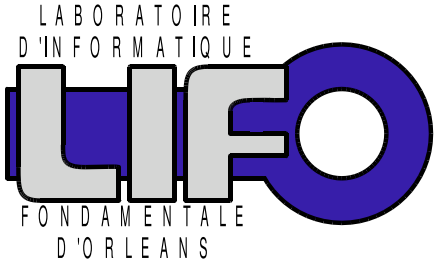

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Updating of RDF/S Databases under Negative and Tuple-Generating Constraints

Mirian Halfeld Ferrari and Dominique Laurent
LIFO, Université d'Orléans

Rapport n^o **RR-2017-05**

Updating of RDF/S Databases under Negative and Tuple-Generating Constraints

Mirian Halfeld Ferrari
Université d'Orléans, INSA CVL - LIFO EA
Orléans - France
mirian@univ-orleans.fr

Dominique Laurent
ETIS - Université Paris Seine - CNRS
Cergy-Pontoise - France
dominique.laurent@u-cergy.fr

ABSTRACT

In this paper, we address the issue of updating RDF/S databases, in which constraints are imposed. Contrary to standard approaches where constraints are restricted to those inherently defined by the data model, we also consider constraints imposed by the particular application modelled by the database. All these constraints fall in two categories, called *positive* and *negative*, generalizing the well known key-foreign key constraints.

Based on a chasing technique, we propose a deterministic update strategy which deals with sets of insertions and deletions over RDF/S instances, while satisfying consistency and minimal change requirements. The time complexity of our approach is polynomial.

1. INTRODUCTION

The maintenance of data consistency is an essential problem in databases. In modern scenarios, constraint verification has been neglected in the name of change velocity. But constraints *still* are an imperative quality label – the only way to ensure the reliability of database applications.

The Resource Description Framework (RDF) is a flexible graph-based data model, not only restricted to the semantic web, but now largely adopted for data publishing and sharing ([19]). The increasing number of distributed RDF datasets brings the need of providing reliable information to a user worrying about data quality and validity. In this scenario, it is relevant to consider centralized solutions that we refer to as integration systems (also called warehousing solutions in [16]). In these systems, *needed data* extracted from distributed sources is stored in an integrated database where consistency must be preserved. Then, in some practical situations (as in ESCO-Portail of the GIP RECIA Project[1]), updating the integrated database and propagating the updates towards the sources may be necessary.

Consistent updating of an RDF integrated database is the core of our work. Consistency is a quality certificate, and restrictions imposed to the data allow to customize this expected quality. For this reason, we deal with constraints as in a traditional database viewpoint, and not as so-called ontological constraints ([7]), seen as inference rules. To illustrate this important point, consider a constraint stating that all papers must be published in a journal. Then, in our approach a database containing the only fact $\mathsf{Paper}(rdfP)$ is *not* consistent, whereas it is consistent when constraints are seen as inference rules.

We propose a consistency preserving update method where RDF/S semantic constraints and application restrictions are treated in the same manner. We distinguish between schema and instance updates, focusing on the latter task. Our method ensures deterministic updates, while avoiding arbitrary choices. Constraints generate side effects, defining other updates that maintain the database consistency.

Considering linear Tuple Generating Dependencies (TGDs) and negative constraints in the datalog$^{\pm}$ language ([2]), along with database instances not containing blank nodes, our main contributions are the following:
(1) A *deterministic* update strategy ensuring database *consistency* with respect to a set of constraints and satisfying *minimal change* requirements.
(2) Constraints as *active rules* ([9]). The insertion or deletion of a set of facts triggers rules and generates *side effects*.
(3) A *uniform* treatment for constraints coming either from a specific application or from RDF/S semantics.
(4) The possibility of dealing with a set of insertions and deletions as an *update transaction*.
(5) A polynomial time two step approach where the first step generates templates *independently* from the instance.

Paper organization: After a motivating example (Section 2) and some background definitions (Section 3), in Section 4 we introduce our update method. In Section 5 we show how application and RDF/S constraints can be treated together. Sections 6 and 7 focus on related work and perspectives.

2. RUNNING EXAMPLE

As illustrated in Figure 1, we consider an integrated RDF/S database defined by a schema, a set of constraints $\mathcal{C}$ and an instance such that: (1) The instance contains information collected from data sources through a mediator, and explicitly stored. (2) The instance is valid with respect to the schema and the constraints.

We consider updates on the integrated RDF/S database instance, and we notice that in this setting, updated data can be propagated to sources, according to update routines.

[1] `https://github.com/GIP-RECIA/esco-portail`

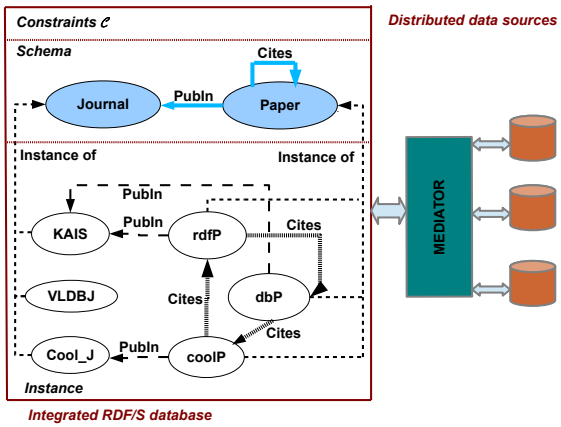

Figure 1: Global integration system

Paper(dbP), Paper($rdfP$), Paper($coolP$)
Journal($KAIS$), Journal($VLDBJ$), Journal($Cool\_J$)
PubIn($dbP, KAIS$), PubIn($rdfP, KAIS$), PubIn($coolP, Cool\_J$)
Cites($dbP, coolP$), Cites($rdfP, dbP$), Cites($coolP, rdfP$)

Figure 2: Facts in the RDF/S instance

RDF is a language for expressing facts, where each fact consists of a triple made of a subject, a predicate and an object. An RDF triple $\langle aPb \rangle$ is interpreted as the first-order logic (FOL) fact $P(a, b)$ where P is a predicate and a and b constants. An RDF/S database may be seen as an RDF graph instance associated to a graph schema, which describes resource relationships via properties. The upper part of the integrated system shown in Figure 1 depicts the RDF/S schema while the RDF/S instance is shown in the lower part (this example is borrowed from [6]). It can be seen that this instance can be expressed as a set of facts as shown in Figure 2 where the predicates have the following intuitive meaning: Paper (respectively Journal) deals with paper titles (respectively journal names), PubIn informs on the journal a paper is published in and Cites informs on who cites whom. We also assume the following constraints in $\mathcal{C}$:

- $c_1 : (\forall x, y)(\mathsf{PubIn}(x, y) \Rightarrow \mathsf{Paper}(x))$
- $c_2 : (\forall x, y)(\mathsf{PubIn}(x, y) \Rightarrow \mathsf{Journal}(y))$
- $c_3 : (\forall x, y)(\mathsf{Cites}(x, y) \Rightarrow \mathsf{Paper}(x))$
- $c_4 : (\forall x, y)(\mathsf{Cites}(x, y) \Rightarrow \mathsf{Paper}(y))$
- $c_5 : (\forall x)(\mathsf{Paper}(x) \Rightarrow (\exists y)(\mathsf{PubIn}(x, y)))$
- $c_6 : (\forall x, y, z)((y \neq z) \wedge \mathsf{PubIn}(x, y) \wedge \mathsf{PubIn}(x, z) \Rightarrow \bot)$
- $c_7 : (\forall x)(\mathsf{Paper}(x) \Rightarrow (\exists y)(\mathsf{Cites}(x, y)))$
- $c_8 : (\forall x, y)((x = y) \wedge \mathsf{Cites}(x, y) \Rightarrow \bot)$

The intuitive meaning of these constraints is as follows: papers are published in journals (c_1 and c_2) and cite papers (c_3 and c_4); each paper must be published (c_5), but a paper cannot be published in different journals (c_6); each paper must cite at least one paper (c_7) but cannot cite itself (c_8). Notice that the graph of Figure 1 respects these constraints.

We explain our approach to updates through two examples: one for deletion and one for insertion. Regarding the deletion example, we assume as in [6], that the deletion of $\mathsf{PubIn}(coolP, Cool\_J)$ is requested. As noticed in [6], simply removing this fact from the database implies that constraints are violated and, unfortunately, several ways of preserving consistency are possible, implying that a choice must be made. More precisely, the options are as follows:

1. One can decide that $coolP$ is actually published in another journal say J. Then, replacing $\mathsf{PubIn}(coolP, Cool\_J)$ with $\mathsf{PubIn}(coolP, J)$ in the database restores consistency. However, the problem is to determine the value of J which is *not* specified in the update.
2. One can decide that $\mathsf{PubIn}(coolP, Cool\_J)$ has to be deleted with no replacement. In this case, in order to satisfy c_5, $\mathsf{Paper}(coolP)$ has to be deleted, and so, in order to satisfy c_3 and c_4, $\mathsf{Cites}(coolP, rdfP)$ and $\mathsf{Cites}(dbP, coolP)$ have to be deleted as well.

In [6] the authors argue that option 1 is preferable to option 2 because the former implies less side effects than the latter. However, in order to be deterministic, option 1 requires to know which journal to choose. To cope with this problem, a specific total ordering, called change-generating ordering is assumed. For instance, this ordering might stipulate that $VLDBJ$ is 'better' than $KAIS$, in which case $J = VLDBJ$ is chosen. However, such comparison is clearly arbitrary, depends on the update, and might not be suitable since a new journal can be involved in the fact to be inserted.

In order to cope with the basic problem of determinism while avoiding arbitrary choices, we argue that option 2 is preferable to option 1 because: (i) no further assumption than the specified update is needed, (ii) the update is actually performed, (iii) the consistency with respect to constraints is preserved, and (iv) the minimal change requirement is satisfied, in the sense that canceling one of the update side effects does not preserve consistency.

Our approach allows to follow option 1 if the journal J is explicitly chosen by the user. This amounts to consider that the two updates (1) delete $\mathsf{PubIn}(coolP, Cool\_J)$ and (2) insert $\mathsf{PubIn}(coolP, VLDBJ)$ have to be processed *at the same time*. Such sets of updates, called change requests in [6], are referred to as *global updates* in our approach.

As for insertions in the framework of our running example, assume first that the insertion of $\mathsf{PubIn}(coolP, VLDBJ)$ is requested in the instance of Figure 2. In this case, because of c_6, the deletion of all facts of the form $\mathsf{PubIn}(coolP, J)$ where $J \neq VLDBJ$ is necessary to preserve database consistency. This insertion thus requires the deletion of $\mathsf{PubIn}(coolP, Cool\_J)$, generating the same updates as above.

Consider now the insertion of $\mathsf{Paper}(newP)$. Due to c_5 and c_7, the insertion requires to also insert $\mathsf{PubIn}(newP, J)$ and $\mathsf{Cites}(newP, P)$ where J (respectively P) stands for an arbitrary journal (respectively an arbitrary paper, other than $newP$ due to c_8). We are thus facing a problem of non determinism as above, and as we do not assume any further information (contrary to [6], with the change-generating ordering), the insertion is rejected. However, in our approach it is possible to warn the user about the missing values, in which case the requested update could be transformed as the insertion of $\mathsf{Paper}(newP)$, $\mathsf{PubIn}(newP, KAIS)$ and $\mathsf{Cites}(newP, coolP)$, which is processed with no side effects.

3. BACKGROUND

3.1 Basic Notation

Alphabet. Let $\mathbf{A}$ be an alphabet consisting of the following pairwise disjoint sets: $\mathbf{A}_C$, a countably infinite set

of constant; $\mathbf{A}_N$, a countably infinite set of labelled nulls which are placeholders for unknown values; VAR an infinite set of variables ranging over $\mathbf{A}_C \cup \mathbf{A}_N$ (we use $\mathbf{X}$ as an abbreviation for $X_1 \dots X_k$ where $k > 0$ or to denote the set $\{X_1, \dots, X_k\}$); PRED, a *finite* set of predicates, each being associated with a positive number called its arity.

Since we consider a function-free language, the only possible terms are constants, nulls or variables. An *atomic formula* (or *atom*) has one of the forms: (i) $P(t_1, ..., t_n)$, where P is an n-ary predicate and $t_1, ..., t_n$ are terms; (ii) $\top$ (meaning true) or $\bot$ (meaning false); (iii) $(t_1\ op\ t_2)$, where t_1 and t_2 are terms and op is a comparison operator ($=$, $<$, $>$, $\leq$, $\geq$) assuming that such comparison makes sense. A *literal* is an atom of the form $P(t_1, ..., t_n)$ (*i.e.,* a positive atom) or $\neg P(t_1, ..., t_n)$ (*i.e.,* a negative atom). Given a predicate P in PRED, an *instantiated literal* over P is an atom of the form $P(u)$ where $u \in (\mathbf{A}_C \cup \mathbf{A}_N)^n$, and a *fact* over P is an atom of form $P(u)$ where $u \in (\mathbf{A}_C)^n$.

Substitution. A *substitution* from the set of symbols E_1 to the set of symbols E_2 is a function $h : E_1 \to E_2$. A *homomorphism* from the set of atoms A_1 to the set of atoms A_2, both over the same predicate P, is 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 $P(h(t_1), ..., h(t_n)) \in A_2$.

A substitution that associates the variable x with the constant a and the variable y with the null N, is denoted as the set $\{x/a, y/N\}$. Moreover, if h is a homomorphism, in order to simplify notation, $P(h(t_1), ..., h(t_n))$ is denoted by $h(P(t_1, ..., t_n))$. It should also be clear that the notion of homomorphism naturally extends to conjunctions of atoms.

The following terminology is also used: (i) An *endomorphism* h on a finite set of atoms A_1 is a homomorphism such that $h(A_1) \subseteq A_1$; (ii) A *valuation* is a homomorphism whose image of each symbol is a constant in $\mathbf{A}_C$.

3.2 Database Instance and Constraints

Definition 1. Let $\mathbf{A}$ be an alphabet. We define two kinds of constraints, respectively referred to as *positive constraints* and *negative constraints* as follows:

1. A *positive constraint* is a formula of form:
   – $(\forall \mathbf{X}, \mathbf{Y})(L_1(\mathbf{X}, \mathbf{Y}) \Rightarrow (\exists \mathbf{Z})(L_2(\mathbf{X}, \mathbf{Z})))$
   where $L_1(\mathbf{X}, \mathbf{Y})$ and $L_2(\mathbf{X}, \mathbf{Z})$ are positive atoms.
2. A *negative constraint* is a formula of one of the following two forms:
   – $(\forall \mathbf{X})((comp(\mathbf{X}') \wedge L(\mathbf{X})) \Rightarrow \bot)$ or
   – $(\forall \mathbf{X})((comp(\mathbf{X}') \wedge L_1(\mathbf{X}_1) \wedge L_2(\mathbf{X}_2)) \Rightarrow \bot)$
   where $\mathbf{X}_1 \cap \mathbf{X}_2 \neq \emptyset$ and $comp(\mathbf{X}')$ is a (possibly empty) formula involving comparison atoms with variables $\mathbf{X}'$ that all occur in $\mathbf{X}$, and where $L(\mathbf{X})$, $L_1(\mathbf{X}_1)$ and $L_2(\mathbf{X}_2)$ are atoms.

Given a constraint c, the left hand-side and the right hand-side of c are respectively denoted by $body(c)$ and $head(c)$. When no confusion is possible, quantifiers are omitted. □

We emphasize that positive constraints are a special case of linear TGD (Tuple Generating Dependency) which contain only one atom in the head, whereas negative constraints specify conditions under which facts cannot be in the database instance. We recall that these constraints generalize key-foreign key constraints in relational databases. We also notice that positive constraints can be seen as update rules as defined in [9], with the following important differences: (i) only positive literals are allowed in a positive constraint (whereas positive and negative literals are allowed in update rules) and (ii) existential variables are possible in a positive constraint head (which is not possible in update rules). Constraint satisfaction is defined as follows.

Definition 2. Let I be a set of instantiated atoms and c a constraint as in Definition 1. I *satisfies* c, denoted by $I \models c$, if for every homomorphism h from the variables in $body(c)$ into constants or nulls in I, the following holds:

- If c is positive: if $h(body(c))$ is in I, then there is an extension h' of h such that $h'(head(c))$ is in I.
- If c is negative: if $h(comp(\mathbf{X}'))$ is true in I then depending on the form of c, either $h(L(\mathbf{X}))$ is not in I or at least one of the two ground literals $h(L_1(\mathbf{X}_1))$ or $h(L_2(\mathbf{X}_2))$ is not in I.

Given a *set* of constraints $\mathcal{C}$, I satisfies $\mathcal{C}$, denoted by $I \models \mathcal{C}$, if for every c in $\mathcal{C}$, $I \models c$ holds. □

Definition 3. Let $\mathbf{A}$ be an alphabet. A database instance over $\mathbf{A}$ is a pair $\Delta = (D, \mathcal{C})$ where D is a set of facts over $\mathbf{A}$ and $\mathcal{C}$ is a set of constraints such that $D \models \mathcal{C}$. □

As a consequence of Definition 3, we notice that in our approach, blank nodes, or equivalently facts with labelled nulls, are *not* allowed in a database instance.

As will be seen later, computing side effects of updates when only positive constraints are considered relies on the notion of *trigger* that we borrow from [15].

Definition 4. Let $c : L_1(\mathbf{X}, \mathbf{Y}) \Rightarrow L_2(\mathbf{X}, \mathbf{Z})$ be a positive constraint and I a set of instantiated atoms. A triple (c, h_1, h_2) is *a trigger in* I if there exists $L_1(\alpha, \beta)$ in I such that $h_1(L_1(\mathbf{X}, \mathbf{Y})) = h_2(L_1(\alpha, \beta))$ where h_1 is a homomorphism from VAR to $\mathbf{A}_C \cup \mathbf{A}_N$ and h_2 is an endomorphism on $\mathbf{A}_C \cup \mathbf{A}_N$. □

When there is a trigger in I for a positive constraint c, we say that c is *activated* to produce a new instantiated atom. In this case, when existential variables $\mathbf{Z}$ are present in $head(c)$, if (c, h_1, h_2) is a trigger in I then we define an extension h_1' of h_1 ($h_1' \supseteq h_1$) such that, for every Z_i in $\mathbf{Z}$, $h_1'(Z_i) = N_i$, where N_i is a fresh labelled null in $\mathbf{A}_N$ not introduced before.

Example 1. Considering the two positive constraints $c_1 : A(x, x) \Rightarrow B(x)$ and $c_2 : B(x) \Rightarrow D(x, y)$, let $I_0 = \{A(a, b), A(a, N_1), A(b, b)\}$ be a set of instantiated atoms where N_1 is in $\mathbf{A}_N$. The following two triggers exist in I_0:

1. (c_1, h_1^1, h_2^1) where $h_1^1 = \{x/a\}$ and $h_2^1 = \{N_1/a\}$, since $A(a, N_1)$ is in I and $h_1^1(A(x, x)) = h_2^1(A(a, N_1)) = A(a, a)$. This trigger activates c_1 to produce $B(a)$.
2. (c_1, h_1^2, h_2^2) where $h_1^2 = \{x/b\}$ and $h_2^2 = \emptyset$, since $A(b, b)$ is in I and $h_1^2(A(x, x)) = h_2^2(A(b, b)) = A(b, b)$. This trigger activates c_1 to produce $B(b)$.

Notice that, since $head(c_1)$ has no existential variables, no extensions of h_1^1 or of h_1^2 are needed in order to define the atoms produced by the previous two triggers.

Now, let $I_1 = I_0 \cup \{B(a), B(b)\}$. In this case, the two triggers above exist in I_1 but do not produce new atoms. On the other hand, two new triggers exist in I_1, namely:

1. (c_2, h_1^1, h_2^2) where $h_1^1 = \{x/a\}$ and $h_2^2 = \emptyset$, which activates c_2 to produce $D(a, N_2)$ where N_2 is a fresh null in $\mathbf{A}_N$. In this case the extension $(h_1^1)'$ of h_1^1 such that $(h_1^1)' = \{x/a, y/N_2\}$ is needed.
2. (c_2, h_1^2, h_2^2) where $h_1^2 = \{x/b\}$ and $h_2^2 = \emptyset$, which activates c_2 to produce $D(b, N_3)$ where N_3 is a fresh null in $\mathbf{A}_N$. In this case the extension $(h_1^2)'$ of h_1^2 such that $(h_1^2)' = \{x/b, y/N_3\}$ is needed.

Considering now $I_2 = I_1 \cup \{D(a, N_2), D(b, N_3)\}$, no new trigger exists in I_2. It can be seen that I_2 is the result of the *chase algorithm* of [15], applied to I_0, c_1 and c_2. □

4. UPDATES

In this section, we first consider separately the insertion or the deletion of sets of facts in the case where the constraints are restricted to be positive. Then, we consider the general case of arbitrary sets of updates, combining insertions and deletions in the presence of positive and negative constraints. Before going into the details of our approach, we state the main requirements updates must satisfy.

Given a database instance $\Delta = (D, \mathcal{C})$, we assume two sets *IReq* and *DReq* specifying respectively insertions and deletions. Updating Δ with respect to *IReq* and *DReq* means either finding that the requested updates are not possible or building a new database instance $\Delta' = (D', \mathcal{C})$ such that:

1. *Whenever possible, the updates are performed:* $IReq \subseteq D'$ and $DReq \cap D' = \emptyset$.
   If the updates cannot be performed, Δ is not changed.
2. *The resulting database is consistent:* $D' \models \mathcal{C}$.
3. *The process is deterministic:* Constraint satisfaction is enforced through the computation of side effects, using no information other than Δ, *IReq* and *DReq*.
4. *The minimal change requirement is satisfied:* Let $ISet = D' \setminus D$ and $DSet = D \setminus D'$. For every φ in $ISet \setminus IReq$ (respectively in $DSet \setminus DReq$), $D' \setminus \{\varphi\}$ (respectively $D' \cup \{\varphi\}$) does *not* satisfy $\mathcal{C}$.

As will be seen next, deletions are always possible, whereas some insertions might be rejected. Moreover, the insertion or deletion only of a set of facts in the presence of positive constraints is defined as a two step processing as follows:

1. Build a chasing tableau using the constrains in $\mathcal{C}$ and the given set of updates (either *IReq* or *DReq*).
2. Implement the update using the tableau of the previous step and the database instance Δ.

4.1 Insertions under Positive Constraints

Here we assume we are given a database instance $\Delta = (D, \mathcal{C})$ such that $\mathcal{C}$ contains only *positive* constraints along with a set of facts I meant to be inserted into Δ.

To perform the insertions we first build up a tableau denoted by T-INS as the union of two sub-tableaux: C-INS(I) whose rows contain constants only (and no labelled null), and N-INS(I) in which each row contains at least one labelled null. This construction, which is achieved by Algorithm 1, is inspired by that of chasing tableaux as defined in [15].

We now explain the main steps of Algorithm 1. The tableaux C-INS(I) and N-INS(I) are initialized on lines 1 and 2, and then, facts in C-INS(I) are used to trigger constraints in $\mathcal{C}$ (line 7). While such triggers exist and facts

Algorithm 1: $BuildInsTab(I, \mathcal{C}, \text{T-INS})$

Input: A set of positive constraints $\mathcal{C}$ and a set of facts I.
Output: The tableau T-INS = C-INS(I) ∪ N-INS(I).

```
1:  C-INS(I) := I
2:  N-INS(I) := ∅
3:  continue := true
4:  while continue do
5:      // continue is false when no constant row is added
6:      continue := false
7:      for all c having a trigger (c, h1, h2) in C-INS(I) do
8:          // c is written as c : L1(X, Y) ⇒ L2(X, Z)
9:          Let h1' be an extension of h1 such that for each
            Zi ∈ Z, h1'(Zi) = Ni where Ni is a fresh labelled
            null in A_N not introduced before
10:         if h1'(L2(X, Z)) contains no null then
11:             if h1'(L2(X, Z)) ∉ C-INS(I) then
12:                 C-INS(I) := C-INS(I) ∪ {h1'(L2(X, Z))}
13:                 continue := true
14:         else
15:             N-INS(I) := N-INS(I) ∪ {h1'(L2(X, Z))}
16: return T-INS = C-INS(I) ∪ N-INS(I)
```

are produced (line 10), the *while* loop line 4 goes on. The loop stops when only instantiated atoms containing nulls are generated (line 15). The two tableaux are returned (line 16) as the tableau T-INS. The following proposition shows that Algorithm 1 always terminates.

PROPOSITION 1. *For every finite set of positive constraints $\mathcal{C}$ and every finite set of facts I, Algorithm 1 applied to $\mathcal{C}$ and I always terminates.*

PROOF. Since I is finite, so is the number of potential facts to add in C-INS(I), which shows the proposition. □

Notice that Proposition 1 implies that our approach applies even in the case of constraints having a non weakly acyclic graph, according to the terminology of [4] or [15].

Algorithm 1 is the first step of our processing, and we emphasize that this step does *not* require to access the database instance. The tableau obtained by Algorithm 1 is used in a second step as stated in Algorithm 2.

Algorithm 2: $PerfIns(\Delta, IReq)$

Input:
- $\Delta = (D, \mathcal{C})$ where $\mathcal{C}$ is a set of positive constraints.
- *IReq* the set of facts to be inserted.

Output: The updated database instance $\Delta' = (D', \mathcal{C})$.

```
1: BuildInsTab(IReq, C, T-INSIReq))
2: if there is a valuation v of the nulls in N-INS(IReq)
   such that v(N-INS(IReq)) ⊆ (D ∪ C-INS(IReq)) then
3:     D' := D ∪ C-INS(IReq)
4: else
5:     // The insertion is not possible
6:     D' := D
7: return Δ' = (D', C)
```

Intuitively, according to Algorithm 2, the insertions are performed only when the nulls in N-INS(*IReq*) can be instantiated into facts either in the database instance or in the set C-INS(*IReq*); otherwise, the database instance is not

changed. The following examples illustrate this important feature of our approach.

Example 2. In the context of our running example, let Δ_P be the database instance $(D, \mathcal{C}_P)$ where D is the set shown in Figure 2 and where $\mathcal{C}_P$ contains the positive constraints c_1–c_5 and c_7, recalled below in their simplified form.

$c_1 : \mathsf{PubIn}(x, y) \Rightarrow \mathsf{Paper}(x)$ $\quad c_2 : \mathsf{PubIn}(x, y) \Rightarrow \mathsf{Journal}(y)$
$c_3 : \mathsf{Cites}(x, y) \Rightarrow \mathsf{Paper}(x)$ $\quad c_4 : \mathsf{Cites}(x, y) \Rightarrow \mathsf{Paper}(y)$
$c_5 : \mathsf{Paper}(x) \Rightarrow \mathsf{PubIn}(x, y)$ $\quad c_7 : \mathsf{Paper}(x) \Rightarrow \mathsf{Cites}(x, y)$

For $IReq = \{\mathsf{Paper}(newP)\}$, when running Algorithm 1, we find two triggers involving c_5 and c_7, producing respectively the atoms $\mathsf{PubIn}(newP, N_1)$ and $\mathsf{Cites}(newP, N_2)$. Since no fact is generated, the algorithm stops, returning T-INS $= \{\mathsf{Paper}(newP)\} \cup \{\mathsf{PubIn}(newP, N_1), \mathsf{Cites}(newP, N_2)\}$. As no instantiation of atoms in N-INS($IReq$) is in $D \cup$ C-INS($IReq$), Δ_P is not changed by Algorithm 2.

The case above illustrates that, in our update processing, contrary to [6], we do not make any choice regarding missing information with respect to constraints, but we rather reject the update. However, instead of rejecting, the tableau N-INS($IReq$) can be used by the user to provide the missing information. In our example, the tableau N-INS($IReq$) can suggest the following update request:

$$IReq = \{\mathsf{Paper}(newP), \mathsf{PubIn}(newP, KAIS), \mathsf{Cites}(newP, coolP)\}.$$

In this case of new request, Algorithm 1 computes the tableau T-INS = C-INS($IReq$) $\cup$ N-INS($IReq$) where

$$\text{C-INS}(IReq) = \{\mathsf{Paper}(newP), \mathsf{PubIn}(newP, KAIS), \mathsf{Cites}(newP, coolP), \mathsf{Journal}(KAIS), \mathsf{Paper}(coolP)\}$$
$$\text{N-INS}(IReq) = \{\mathsf{PubIn}(newP, N_1), \mathsf{Cites}(newP, N_2), \mathsf{PubIn}(coolP, N_3), \mathsf{Cites}(coolP, N_4)\}.$$

Since $N_1, \ldots, N_4$ can be instantiated in $IReq \cup D$, the insertions are processed as in our running example. □

Clearly, our approach satisfies our requirements that insertions are performed in a deterministic way and require no information other than Δ and the facts to be inserted.

The following proposition shows that, when the instance is changed by Algorithm 2, the insertions are actually performed and that the minimal change requirement is satisfied.

PROPOSITION 2. *Let $\Delta = (D, \mathcal{C})$ be a database instance and IReq a finite set of facts. If Algorithm 2 does not execute the* else *statement line 4, then $\Delta' = (D', \mathcal{C})$ is such that:*

1. *D' contains IReq and $D' \models \mathcal{C}$.*
2. *For every φ in* C-INS($IReq$) $\setminus$ *IReq, $D' \setminus \{\varphi\} \not\models \mathcal{C}$.*

PROOF. 1. For every set of facts (without nulls) I, we clearly have $I \subseteq$ C-INS(I), and so $IReq \subseteq D'$ holds.

Assume that D' does not satisfy $c : L_1(\mathbf{X}, \mathbf{Y}) \Rightarrow L_2(\mathbf{X}, \mathbf{Z})$. Thus D' contains a fact of the form $L_1(\alpha, \beta)$ but no fact of the form $L_2(\alpha, \gamma)$. As D satisfies $\mathcal{C}$, $L_1(\alpha, \beta)$ is not in D and so this fact belongs to C-INS($IReq$). Hence, c has a trigger in C-INS($IReq$) which does not change C-INS($IReq$) (otherwise, the algorithm would have continued and produced a different result). As a consequence, when applying the trigger, we obtain the atom $L_2(\alpha, N)$ that is put in N-INS($IReq$). Since the insertion is accepted, $D \cup$ C-INS($IReq$) contains an instantiation of $L_2(\alpha, N)$ which is a contradiction with the hypothesis that $D' = D \cup$ C-INS($IReq$) contains no fact of the form $L_2(\alpha, \gamma)$.

2. Since φ is in C-INS($IReq$) $\setminus$ $IReq$, φ is an instantiation of the head of a constraint c in $\mathcal{C}$ and an instantiation b of the body of c appears in C-INS($IReq$). As above, assuming that $c : L_1(\mathbf{X}, \mathbf{Y}) \Rightarrow L_2(\mathbf{X}, \mathbf{Z})$, $b = L_1(\alpha, \beta)$ and $\varphi = L_2(\alpha, \gamma)$. By contraposition, let us assume that all constraints in $\mathcal{C}$ are satisfied by $D' \setminus \{\varphi\}$. In this case, c is satisfied by $D' \setminus \{\varphi\}$, meaning that D' contains a fact of the form $L_2(\alpha, \gamma')$ with $\gamma' \neq \gamma$. In this case, when applying c with $L_1(\alpha, \beta)$, the atom $L_2(\alpha, N)$ contains at least one null, which is a contradiction with the fact that φ is in C-INS($IReq$). □

4.2 Deletions under Positive Constraints

We first motivate our approach to deletions.

Example 3. Consider the database instance $\Delta_7 = (D_7, \mathcal{C}_7)$ with $\mathcal{C}_7 = \{c_7 : \mathsf{Paper}(x) \Rightarrow \mathsf{Cites}(x, y)\}$ and D_7 such that:

$$D_7 = \{\mathsf{Paper}(rdfP), \mathsf{Paper}(dbP), \mathsf{Paper}(coolP), \mathsf{Cites}(rdfP, coolP), \mathsf{Cites}(dbP, coolP), \mathsf{Cites}(dbP, rdfP), \mathsf{Cites}(coolP, dbP)\}.$$

If $\mathsf{Cites}(dbP, coolP)$ is to be deleted, then $(D_7 \setminus \{\mathsf{Cites}(dbP, coolP)\}, \mathcal{C}_7)$ satisfies c_7 and so, is the result of the deletion.

On the other hand, when deleting $\mathsf{Cites}(coolP, dbP)$, $D_7 \setminus \{\mathsf{Cites}(coolP, dbP)\}$ does *not* satisfy c_7. In order to maintain satisfaction, we also have to delete $\mathsf{Paper}(coolP)$.

To take into account these remarks, we define a tableau, denoted by T-DEL with the following three columns:

1. Column Del contains a (partial) instantiation of an atom to be deleted;
2. Column Query contains an instantiated literal that results from a constraint c applied to the Del atom;
3. Column Ref references to the lines in T-DEL from which the current line is constructed.

Here, the two deletions give similar tableaux that we summarize below, denoting the fact to be deleted as $\mathsf{Cites}(\alpha, \beta)$.

	Del	Query	Ref
1	$\mathsf{Cites}(\alpha, \beta)$	–	–
2	$\mathsf{Paper}(\alpha)$	$\mathsf{Cites}(\alpha, N_1)$	1

This tableau is obtained through the following steps:

- The fact to be deleted is placed in the first column of the first row, *i.e.,* T-DEL$[1, \mathsf{Del}] = \mathsf{Cites}(\alpha, \beta)$, where (α, β) is either $(dbP, coolP)$ or $(coolP, db\_P)$.
- Every constraint is triggered 'backwards' (*i.e.,* from the head to the body) using the fact in T-DEL$[1, \mathsf{Del}]$. Here, the unique constraint is processed as follows:
  1. Applying the constraint backwards on $\mathsf{Cites}(\alpha, \beta)$ generates a new line numbered 2 and T-DEL$[2, \mathsf{Del}]$ is set to $\mathsf{Paper}(\alpha)$.
  2. This constraint, now applied forwards, produces $\mathsf{Cites}(\alpha, N_1)$, which is stored in T-DEL$[2, \mathsf{Query}]$.
  3. Since this line has been obtained from T-DEL$[1, \mathsf{Del}]$, 1 is stored in T-DEL$[2, \mathsf{Ref}]$.

Using the tableau shown above, T-DEL$[2, \mathsf{Query}]$ is seen as a query processed against the database instance. In our example, when deleting the fact $\mathsf{Cites}(dbP, coolP)$ (respectively $\mathsf{Cites}(coolP, dbP)$) the query asks for all facts of the form $\mathsf{Cites}(dbP, N_1)$ (respectively $\mathsf{Cites}(coolP, N_1)$). The number of facts in the answer allows to decide whether the instantiated body $\mathsf{Paper}(dbP)$ (respectively $\mathsf{Paper}(coolP)$) has to

be deleted or not. As explained above, for the first deletion, $\mathsf{Cites}(dbP, coolP)$ and $\mathsf{Cites}(dbP, rdfP)$ are returned and thus, no other deletion is needed; for the second deletion, $\mathsf{Cites}(coolP, dbP)$ is retuned, implying that $\mathsf{Paper}(coolP)$ must be deleted. □

As suggested by the previous example, our deletion processing follows two steps, in much the same way as insertions. The first step, *i.e.,* the construction of a tableau denoted T-DEL, is performed according to Algorithm 3.

It should be noticed that in this algorithm, as well as in the remainder of this paper, every positive constraint $c : L_1(\mathbf{X}, \mathbf{Y}) \Rightarrow L_2(\mathbf{X}, \mathbf{Z})$ is associated with its 'backward' or inverse form $\overline{c} : L_2(\mathbf{X}, \mathbf{Z}) \Rightarrow L_1(\mathbf{X}, \mathbf{Y})$.

Algorithm 3: *BuildDelTab*$(\mathcal{C}, I)$

Input:
- $\mathcal{C}$, a set of positive constraints
- I, a set of facts.

Output: The tableau T-DEL

```
1: i := 0;
2: for all φ ∈ I do
3:     i := i + 1
4:     T-DEL[i, Del] := φ
5: N := i + 1
6: endline := N
7: startline := 1
8: while startline < endline do
9:     for i := startline to (endline − 1) do
10:        for all trigger (c̄, h1, h2) with respect to
           T-DEL[i, Del] do
11:            Let h1' be an extension of h1 to the variables
               of Y built from the application of the trigger
               (c̄, h1, h2)
12:            Denoting by h1'[XY] the restriction of h1' to
               the variables in X or in Y, let h1''[XY] be an
               extension of h1'[XY], obtained by the
               application of c with h1'[XY](L1(X, Y))
13:            IsoRow :=
                 Iso(h1'[XY](L1(X, Y)), h1''[XY](L2(X, Z)))
14:            if IsoRow = ∅ then
15:                T-DEL[N, Del] := h1'[XY](L1(X, Y))
16:                T-DEL[N, Query] := h1''[XY](L2(X, Z))
17:                T-DEL[N, Ref] := i
18:                N := N + 1
19:            else
20:                // Assume that IsoRow = {ρ}
21:                Add i in the list stored in T-DEL[ρ, Ref]
22:    startline := endline
23:    endline := N
24: return T-DEL
```

The initialization of T-DEL is done lines 2−4 by copying each fact to be deleted in the column Del of a new row of the tableau. Then, the loop starting line 8 processes the rows just added in the tableau, that is the rows numbered from *startline* up to *endline* − 1, so as to construct new rows based on the constraints in $\mathcal{C}$.

To this end, for $i = startline, \ldots, (endline - 1)$, triggers with respect to T-DEL$[i, \mathsf{Del}]$ and involving an inverse constraint $\overline{c} : L_2(\mathbf{X}, \mathbf{Z}) \Rightarrow L_1(\mathbf{X}, \mathbf{Y})$ are detected and, for each of them, instructions lines 10–21 are performed:

- First, in line 11, for each trigger $(\overline{c}, h_1, h_2)$ such that $h_1(L_2(\mathbf{X}, \mathbf{Z})) = h_2(\text{T-DEL}[i, \mathsf{Del}])$, an extension of h_1 is defined so as to generate the new atom $h_1'(L_1(\mathbf{X}, \mathbf{Y}))$.
- Then, denoting by $h_1'[\mathbf{XY}]$ the restriction of h_1' to the variables in $\mathbf{X}$ or in $\mathbf{Y}$, we have $h_1'[\mathbf{XY}](L_1(\mathbf{X}, \mathbf{Y})) = h_1'(L_1(\mathbf{X}, \mathbf{Y}))$. Thus, $(c, h_1'[\mathbf{XY}], \emptyset)$ is a trigger with respect to $h_1'(L_1(\mathbf{X}, \mathbf{Y}))$. This trigger is defined line 12 to generate $h_1''[\mathbf{XY}](L_2(\mathbf{X}, \mathbf{Z}))$ where $h_1''[\mathbf{XY}]$ is an extension of $h_1'[\mathbf{XY}]$ as described in the algorithm.

Based on these computations, the new row N is expected to be the triple $\langle h_1'[\mathbf{XY}](L_1(\mathbf{X}, \mathbf{Y})), h_1''[\mathbf{XY}](L_2(\mathbf{X}, \mathbf{Z})), i\rangle$. However, in order to avoid redundancies and more importantly, to ensure the termination of the construction, before inserting this row, we check line 14 whether the tableau contains a row whose atoms in columns Del and Query are respectively equal to $h_1'[\mathbf{XY}](L_1(\mathbf{X}, \mathbf{Y}))$ and $h_1''[\mathbf{XY}](L_2(\mathbf{X}, \mathbf{Z}))$, up to a renaming of the nulls.

Calling such rows *isomorphic rows*, this task is achieved by the function called *IsoRow* which takes as input the two atoms, scans the current state of the tableau and returns either the empty set or the row number where isomorphic atoms have been found. If the empty set is returned, the triple is inserted as the row N of T-DEL as stated in lines 15, 16 and 17; otherwise the row number i whose atom T-DEL$[i, \mathsf{Del}]$ has triggered the inverse constraint $\overline{c}$ is added in the column Ref of row ρ (line 21). The following proposition shows that Algorithm 3 always terminates.

PROPOSITION 3. *For every finite set of positive constraints $\mathcal{C}$ and every finite set of facts I, Algorithm 3 applied to $\mathcal{C}$ and I always terminates.*

PROOF. The proof is an immediate consequence of the fact that since $\mathcal{C}$ and I are finite, and since equal atoms up to a renaming of the nulls are not inserted, the potential triples to be added in T-DEL is finite. □

Algorithm 3 is the first step of our processing, which clearly does not require any access to the database instance. The second step consists in performing the deletions against the database instance, which is achieved by Algorithm 5. These two algorithms are thus the basic steps of Algorithm 4, in charge of the whole deletion process.

Notice that in the second step of the processing, access to D is performed through a function called *eval* that takes as input an atom φ possibly containing nulls and returns the set of all facts in D that are instantiations of φ. Algorithms 4, 5 and 6 are explained in the following example.

Algorithm 4: *PerfDel*$(\Delta, DReq)$

Input:
- $\Delta = (D, \mathcal{C})$ where $\mathcal{C}$ is a set of positive constraints.
- *DReq*, a set of facts requested to be deleted.

Output: The updated database Δ'

```
1: T-DEL := BuildTableau(C, DReq)
2: DSet := BuildDelSet(T-DEL, Δ, DReq)
3: D' := D \ DSet
4: return Δ' := (D', C)
```

Example 4. Let $\Delta = (D, \mathcal{C})$ be the database instance where D and $\mathcal{C}$ are shown in Figure 3. For $DReq = \{B(b, a),$

Algorithm 5: *BuildDelSet*(T-DEL, Δ, *DReq*)

Input:
- $\Delta = (D, \mathcal{C})$ where $\mathcal{C}$ is a set of positive constraints.
- *DReq*, a set of facts requested to be deleted.
- The tableau T-DEL obtained by the call *BuildTableau*($\mathcal{C}$, *DReq*) of Algorithm 3

Output: A set *DSet* of facts to be deleted.

```
1: DSet := ∅
2: for all l ∈ DReq do
3:     i := FindRow(T-DEL, l)
4:     AnsQ0 := eval(T-DEL[i, Del], Δ) \ DSet
5:     if AnsQ0 ≠ ∅ then
6:         for all φ0 ∈ AnsQ0 do
7:             DelSideEffects(i, φ0, DSet)
8: return DSet
```

Algorithm 6: *DelSideEffects*(*line*, φ, *DSet*)

Input:
- A *line* number of T-DEL.
- The fact φ for which side effects are computed.
- The current set *DSet*.

Output: The updated set *DSet*.

```
1: DSet := DSet ∪ {φ}
2: S := {pos | line occurs in T-DEL[pos, Ref]}
3: if S ≠ ∅ then
4:     for all j ∈ S do
5:         AnsQ1 := eval(T-DEL[j, Del], Δ) \ DSet
6:         if AnsQ1 ≠ ∅ then
7:             for all φ1 ∈ AnsQ1 do
8:                 Let h be a homomorphism such that
                   h(T-DEL[j, Del]) = φ1
9:                 if φ is an instance of h(T-DEL[j, Query])
                   then
10:                    AnsQ2 :=
                       eval(h(T-DEL[j, Query]), Δ) \ DSet
11:                    if AnsQ2 = ∅ then
12:                        DelSideEffects
                                   (j, h(T-DEL[j, Del]), DSet)
13: return DSet
```

$B(b,b)$, $B(c,b)\}$, the tableau T-DEL shown in Figure 3 is built line 1 of Algorithm 4 (explanations of the easy construction of T-DEL are omitted, due to lack of space). Then Algorithm 5 is called line 2 of Algorithm 4. *DSet* is set to $\emptyset$ (line 1) and the loop line 2 is run for each fact in *DReq*. For $B(b,c)$, $AnsQ_0$ is empty and nothing else happens. In the other two cases, we have $AnsQ_0 = \{B(b,b)\}$ and $AnsQ_0 = \{B(c,b)\}$. Thus, Algorithm 6 is first called with $DSet = \emptyset$, $line = 2$ and $\varphi = B(b,b)$.

After inserting $B(b,b)$ in *DSet* (line 1), Algorithm 6 computes rows in T-DEL whose column Ref corresponds to the deletion request being considered. This is so because in Algorithm 3 when a row j is built from T-DEL[*line*, Del], we have T-DEL[j, Ref] = *line*. Each row in S (Algorithm 6, line 2) is then considered. In our example, $S = \{4\}$ and the loop line 4 is executed once for $j = 4$ as follows.

Firstly, in line 5, we evaluate whether Δ contains instantiated facts corresponding to T-DEL[4, Del], which returns $AnsQ_1 = \{A(b,a)\}$. Then, for $\varphi_1 = A(b,a)$, the homomorphism h line 8 is defined by $h = \{N_1/a\}$.

$\mathcal{C}$	D		T-DEL		
			Del	Query	Ref
$A(x,y) \Rightarrow$	$A(b,a)$ $B(b,a)$	1	$B(b,c)$	–	–
$B(x,z)$	$A(c,a)$ $B(b,b)$	2	$B(b,b)$	–	–
	$A(c,c)$ $B(c,b)$	3	$B(c,b)$	–	–
		4	$A(b,N_1)$	$B(b,N_2)$	1-2
		5	$A(c,N_3)$	$B(c,N_4)$	3

Figure 3: Database and tableau of Example 4

When applying h on T-DEL[4, Query] line 10, we are simulating the instantiation of the constraint c used to obtain row 4: when *body*(c) is instantiated by φ_1 we obtain *head*(c) = h(T-DEL[4, Query]). Since Δ is consistent, if φ_1 is in D, D also contains at least one instance of h(T-DEL[4, Query]). In our example, as $h = \{N_1/a\}$, line 10 returns $AnsQ_2 = \{B(b,a)\}$ because $B(b,b)$ is in *DSet*.

Since the current values of φ and φ_1 are respectively $B(b,b)$ and $A(b,a)$, the test line 9 returns `true`. We notice here that when the returned value is `false`, the current values of φ and φ_1 do not correspond to the same homomorphism, in which case, the process stops.

We now emphasize that, since $AnsQ_2 \neq \emptyset$, D contains at least two atoms (namely $B(b,b)$ under processing and the atoms in $AnsQ_2$) that can be invoked for stating that $D \models \mathcal{C}$. Consequently, removing $B(b,b)$ does not invalidate that $D \models \mathcal{C}$ and so, deleting φ does not require deleting φ_1. Algorithm 6 implements these remarks as follows:

- If the test line 11 returns `false` no further deletion has to be processed.
- If the test line 11 returns `true`, then the deletion of φ_0 requires to also delete the corresponding instantiations of its head (that is h(T-DEL[j, Del])). This is precisely what is done through the recursive call line 12. This case will be illustrated next in our example.

The process continues with the loop line 6 of Algorithm 5, for which we now have $line = 2$, $\varphi_0 = B(c,b)$ and $DSet = \{B(b,b)\}$ as input for Algorithm 6. Thus, we obtain $DSet = \{B(b,b), B(c,b)\}$, $S = \{5\}$ and the execution of the loop line 4 for $j = 5$ implies that $AnsQ_1 = \{A(c,a), A(c,c)\}$.

For $\varphi_1 = A(c,a)$, $AnsQ_2 = \emptyset$ and the call *DelSideEffects*(5, $A(c,a)$, $\{B(b,b), B(c,b)\}$) line 12 results in adding $A(c,a)$ to *DSet* (line 1), and setting S to $\emptyset$. Similar computations are done for $\varphi_1 = A(c,c)$ and the call *DelSideEffects*(5, $A(c,c)$, $\{B(b,b), B(c,b), A(c,a)\}$). Hence, $DSet = \{B(b,b), B(c,b), A(c,a), A(c,c)\}$ is returned line 13 of Algorithm 6 and the processing returns $D' = \{A(b,a), B(b,a)\}$. □

As for insertions, deletions are performed in a deterministic way and require no further information other than the database instance Δ and the facts to be deleted. The following proposition shows that deletions are actually performed, satisfying the minimal change requirement.

PROPOSITION 4. *Let $\Delta = (D, \mathcal{C})$ be a database instance and DReq a finite set of atoms to be deleted. The following holds regarding the database instance $\Delta' = (D', \mathcal{C})$ returned by the call PerfDel(Δ, DReq) of Algorithm 4:*

1. *The call PerfDel(Δ, DReq) always terminates.*
2. *D' contains no facts of DReq and $D' \models \mathcal{C}$.*
3. *For every φ in DSet \ DReq, $D' \cup \{\varphi\} \not\models \mathcal{C}$.*

PROOF. 1. It is easy to see that in Algorithm 6 when $AnsQ_1$ is empty, then no further recursive call is processed. Moreover, as $DSet$ contains only facts in D, its cardinality is finite. Therefore only a finite number of recursive calls is possible, thus implying that Algorithm 4 always terminates.

2. Since $DReq \subseteq DSet$, $D' = D \setminus DSet$, $DReq \cap D' = \emptyset$.

Let us now assume that D' does not satisfy $c : L_1(\mathbf{X}, \mathbf{Y}) \Rightarrow L_2(\mathbf{X}, \mathbf{Z})$. This means that D' contains a fact of the form $L_1(\alpha, \beta)$ but no fact of the form $L_2(\alpha, \gamma)$. As D satisfies $\mathcal{C}$, $L_1(\alpha, \beta)$ and $D' \subseteq D$, $L_1(\alpha, \beta)$ is in D and thus D also contains a fact of the form $L_2(\alpha, \gamma)$. Therefore $L_1(\alpha, \beta)$ is not in $DSet$, whereas all facts of the form $L_2(\alpha, \gamma)$ are in $DSet$. This in particular means that all facts $L_2(\alpha, \gamma)$ are involved in the recursive calls of Algorithm 6. During the last call for these facts and for which the test line 9 succeeds, we have $AnsQ_2 = \emptyset$ and thus, a recursive call concerning $L_1(\alpha, \beta)$ is generated line 12, and the processing of this call adds $L_1(\alpha, \beta)$ in $DSet$. This is a contradiction because we assumed that $L_1(\alpha, \beta)$ was not in $DSet$.

3. Since φ is in $DSet \setminus DReq(\Delta)$, φ is a side effect of the deletion of a fact in $DReq(\Delta)$. Let $DelSideEffects(i_1, \varphi, DSet)$ be the call by which φ has been inserted in $DSet$. Denoting by ϕ the fact (in T-DEL$[i_0, \mathsf{Del}]$) being processed when this call occurred, we know that φ and ϕ are in D and that there is a row i_1 in T-DEL such that (i) i_0 occurs in T-DEL$[i_1, \mathsf{Ref}]$, (ii) φ is an instantiation of T-DEL$[i_1, \mathsf{Del}]$ and (iii) ϕ is an instantiation of T-DEL$[i_0, \mathsf{Del}]$ and of T-DEL$[i_1, \mathsf{Query}]$.

By Algorithm 3, there exists $c : L_1(\mathbf{X}, \mathbf{Y}) \Rightarrow L_2(\mathbf{X}, \mathbf{Z})$ in $\mathcal{C}$ such that T-DEL$[i_1, \mathsf{Del}]$ (respectively T-DEL$[i_1, \mathsf{Query}]$) is an instance of $L_1(\mathbf{X}, \mathbf{Y})$ (respectively of $L_2(\mathbf{X}, \mathbf{Z})$). Moreover, as the recursive call $DelSideEffects(i_1, \varphi, DSet)$ happens line 12 of Algorithm 6, the test line 11 of Algorithm 6 succeeds, that is, all instances of $L_2(\mathbf{X}, \mathbf{Z})$ in D matching with φ (*i.e.,* the facts in $eval(h(\text{T-DEL}[i_1, \mathsf{Query}]), \Delta)$, line 10 of Algorithm 6) are in $DSet$. Thus, writing φ as $L_1(\alpha, \beta)$, we have the following situation: $D' \cup \{\varphi\}$ contains $L_1(\alpha, \beta)$ but no fact of the form $L_2(\alpha, \gamma)$. Therefore, $D' \cup \{\varphi\}$ does not satisfy c and the proof is complete. □

4.3 Global Updates

We now generalize the previous two sub-sections in the following two respects: we consider (i) database instances in which *positive and negative constraints* are defined, and (ii) update requests containing *insertion and deletions*.

Assuming a database instance $\Delta = (D, \mathcal{C})$ where $\mathcal{C} = \mathcal{C}_P \cup \mathcal{C}_N$, and sets of facts $IReq$ and $DReq$ meant to be respectively inserted and deleted, updates are processed according to Algorithm 7 as follows. Line 1, the tableau T-INS for $IReq$ is built up, considering only constraints in $\mathcal{C}_P$. The test line 2 checks whether the insertions are possible. If not, the process stops, otherwise the following steps are run.

Consistency of the insertions with respect to negative constraints is checked line 3, through a call of Algorithm 8. If an inconsistency is found, the process stops. Otherwise the deletions necessary to restore consistency are the facts in the answers to the queries in *queries2Del*. These answers are added into $DReq$ line 6, and the following step is run.

The tableau T-DEL is computed line 7 and the set $DSet$ of all side effects is computed line 10. Line 11, we check whether these deletions are conflicting with the requested insertions (*i.e.,* no fact has to be inserted *and* deleted at the same time). If no conflict is detected the database instance is modified line 12 and returned line 13. Otherwise no modification is done and in both cases, the process stops.

Algorithm 7: $GlobalUpd(\Delta, IReq, DReq)$

Input:
- $\Delta = (D, \mathcal{C})$ with $\mathcal{C} = \mathcal{C}_P \cup \mathcal{C}_N$.
- $IReq$ (respectively $DReq$), a set of atoms requested to be inserted (respectively deleted).

Output: The updated database Δ' (possibly equal to Δ).

1: $BuildInsTab(IReq, \mathcal{C}_P, \text{T-INS})$
2: **if** $PossibleIns(\text{T-INS})$ **then**
3: $\quad ConsCheck(\text{T-INS}, queries2Del, Consistent)$
4: $\quad$ **if** $Consistent$ **then**
5: $\quad\quad$ **for all** $Q \in queries2Del$ **do**
6: $\quad\quad\quad DReq := DReq \cup eval(Q, \Delta)$
7: $\quad\quad \text{T-DEL} := BuildDelTab(\mathcal{C}_P, DReq)$
8: $\quad\quad D' := D \cup \text{C-INS}(IReq)$
9: $\quad\quad$ // The instance $(D', \mathcal{C}_P)$ is denoted by Δ_{Ins}
10: $\quad\quad DSet := BuildDelSet(\text{T-DEL}, \Delta_{Ins}, DReq)$
11: $\quad\quad$ **if** $\text{C-INS}(IReq) \cap DSet = \emptyset$ **then**
12: $\quad\quad\quad D' := D' \setminus DSet$
13: $\quad\quad\quad$ **return** $\Delta' = (D', \mathcal{C})$
14: $\quad\quad$ **else**
15: $\quad\quad\quad$ **return** $\Delta' = (D, \mathcal{C})$
16: $\quad\quad\quad$ **output**('Insertions and deletions conflict')
17: $\quad$ **else**
18: $\quad\quad$ **return** $\Delta' = (D, \mathcal{C})$
19: $\quad\quad$ **output**('Insertions violate negative constraints')
20: **else**
21: $\quad$ **return** $\Delta' = (D, \mathcal{C})$
22: $\quad$ **output**('Insertions are not possible')

Regarding now Algorithm 8, every fact φ to be inserted is checked against the constraints in $\mathcal{C}_N$ as follows:

- If φ is an instantiation of $L(\mathbf{X})$ (respectively of $L_1(\mathbf{X}_1)$ *and* $L_2(\mathbf{X}_2)$) in a constraint with a single atom (respectively with two atoms), and if the instantiated comparison formula evaluates to `true`, then the consistency cannot be restored. Thus $Consistent$ is set to `false` (lines 6 and 11).
- Summarizing the two cases lines 13 and 16, when φ is an instance of one of the two literals in c, then an inconsistency appears if the database instance contains an instance of the other literal and if the variables are valued so as the comparisons in c evaluate to `true`. This is why, in lines 14 and 17, we keep track of the queries that allow to retrieve all the instances that cause the inconsistency. Notice that these queries are selection queries where the selection condition is set by $v(comp(\mathbf{X}'))$. The answer to such query contains all instances of $v(L_2(\mathbf{X}_2))$ (line 13) or $v(L_1(\mathbf{X}_1))$ (line 16) in D that satisfy $v(comp(\mathbf{X}'))$.

Example 5. Let $\Delta = (D, \mathcal{C})$ be the database instance of our running example where D is shown Figure 2 and where $\mathcal{C}$ is the set of constraints $c_1 - c_8$ listed earlier.

Consider first $IReq = \{\mathsf{Cites}(coolP, coolP)\}$ and $DReq = \emptyset$. Due to line 1 of Algorithm 7, Algorithm 1 returns T-INS where C-INS$(IReq) = \{\mathsf{Cites}(coolP, coolP), \mathsf{Paper}(coolP)\}$ and N-INS$(IReq) = \{\mathsf{PubIn}(coolP, N_1), \mathsf{Cites}(coolP, N_2)\}$. $\mathsf{PubIn}(coolP, Cool\_J)$ and $\mathsf{Cites}(coolP, rdfP)\}$ being in D, the insertion is possible.

Then, line 3 of Algorithm 7, Algorithm 8 is called with

Algorithm 8: *ConsCheck*(I, *queries2Del*, *Consistent*)

Input: I, a set of facts requested to be inserted.
Output:
- The value of the boolean variable *Consistent*.
- The set *queries2Del* of queries allowing to retrieve the atoms to be deleted if *Consistent* is `true`.

1: *queries2Del* := $\emptyset$
2: *Consistent* := `true`
3: **for all** $c : (comp(\mathbf{X}') \wedge L(\mathbf{X})) \Rightarrow \bot$ in $\mathcal{C}_N$ **do**
4: **if** *Consistent* **then**
5: **if** there exists a valuation v of $\mathbf{X}$ such that $v(\mathbf{X}')$ evaluates to `true` and $v(L(\mathbf{X}))$ is in I **then**
6: *Consistent* := `false`
7: **if** *Consistent* **then**
8: **for all** $c : (comp(\mathbf{X}') \wedge L_1(\mathbf{X}_1) \wedge L_2(\mathbf{X}_2)) \Rightarrow \bot$ in $\mathcal{C}_N$ **do**
9: **if** *Consistent* **then**
10: **if** there exists a valuation v of $\mathbf{X}$ such that $v(\mathbf{X}')$ evaluates to `true` and $v(L_1(\mathbf{X}_1))$ and $v(L_2(\mathbf{X}_2))$ are in I **then**
11: *Consistent* := `false`
12: **else**
13: **if** there exists a valuation v_1 of $\mathbf{X}_1$ such that $v(L_1(\mathbf{X}_1)) \in I$ **then**
14: *queries2Del* := *queries2Del* $\cup \{v_1(comp(\mathbf{X}') \wedge L_2(\mathbf{X}_2))\}$
15: **else**
16: **if** there exists a valuation v_2 of $\mathbf{X}_2$ such that $v(L_2(\mathbf{X}_2)) \in I$ **then**
17: *queries2Del* := *queries2Del* $\cup \{v_2(comp(\mathbf{X}') \wedge L_1(\mathbf{X}_1))\}$
18: **return** *Consistent* ; *queries2Del*

I = C-INS(*IReq*). In the loop line 3 of Algorithm 8, the test line 5 succeeds due to c_8 and $\mathsf{Cites}(coolP, coolP)$ in I. Thus, line 6 of Algorithm 8, *Consistent* is set to `false`, and Algorithm 7 returns $\Delta' = \Delta$ and outputs 'Insertions violate negative constraints'.

To model the deletion of $\mathsf{PubIn}(coolP, Cool\_J)$ in [6], due to the fact that *coolP* is actually published in $KAIS$, consider the global update defined by $IReq = \{\mathsf{PubIn}(coolP, KAIS)\}$ and $DReq = \{\mathsf{PubIn}(coolP, Cool\_J)\}$.

According to line 1 of Algorithm 7, T-INS as computed by Algorithm 1 is defined by C-INS(*IReq*) = $\{\mathsf{PubIn}(cool, KAIS), \mathsf{Paper}(coolP), \mathsf{Journal}(KAIS)\}$ and N-INS(*IReq*) = $\{\mathsf{PubIn}(coolP, N_1), \mathsf{Cites}(coolP, N_2)\}$. The insertion is possible with respect to positive constraints because $\mathsf{PubIn}(coolP, Cool\_J)$ and $\mathsf{Cites}(coolP, rdfP)\}$ are in D. Therefore the test line 2 of Algorithm 7 succeeds, implying that Algorithm 8 is called line 3 of Algorithm 7 with C-INS(*IReq*) to check validity with respect to negative constraints.

In this case, in Algorithm 8, the tests lines 5 and 10 both fail, but the test line 13 succeeds for $c_6 : (y \neq z) \wedge \mathsf{PubIn}(x, y) \wedge \mathsf{PubIn}(x, z) \Rightarrow \bot)$ and $v = \{x/coolP, y/KAIS\}$. Consequently, line 14 of Algorithm 8, the query Q defined by $(KAIS \neq z) \wedge \mathsf{PubIn}(coolP, z)$ is inserted in *queries2Del*. Algorithm 8 stops returning this set *queries2Del* along with the value `true` for *Consistent*. Since the test line 4 of Algorithm 7 succeeds, the answer to Q is evaluated line 6 of Algorithm 7, returning $\{\mathsf{PubIn}(coolP, Cool\_J)\}$. As *DReq* contains this fact, the call line 7 of Algorithm 7 with this fact as input computes the tableau T-DEL shown in Figure 4.

	Del	Query	Ref
1	$\mathsf{PubIn}(coolP, Cool\_J)$	–	–
2	$\mathsf{Paper}(coolP)$	$\mathsf{PubIn}(coolP, N_1)$	1-3
3	$\mathsf{PubIn}(coolP, N_2)$	$\mathsf{Paper}(coolP)$	2-6
4	$\mathsf{Cites}(coolP, N_3)$	$\mathsf{Paper}(coolP)$	2-6
5	$\mathsf{Cites}(N_4, coolP)$	$\mathsf{Paper}(coolP)$	2-6
6	$\mathsf{Paper}(coolP)$	$\mathsf{Cites}(coolP, N_5)$	4
7	$\mathsf{Paper}(N_4)$	$\mathsf{Cites}(N_4, N_6)$	5-9-10
8	$\mathsf{PubIn}(N_4, N_7)$	$\mathsf{Paper}(N_4)$	7-11
9	$\mathsf{Cites}(N_4, N_8)$	$\mathsf{Paper}(N_4)$	7-11
10	$\mathsf{Cites}(N_9, N_4)$	$\mathsf{Paper}(N_4)$	7-11
11	$\mathsf{Paper}(N_4)$	$\mathsf{PubIn}(N_4, N_{10})$	8

Figure 4: The tableau T-DEL in Example 5

The computation of the set *DSet* is processed line 10 of Algorithm 7. Since by line 8 of Algorithm 7, $\mathsf{PubIn}(coolP, KAIS)$ is in the instance Δ_{Ins}, in the call of *DelSideEffects*, the test line 11 in Algorithm 6 fails, implying that no recursive call occurs. Hence, the test line 11 of Algorithm 7 succeeds, and by lines 12 and 13 of Algorithm 7, we obtain:

$D' = (D \cup \{\mathsf{PubIn}(coolP, KAIS)\}) \setminus \{\mathsf{PubIn}(coolP, Cool\_J)\}$

showing that the updates are performed as expected, replacing $\mathsf{PubIn}(coolP, KAIS)$ with $\mathsf{PubIn}(coolP, Cool\_J)\}$. □

The following proposition states that global update processing satisfies the requirements stated earlier.

PROPOSITION 5. *Let $\Delta = (D, \mathcal{C})$ be a database instance. For all finite sets IReq and DReq the following holds:*

1. *The call GlobalUpd(Δ, IReq, DReq) of Algorithm 7 always terminates.*
2. *If the call GlobalUpd(Δ, IReq, DReq) of Algorithm 7 returns no inconsistency then:*
   (a) $IReq \subseteq D'$, $DReq \cap D' = \emptyset$ and $D' \models \mathcal{C}$.
   (b) For every φ in C-INS($IReq$) $\setminus$ $IReq$, $D' \setminus \{\varphi\} \not\models \mathcal{C}$.
   (c) For every φ in $DSet \setminus DReq$, $D' \cup \{\varphi\} \not\models \mathcal{C}$.

PROOF. 1. This result holds because, by Proposition 1, Proposition 3 and Proposition 4.1, insertion and deletion processing terminates and because Algorithm 8 terminates.

2(a). By Proposition 2.1 and Proposition 4.2, we have $IReq \subseteq D'$, $DReq \cap D' = \emptyset$ and $D' \models \mathcal{C}_P$. We thus have to prove that $D' \models \mathcal{C}_N$. This is so because when the boolean variable *Consistent* of Algorithm 8 is `true`, deleting the facts computed in the loop line 5 of Algorithm 7 restores consistency with respect to negative constraints. Therefore, $D' \models \mathcal{C}$.

2(b). By Proposition 2.2, we know that if φ is in C-INS($IReq$)$\setminus$ $IReq$, then $D' \setminus \{\varphi\} \not\models \mathcal{C}_P$. Thus, we have that $D' \setminus \{\varphi\} \not\models \mathcal{C}$.

2(c). Assume that φ does not belong to the answer to a query Q in *query2Del*. By Proposition 4.2, since φ is in $DSet \setminus DReq$, $D' \cup \{\varphi\} \not\models \mathcal{C}_P$, and so, $D' \cup \{\varphi\} \not\models \mathcal{C}$. Assume now that φ belongs to the answer to a query Q in *query2Del*. It can be seen that the negative constraint involved when inserting Q into *query2Del* is not satisfied, or in other words that $D' \cup \{\varphi\} \not\models \mathcal{C}_N$. Therefore, we have $D' \cup \{\varphi\} \not\models \mathcal{C}$. □

4.4 Complexity and Implementation Issues

We know from [18] that checking whether all previously satisfied TGDs are still satisfied after an update is an NP-complete problem in the general case (and co-NP-complete

for restricted forms of TGDs). Due to the difficulty of the problem, authors in [6] propose a special-purpose algorithm, working uniquely with RDF/S constraints, as a polynomial version of their general-purpose version where this restriction is not envisaged. Even if the work in [6] is close to ours, their polynomial version is *not* designed to deal with application *and* RDF/S constraints as we do in our work. In what follows, we show that, although our approach is more general than [6], its time complexity is polynomial with respect to the sizes of D in the database instance and the sets $IReq$ and $DReq$ defining the update.

To this end, we denote by α the maximal arity of predicates in PRED and $|E|$ denotes the cardinality of a set E. Given a set of facts I and a set of positive constraints $\mathcal{C}$, the number of constants occurring in I is bounded by $\alpha \times |I|$, and so, the number of possible instantiations of an atom $L(\mathbf{X})$ is bounded by $(\alpha \times |I|)^\alpha$, that is $O(|I|^\alpha)$. On the other hand, computing all triggers of a constraint c in I is $O(|I|)$.

Regarding the complexity of insertion, the set C-INS($IReq$) contains at most all possible instances of the heads of the constraints, which is $O(|\mathcal{C}| \times |IReq|^\alpha)$ or $O(|IReq|^\alpha)$. Since inserting a fact in C-INS($IReq$) requires to scan the tableau, the computation of C-INS($IReq$), that is the computation of T-INS, in Algorithm 1 is $O(|IReq|^{2.\alpha})$.

Now, let $N_1, \ldots, N_k$ be the labelled nulls N-INS($IReq$). Valuating these nulls in $D \cup$ C-INS($IReq$) amounts to test whether the query $\langle (N_1, \ldots, N_k) \mid \Gamma \rangle$, where Γ is the conjunction of all atoms in N-INS($IReq$), has a nonempty answer. Since this is linear, the complexity of Algorithm 2 is $O(|D| + |\text{C-INS}(IReq)|)$, and thus, the overall complexity of insertion is $O(|D| + |IReq|^{2.\alpha})$.

Regarding deletions, similar arguments as those above for the construction of T-INS show that the complexity of Algorithm 3 (*i.e.,* the construction of T-DEL) is $O(|DReq|^{2.\alpha})$. On the other hand, there cannot be more recursive calls of Algorithm 6 than there are facts in $D \cup DReq$ because for each of these calls, a new fact from $D \cup DReq$ is added in the set $DSet$. Each of these calls consists in a scan of T-DEL along with the execution of linear queries against a set of at most $|D|$ facts. Hence the complexity of all recursive calls is $O((|D| + |DReq|^{2.\alpha}) \times (|D| + |DReq|))$ and thus, the complexity of deletion is $O(|D|^2 + |DReq|^{3.\alpha})$.

Regarding global updates, the complexity of Algorithm 7 is the sum of the complexities of insertion, deletion and of Algorithm 8. It is easy to see that the latter complexity is linear in the size of C-INS($IReq$), and thus is $O(|IReq|^\alpha)$. Therefore, the complexity of global updates is $O((|D| + |IReq|^{2.\alpha}) + (|D|^2 + |DReq|^{3.\alpha}) + |IReq|^\alpha)$, or more simply $O(|D|^2 + |IReq|^{2.\alpha} + |DReq|^{3.\alpha})$.

Substantial optimization is possible regarding the construction of T-INS and T-DEL, because these tableaux can be obtained from instantiations of *generic* tableaux, built up for every predicate *once for all*, during the design phase. This optimization is outlined in the following example.

Example 6. In the context of Example 4 where the only constraint is $c : A(x, y) \Rightarrow B(x, z)$, the generic tableaux shown in Figure 5 are built using Algorithm 1 and Algorithm 3.

For the deletion defined by $DReq = \{B(b, a),\ B(b, b),\ B(c, b)\}$, the tableau T-DEL of Figure 3 can be obtained by instantiating (α, β) in the generic tableau T-DEL^B successively with (b, a), (b, b) and (c, b). □

T-INS^A

$A(\alpha, \beta)$
$B(\alpha, N_1)$

T-INS^B

$B(\alpha, \beta)$

T-DEL^A

	Del	Query	Ref
1	$A(\alpha, \beta)$	–	–

T-DEL^B

	Del	Query	Ref
1	$B(\alpha, \beta)$	–	–
2	$A(\alpha, N_1)$	$B(\alpha, N_2)$	1

Figure 5: Generic tableaux for Example 6

5. RDF/S SEMANTICS AS CONSTRAINTS

Additionally to application constraints, we now take into account the RDF/S constraints recalled in a simplified and compacted form in Figure 6. As in [6], we use the predicates Cl and $CSub$ for classes and sub-classes, Pr and $PSub$ for properties and sub-properties, Dom and Rng for property domain and range, CI and PI for class and property instances, and Ind and URI for individuals and URIs. We call *schema* predicates (respectively *instance* predicate) any symbol from the set SCHPRED = $\{Cl, Pr, CSub, PSub, Dom, Rng\}$ (respectively INSTPRED = $\{CI, PI, Ind, URI\}$).

Based on these two kinds of predicates, the constraints listed in [6] are split into three kinds, as shown in Figure 6:

1. *Typing constraints* involving predicates in SCHPRED or in INSTPRED.
2. *Schema constraints* involving predicates in SCHPRED and stating properties of the schema predicates.
3. *Instance constraints* written $(\forall \mathbf{X})(\varphi_{Sch}(\mathbf{X}') \Rightarrow \varphi_{Inst}(\mathbf{X}))$ where variables in $\mathbf{X}'$ are in $\mathbf{X}$, $\varphi_{Sch}(\mathbf{X}')$ is an atom over a predicate in SCHPRED, and $\varphi_{Inst}(\mathbf{X})$ is a positive constraint involving instance predicates.

Moreover, we consider a fourth kind of constraints, called *application constraints*. These constraints defined for a given application involve only predicates in INSTPRED and follow the format given in Definition 1.

In this setting, an RDF/S database $\Delta = (\Delta_{Sch}, \Delta_{Inst})$ is a pair where $\Delta_{Sch} = (D_{Sch}, \mathcal{C}_{Sch})$ and $\Delta_{Inst} = (D_{Inst}, \mathcal{C}_{Inst})$, called the schema and the instance of Δ, are such that:

- $\mathcal{C}_{Sch}$ is the set of schema constraints and D_{Sch} is a set of facts over predicates in SCHPRED satisfying all constraints in $\mathcal{C}_{Sch}$.
- $\mathcal{C}_{Inst}$ is the set of instance constraints or application constraints, and D_{Inst} is a set of facts over predicates in INSTPRED such that $D_{Sch} \cup D_{Inst}$ satisfies all constraints in $\mathcal{C}_{Inst}$.
- All typing constraints are satisfied by $D_{Sch} \cup D_{Inst}$.

In order to consider only instance constraints that fit Definition 1, the four RDF/S instance constraints of Figure 6 are replaced with their partial instantiations over schema predicates. More formally, given $(\forall \mathbf{X})(\varphi_{Sch}(\mathbf{X}') \Rightarrow \varphi_{Inst}(\mathbf{X}))$, the set of all these instantiations is the set of all $h(\varphi_{Inst}(\mathbf{X}''))$ where variables in $\mathbf{X}''$ belong to $\mathbf{X}$ but not to $\mathbf{X}'$, and h is a homomorphism of $\mathbf{X}'$ such that $D_{Sch} \models h(\varphi_{Sch}(\mathbf{X}'))$.

Example 7. Referring to our running example, let $\Delta = (\Delta_{Sch}, \Delta_{Inst})$ be the integrated database of Section 2. As for Δ_{Sch}, Figure 1 shows that D_{Sch} contains Cl(Journal), Cl(Paper), Pr(PubIn), Pr(Cites), Dom(PubIn, Paper), Rng(PubIn, Journal), Dom(Cites, Paper), Rng(Cites, Paper). It can be seen that $D_{Sch} \models \mathcal{C}_{Sch}$ holds.

• **Typing Constraints:**
$(\forall x)(Cl(x) \Rightarrow URI(x))$ // $(\forall x)(Pr(x) \Rightarrow URI(x))$ // $(\forall x)(Ind(x) \Rightarrow URI(x))$ // $(\forall x,y)(CSub(x,y) \Rightarrow Cl(x) \wedge Cl(y))$
$(\forall x,y)(PSub(x,y) \Rightarrow Pr(x) \wedge Pr(y))$ // $(\forall x,y)(Dom(x,y) \Rightarrow Pr(x) \wedge Cl(y))$ // $(\forall x,y)(Rng(x,y) \Rightarrow Pr(x) \wedge Cl(y))$
$(\forall x,y)(CI(x,y) \Rightarrow Ind(x) \wedge Ind(y))$ // $(\forall x,y,z)(PI(x,y,z) \Rightarrow Ind(x) \wedge Ind(y) \wedge Pr(z))$
$(\forall x)((Cl(x) \wedge Pr(x)) \Rightarrow \bot)$ // $(\forall x)((Cl(x) \wedge Ind(x)) \Rightarrow \bot)$ // $(\forall x)((Pr(x) \wedge Ind(x)) \Rightarrow \bot)$
$(\forall x)((Cl(x) \Rightarrow CSub(x, rdfs{:}Resource)$ // $(\forall x)((Ind(x) \Rightarrow CI(x, rdfs{:}Resource)$
• **Schema Constraints:**
$(\forall x)(Pr(x) \Rightarrow (\exists y,z)(Dom(x,y) \wedge Rng(x,y)))$ // $(\forall x,y,z)(((y \neq z) \wedge Dom(x,y) \wedge Dom(x,z)) \Rightarrow \bot)$
$(\forall x,y,z)(((y \neq z) \wedge Rng(x,y) \wedge Rng(x,z)) \Rightarrow \bot)$
$(\forall x,y,z)(CSub(x,y) \wedge CSub(y,z) \Rightarrow CSub(x,z))$ // $(\forall x,y)(CSub(x,y) \wedge CSub(y,x) \Rightarrow \bot)$
$(\forall x,y,z)(PSub(x,y) \wedge PSub(y,z) \Rightarrow PSub(x,z))$ // $(\forall x,y)(PSub(x,y) \wedge PSub(y,x) \Rightarrow \bot)$
$(\forall x,y,z,w)(Psub(x,y) \wedge Dom(x,z) \wedge Dom(x,w) \wedge (z \neq w) \Rightarrow CSub(z,w)$
$(\forall x,y,z,w)(Psub(x,y) \wedge Rng(x,z) \wedge Rng(x,w) \wedge (z \neq w) \Rightarrow CSub(z,w)$
• **Instance Constraints:**
$(\forall x,y,z,w)(Dom(z,w) \Rightarrow (PI(x,y,z) \Rightarrow CI(x,w)))$ // $(\forall x,y,z,w)(Rng(z,w) \Rightarrow (PI(x,y,z) \Rightarrow CI(x,w)))$
$(\forall x,y,z)(CSub(y,z) \Rightarrow (CI(x,y) \Rightarrow CI(x,z)))$ // $(\forall x,y,z,w)(PSub(z,w) \Rightarrow (PI(x,y,z) \Rightarrow PI(x,y,w)))$

Figure 6: Simplified and compacted form of RDF/S constraints

If additionally DB_J is defined as a subclass of Journal, then $CSub$(DB_J, Paper) must occur in D_{Sch}. In this case, the constraint $(\forall x,y)(CSub(x,y) \Rightarrow Cl(x))$ in Figure 6, implies that Cl(DB_J) must also be in D_{Sch}.

Regarding now Δ_{Inst}, D_{Inst} contains the facts of Figure 2, expressed in their RDF/S form (as $CI(dbP$, Paper) or $PI(dbP, KAIS$, PubIn)) and the facts representing individuals and URIs (as $Ind(dbP)$ or $Ind(KAIS)$ or URI(Cites)). Moreover, given the domains and ranges of PubIn and Cites, the partial instantiations of the constraints in $\mathcal{C}_{Inst}$ are:

- $(\forall x,y)(PI(x,y,$PubIn$) \Rightarrow CI(x,$Paper$))$
- $(\forall x,y)(PI(x,y,$PubIn$) \Rightarrow CI(y,$Journal$))$
- $(\forall x,y)(PI(x,y,$Cites$) \Rightarrow CI(x,$Paper$))$
- $(\forall x,y)(PI(x,y,$Cites$) \Rightarrow CI(y,$Paper$))$.

One should notice that these constraints express $c_1 - c_4$ of our running example while $c_5 - c_8$ are seen as application constraints. For example, c_5 and c_8 are respectively written: $(\forall x,y)$ $(CI(x,$Paper$) \Rightarrow (\exists y)(PI(x,y,$ PubIn$)))$ and $(\forall x,y)(((x=y) \wedge PI(x,y,$Cites$)) \Rightarrow \bot)$.

In this setting, the second global update in Example 5 is defined by $IReq = \{PI(coolP, KAIS,$ PubIn$)\}$ and $DReq = \{PI(coolP, cool\_J,$ PubIn$)\}$, and is processed as described earlier in that example. □

6. RELATED WORK

Comparing our work with existing related approaches, we first note that constraint satisfaction in our approach is explicitly enforced in the database instance, as done in [6], and contrary to many other work on web semantics [12 14 17]. For instance, considering c_5 : Paper$(x) \Rightarrow$ PubIn(x,y), in our approach, $\Delta_0 = (D_0, \{c_5\})$ with $D_0 = \{$Paper$(coolP)$, PubIn$(coolP, KAIS)\}$ is consistent while $\Delta_1 = (D_1, \{c_5\})$ with $D_1 = \{$Paper$(coolP)\}$ is not. On the other hand, in [12 14 17] both Δ_0 and Δ_1 are considered consistent.

We consider that updates are performed in response to a change in the real world. In doing so, our work adopts the standard update perspective rather than the belief revision perspective of [10].

Contrary to us, in [6] all non contradictory insertions are possible thanks to a total ordering, which potentially imposes arbitrary choices. In our approach, to ensure determinism on the basis of updates and the database instance only, insertions for which information with respect to constraints is missing are rejected. It also important to notice that the time complexity of updates in our approach is polynomial, as in [6] when updates are restricted to deal with instance predicates. However, we recall that, we consider a more general setting than in [6] because, additionally to RDF/S constraints, we allow application constraints.

Our update strategy is also different than other proposals dealing with RDF/S changes such as [1 3 8 13]; some of them discussing non determinism, and all of them adopting the ontology constraint perspective ([7]), whereby constraints are seen as inference rules as discussed earlier.

In [1], an update semantics classification is proposed, according to which our approach falls in the category called Sem_2^{mat}, where the deletion/insertion of a fact f imposes the elimination/addition of all its causes/consequences. The approach in [3] is more generic than ours because schema updates are allowed. However, contrary to our approach, constraints in [3] are restricted to RDF/S positive constraints and updates are restricted to involve only one atom.

Although the work in [8] also addresses the issue of schema updating, this work is more restrictive than ours, since it is restricted to updates taking into account six of the positive RDF/S constraints considered in our approach. An update language is proposed in [13] allowing the definition of application-driven triggers to maintain ontology consistency. Although this language allows to specify update side effects, the problem is that termination when several updates are involved is handled 'manually', which is not realistic in practice.

Positive constraints extend update rules in [9], by allowing existential variables in the heads of the rules. However in [9], update rules also involve negative atoms to model deletion side effects in a *three valued* logic framework. In the present work, we consider the standard first order logic context in which *negative atoms are not stored*. Constraints that explicitly enforce deletions are negative constraints, which can be seen as a particular case of update rules.

We notice that considering tuple generating constraints increases expressivity at the cost of difficulties in the computation of the semantics. These difficulties have been addressed through a chase procedure (see [15] for a survey). Many work (such as [4 5 18]) have considered the use of the chase procedure in data exchange, *i.e.,* to perform source-target transformation, according to a set of mapping dependencies. In our approach, we consider updates instead

of data exchange. In this context, restricted chasing procedures are used to generate tableaux defining update patterns that are then instantiated in the database instance. We emphasize that our approach has a polynomial time complexity, contrary to chasing algorithms, including those presented in [18] where updates are considered.

7. CONCLUSIONS AND PERSPECTIVES

Dealing with RDF/S databases, this paper proposes a *polynomial* time *deterministic* update strategy ensuring database *consistency* with respect to a set of constraints and satisfying *minimal change* requirements. Updates are *sets* of insertions and deletions treated by a two-step update procedure where the first step is performed *independently* of the database instance. By allowing for application constraints, additionally to RDF/S constraints, our approach generalizes the standard key-foreign key constraints, and thus offers a broad update framework covering most practical cases.

Different research directions are being considered based on this work. The next step concerns an implementation where null values (blank nodes) are also accepted in the database instance. Our update method can then be assessed according to two different scenarios (with and without nulls). In dealing only with instance updates, our approach provides a useful tool capable of processing update requests issued by 'ordinary' users. Extending our strategy to deal with both schema and instance is a relevant but challenging research perspective, which necessitates to consider non linear TGDs. We finally intend to apply our method to *temporal* RDF/S database instances on the basis of our previous work in [11].

8. REFERENCES

[1] A. Ahmeti, D. Calvanese, and A. Polleres. Updating RDFS ABoxes and TBoxes in SPARQL. *CoRR*, abs/1403.7248, 2014.

[2] A. Calì, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. *J. Web Sem.*, 14:57–83, 2012.

[3] R. Chirkova and G. H. L. Fletcher. Towards well-behaved schema evolution. In *12th International Workshop on the Web and Databases, WebDB 2009, USA, June 28*, 2009.

[4] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, pages 207–224, 2003.

[5] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.

[6] G. Flouris, G. Konstantinidis, G. Antoniou, and V. Christophides. Formal foundations for RDF/S KB evolution. *Knowl. Inf. Syst.*, 35(1):153–191, 2013.

[7] G. Gottlob, G. Orsi, and A. Pieris. Ontological queries: Rewriting and optimization. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 2–13, 2011.

[8] C. Gutierrez, C. A. Hurtado, and A. A. Vaisman. RDFS update: From theory to practice. In *The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Greece, May 29 - June 2, 2011, Proceedings, Part II*, pages 93–107, 2011.

[9] M. Halfeld Ferrari Alves, D. Laurent, and N. Spyratos. Update rules in datalog programs. *J. Log. Comput.*, 8(6):745–775, 1998.

[10] H. Katsuno and A. O. Mendelzon. On the difference between updating a knowledge base and revising it. In *Proc. of the 2nd Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25.*, pages 387–394, 1991.

[11] D. Laurent. On monotonic deductive database updating under the open world assumption. In *Information Search, Integration, and Personalization - International Workshop, ISIP 2015. Revised Selected Papers*, volume 622 of *Communications in Computer and Information Science*, page 3–22. Springer, 2016.

[12] G. Lausen, M. Meier, and M. Schmidt. Sparqling constraints for RDF. In *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings*, pages 499–509, 2008.

[13] U. Lösch, S. Rudolph, D. Vrandecic, and R. Studer. Tempus fugit. In *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Crete, Greece, May 31-June 4, 2009, Proceedings*, pages 278–292, 2009.

[14] B. Motik, I. Horrocks, and U. Sattler. Bridging the gap between OWL and relational databases. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 807–816, 2007.

[15] A. Onet. The chase procedure and its applications in data exchange. In *Data Exchange, Integration, and Streams*, pages 1–37. 2013.

[16] M. T. Özsu. A survey of RDF data management systems. *Frontiers of Computer Science*, 10(3):418–432, 2016.

[17] P. F. Patel-Schneider. Using description logics for RDF constraint checking and closed-world recognition. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 247–253, 2015.

[18] R. Pichler and S. Skritek. The complexity of evaluating tuple generating dependencies. In T. Milo, editor, *Database Theory - ICDT 2011, 14th International Conference, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 244–255. ACM, 2011.

[19] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF querying with SPARQL on spark. *PVLDB*, 9(10):804–815, 2016.