Proc of the 28th Int Work on Description Logics (2015)
3. Ahmeti, A., Calvanese, D., Polleres, A., Savenkov, V.: Handling inconsistencies due to class disjointness in SPARQL updates. In: Proc of the 13th Extended Semantic Web Conference. pp. 387–404 (2016)
4. Arenas, M., Barceló, P., Libkin, L., Murlak, F.: Foundations of Data Exchange. Cambridge University Press (2014)
5. Bizer, C., Schultz, A.: The berlin sparql benchmark. International Journal On Semantic Web and Information Systems 5(2), 1–24 (2009)
6. Calì, A., Gottlob, G., Kifer, M.: Taming the infinite chase: Query answering under expressive relational constraints. In: Proc of the 11th Int Conf on Principles of Knowledge Repres and Reasoning. pp. 70–80 (2008)
7. Chabin, J., Halfeld-Ferrari, M., Nguyen, T.B.: Querying Semantic Graph Databases in View of Constraints and Provenance. Tech. rep., LIFO- Université d'Orléans, RR-2016-02 (2016)
8. Flouris, G., Konstantinidis, G., Antoniou, G., Christophides, V.: Formal foundations for RDF/S KB evolution. Knowl. Inf. Syst. 35(1), 153–191 (2013)
9. Frommhold, M., Piris, R.N., Arndt, N., Tramp, S., Petersen, N., Martin, M.: Towards Versioning of Arbitrary RDF Data. In: Proc of the 12th Int Conf on Semantic Systems Proceedings (2016)
10. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Web Semant. 3(2-3), 158–182 (2005)
11. Halfeld-Ferrari, M., Laurent, D., Spyratos, N.: Update rules in datalog programs. J. Log. Comput. 8(6), 745–775 (1998)
12. Hansson, S.O.: Logic of belief revision. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, winter 2016 edn. (2016)
13. Knublauch, H., Ryman, A.: Shapes constraint language (SHACL). W3C first public working draft, w3c. http://www.w3.org/TR/2015/WD-shacl-20151008/. (2017)
14. Liu, L., Özsu, M.T. (eds.): Encyclopedia of Database Systems. Springer US (2009)
15. Magiridou, M., Sahtouris, S., Christophides, V., Koubarakis, M.: RUL: A declarative update language for RDF. In: Proc of the 4th Int Semantic Web Conference. pp. 506–521 (2005)
16. Solbrig, H., hommeaux, E.P.: Shape expressions 1.0 definition. W3C member submission. http://www.w3.org/Submission/2014/SUBM-shex-defn-20140602 (2014)
17. Stardog5: Enterprise knowledge graph. http://www.stardog.com/docs/ (2017)
18. Uber, F.: RDF constraint satisfaction with blank nodes. PhD Dissertation Proposal, UFPR, Brazil, `http://www.inf.ufpr.br/fruber/BNS` (2016)