

# Efficient Query Answering against Dynamic RDF Databases

François Goasdoué, Ioana Manolescu, Alexandra Roatis

▶ **To cite this version:**

François Goasdoué, Ioana Manolescu, Alexandra Roatis. Efficient Query Answering against Dynamic RDF Databases. EDBT - 16th International Conference on Extending Database Technology, Mar 2013, Genoa, Italy. 10.1145/2452376.2452412 . hal-00804503v1

**HAL Id: hal-00804503**

**https://inria.hal.science/hal-00804503v1**

Submitted on 25 Mar 2013 (v1), last revised 1 Oct 2013 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Efficient Query Answering against Dynamic RDF Databases

François Goasdoué Ioana Manolescu Alexandra Roatiş

*Inria Saclay and Université Paris-Sud, Bât. 650, Université Paris-Sud, 91405 Orsay Cedex, France*

fg@lri.fr, ioana.manolescu@inria.fr, alexandra.roatis@inria.fr

## ABSTRACT

A promising method for efficiently querying RDF data consists of translating SPARQL queries into efficient RDBMS-style operations. However, answering SPARQL queries requires handling *RDF reasoning*, which must be implemented outside the relational engines that do not support it.

We introduce the *database (DB) fragment of RDF*, going beyond the expressive power of previously studied RDF fragments. We devise novel *sound and complete techniques* for answering *Basic Graph Pattern (BGP)* queries within the DB fragment of RDF, exploring the two established approaches for handling RDF semantics, namely reformulation and saturation. In particular, we focus on handling database *updates* within each approach and propose a method for incrementally maintaining the saturation; updates raise specific difficulties due to the rich RDF semantics. Our techniques are designed to be deployed on top of any RDBMS(-style) engine, and we experimentally study their performance trade-offs.

## Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

## General Terms

Algorithms, Experimentation, Theory

## Keywords

RDF fragments, query answering, reasoning

## 1. INTRODUCTION

The Resource Description Framework (RDF) [1] is a graph-based data model accepted as the W3C standard for Semantic Web applications. As such, it comes with an ontology language, *RDF Schema* (RDFS), that can be used to enhance the description of RDF *graphs*, i.e., RDF datasets. The W3C standard for querying RDF is SPARQL Protocol and RDF Query Language (SPARQL) [2].

The literature provides several scalable solutions for querying RDF graphs using relational data management systems (RDBMSs, in short) or RDBMS-style specialized engines [3, 4, 5]. These works, however, ignore the essential RDF feature called *entailment*, which allows modeling *implicit information* within RDF. Taking entailment into account is crucial for answering SPARQL queries, as ignoring implicit information leads to incomplete answers [2]. Thus, to capitalize on (and benefit from) scalable RDBMS performance, SPARQL query answering can be split into a *reasoning* step which handles entailment outside the RDBMSs, and a *query evaluation* step delegated to RDBMSs.

**Saturation and reformulation** A popular reasoning step is *graph saturation* (*closure*). This consists of pre-computing (making explicit) and adding to the RDF graph all implicit information. Answering queries using saturation amounts to evaluating the queries against the saturated graph. While saturation leads to efficient query processing, it requires time to be computed, space to be stored, and must be recomputed upon updates. The alternative reasoning step is *query reformulation*. This consists in turning the query into a *reformulated query*, which, evaluated against the original graph, yields the exact answers to the original query. Since reformulation is made at query run-time, it is intrinsically robust to updates; reformulation is also typically very fast. However, reformulated queries are often syntactically more complex than the original ones, thus their evaluation may be costly.

RDF and SPARQL are complex languages with many features. For instance, the RDF specification supports a form of *incomplete information* through *blank nodes* and it provides a large set of *entailment rules* for deriving implicit information. The forthcoming SPARQL 1.1 supports aggregates, negation etc.

If saturation is used, one first chooses an RDF fragment and saturates the RDF graph accordingly. Then in a fully orthogonal way, one chooses the SPARQL dialect to evaluate on the graph thus saturated. In contrast, reformulation leads to a subtle interplay between the RDF and SPARQL dialects, since the query must be reformulated within the latter, so as to compute the query answers with respect to the former. Reformulation-based query answering has been studied for the Description Logics (DL) [6] fragment of RDF and the relational conjunctive SPARQL subset [7, 8, 9], and extensions thereof [10, 11, 12, 13].

**BGP query answering in the DB fragment of RDF** In this paper, we devise and compare *query answering techniques* for the *database (DB) fragment of RDF* and the *Basic Graph Pattern* (BGP) queries of SPARQL. This DB fragment, which we introduce, extends the fragments mentioned above, notably with the support of RDF blank nodes, encoding incomplete information. The Basic Graph Pattern (BGP) queries, identified in the SPARQL recommendation, are more expressive than relational conjunctive queries, since BGPs also allow *querying (and joining on) RDF relation names (called classes and properties)*, which conjunctive queries do not allow.

Importantly, our techniques are devised to work on top of *any conjunctive query evaluation engine*, and in particular: any standard RDBMS, as well as efficient native RDF engines such as [4]

©ACM, 2013. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *EDBT/ICDT '13,* March 18 - 22 2013, Genoa, Italy. http://doi.acm.org/10.1145/2452376.2452412

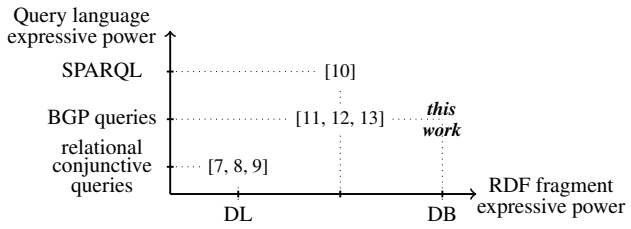



**Figure 1: Outline of the positioning of our work.**

which do not handle reasoning. We delegate RDF graph storage and BGP query evaluation to the underlying engine, while implementing on top of it the reasoning steps required to achieve sound and complete query answering. Thus, our approach allows extending the benefits of RDBMS scalability and reliability to a larger RDF fragment (the DB fragment) than in previous works. The positioning of our work with respect to the most prominent previous ones is sketched in Figure 1 (see Section 8 for details).

**Contributions** Our contributions can be summarized as follows.

1. We identify the novel DB fragment of RDF, extending fragments previously studied [10, 11, 12, 13] by the support of blank nodes.
2. We propose novel BGP query answering techniques for this DB fragment, designed to work on top of on any standard conjunctive query processor (and in particular any off-the-shelf RDBMS). Specifically, we provide: $(i)$ an efficient novel *incremental RDF saturation maintenance algorithm* (Section 5.2), based on which query answering reduces directly to query evaluation; and $(ii)$ a *novel reformulation-based query answering algorithm* (Section 6), required by the augmented expressive power of our DB fragment w.r.t. those in the literature.
3. We have implemented the above query answering techniques and deployed them on top of the Postgres RDBMS. Our experiments demonstrate their feasibility and efficiency on our novel, extended DB fragment.
   The best choice among saturation- or reformulation-based query answering depends on the queries, the amount of implicit data and the frequency and volume of updates to the data and/or schema. Our experiments study these possible cases and show which strategy works best for each of them.

The paper is organized as follows. Section 2 introduces RDF graphs and BGP queries. Section 3 relates RDF to relational databases. Section 4 defines our DB fragment of RDF, for which Section 4.2 presents saturation-based and reformulation-based query answering techniques. These techniques are then studied in-depth in Sections 5 and 6, and experimentally compared in Section 7. We discuss related work in Section 8, then we conclude.

# 2. RDF GRAPHS AND QUERIES

We introduce *RDF graphs*, modeling RDF datasets, in Section 2.1, and *Basic Graph Pattern queries* in Section 2.2.

## 2.1 RDF Graphs

An *RDF graph* (or *graph*, in short) is a set of *triples* of the form $\mathtt{s\ p\ o}$. A triple states that its *subject* $\mathtt{s}$ has the *property* $\mathtt{p}$, and the value of that property is the *object* $\mathtt{o}$. Given a set $U$ of URIs, a set $L$ of literals (constants), and a set $B$ of blank nodes (unknown URIs or literals), such that $U$, $B$ and $L$ are pairwise disjoint, a triple is *well-formed* whenever its subject belongs to $U \cup B$, its property belongs to $U$, and its object belongs to $U \cup B \cup L$. In what follows, we only consider well-formed triples.

| Constructor | Triple | Relational notation |
|---|---|---|
| Class assertion | $\mathtt{s}\ \tau\ \mathtt{o}$ | $\mathtt{o(s)}$ |
| Property assertion | $\mathtt{s\ p\ o}$ | $\mathtt{p(s, o)}$ |

**Figure 2: RDF statements.**

$$\mathtt{G} = \begin{array}{l}\{\mathrm{doi}_1\ \tau\ _{:}b_0,\ \mathrm{doi}_1\ \mathrm{hasTitle}\ \text{“}CAQUMV\text{”},\\ \mathrm{doi}_1\ \mathrm{hasAuthor}\ \text{“}SA\text{”},\ \mathrm{doi}_1\ \mathrm{hasContactA}\ _{:}b_1,\\ \mathrm{doi}_1\ \mathrm{inProceedingsOf}\ _{:}b_2,\ \mathrm{hasName}\ \mathrm{createdBy}\ \text{“}John\ Doe\text{”},\\ \mathrm{edbt2013}\ \tau\ \mathrm{conference},\ _{:}b_2\ \mathrm{hasName}\ \text{“}PODS'98\text{”}\}\end{array}$$

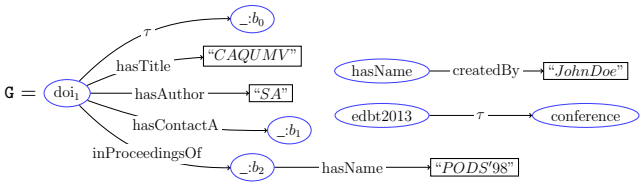



**Figure 3: Alternative RDF graph representations.**

Blank nodes are essential features of RDF allowing to support *unknown URI/literal tokens*. For instance, one can use a blank node $_{:}b_1$ to state that the country of $_{:}b_1$ is *Italy* while the city of the same $_{:}b_1$ is *Genoa*. Many such blank nodes can co-exist within a graph, e.g., one may also state that the country of $_{:}b_2$ is *Romania* while the city of $_{:}b_2$ is *Timişoara*; at the same time, the population of *Timişoara* can be said to be an unspecified value $_{:}b_3$.

**Notations** We use $\mathtt{s}$, $\mathtt{p}$, $\mathtt{o}$ and $_{:}b$ in triples (possibly with subscripts) as placeholders. That is, $\mathtt{s}$ stands for values in $U \cup B$, $\mathtt{p}$ stands for values in $U$, $\mathtt{o}$ represents values from $U \cup B \cup L$, and $_{:}b$ denotes values in $B$. Finally, we use strings between quotes as in “$string$” to denote literals.

Within the RDF standard [1], the built-in property *http://www.w3.org/1999/02/22-rdf-syntax-ns#type*, denoted $\tau$ from now on, is used to specify to which *classes* a resource belongs. This can be seen as a form of resource typing. Figure 2 shows how to use triples to describe resources.

A more intuitive representation of a graph can be drawn from its triples: every distinct subject or object value corresponds to a node labeled with this value; for each triple, there is a directed edge from the subject node to the object one, labeled with the property value.

**EXAMPLE 1 (RUNNING EXAMPLE).** *The representations in Figure 3 are equivalent as they model the same graph* $\mathtt{G}$. *This graph describes the resource* $\mathrm{doi}_1$ *that belongs to an unknown class, whose title is “Complexity of Answering Queries Using Materialized Views", whose author is “Serge Abiteboul" and having an unknown contact author. This paper is in the proceedings of an unknown resource whose name is “PODS'98". Lastly, the URI* $\mathrm{edbt2013}$ *is a conference and* $\mathrm{hasName}$*, the property associating names to resources, is created by “John Doe".*

**RDF Schema** A valuable feature of RDF is RDF Schema (RDFS) that allows enhancing the descriptions in graphs. An RDF Schema declares *semantic constraints* between the classes and the properties used in these graphs, through the use of RDF built-in properties modeling:

- subclass relationships, which we denote by the $\prec_{sc}$ symbol;
- subproperty relationships, denoted $\prec_{sp}$;
- typing the first attribute (domain) of a property, denoted $\hookleftarrow_d$;
- typing the second attribute (range) of a property, denoted $\hookrightarrow_r$.

Figure 4 shows the allowed constraints and how to express them. In this figure, $\mathtt{s}, \mathtt{o} \in U \cup B$, while $\mathrm{domain}$ and $\mathrm{range}$ denote re-

| Constructor | Triple | Relational notation |
|---|---|---|
| Subclass constraint | $\mathtt{s} \prec_{sc} \mathtt{o}$ | $\mathtt{s} \subseteq \mathtt{o}$ |
| Subproperty constraint | $\mathtt{s} \prec_{sp} \mathtt{o}$ | $\mathtt{s} \subseteq \mathtt{o}$ |
| Domain typing constraint | $\mathtt{s} \hookleftarrow_{d} \mathtt{o}$ | $\Pi_{\text{domain}}(\mathtt{s}) \subseteq \mathtt{o}$ |
| Range typing constraint | $\mathtt{s} \hookrightarrow_{r} \mathtt{o}$ | $\Pi_{\text{range}}(\mathtt{s}) \subseteq \mathtt{o}$ |

**Figure 4: RDFS statements.**

$$\mathtt{G}' = \mathtt{G} \cup \{\begin{array}{l}\text{posterCP} \prec_{sc} \text{confP},\\ _{:}b_0 \prec_{sc} \text{confP},\\ \text{confP} \prec_{sc} \text{paper},\\ \text{hasTitle} \hookleftarrow_{d} \text{paper},\\ \text{hasTitle} \hookrightarrow_{r} \text{rdfs:Literal},\\ \text{hasAuthor} \hookleftarrow_{d} \text{paper},\\ \text{hasAuthor} \hookrightarrow_{r} \text{rdfs:Literal},\\ \text{hasContactA} \prec_{sp} \text{hasAuthor},\\ \text{inProceedingsOf} \hookleftarrow_{d} \text{confP},\\ \text{inProceedingsOf} \hookrightarrow_{r} \text{conference},\\ \text{hasName} \hookleftarrow_{d} \text{conference},\\ \text{hasName} \hookrightarrow_{r} \text{rdfs:Literal},\\ \text{createdBy} \hookrightarrow_{r} \text{rdfs:Literal}\}\end{array}$$

**Figure 5: Extended RDF graph for Example 2.**

spectively the first and second attribute of every property. The figure also relates these constraints to relational inclusion constraints under the open-world assumption.

**Open-world interpretation of RDFS constraints** Traditionally, constraints can be interpreted in two ways [14]: under the closed-world assumption (CWA) or under the open-world assumption (OWA). Under CWA, any fact not present in the database is assumed not to hold. Under this assumption, if the set of database facts does not respect a constraint, then the database is *inconsistent*. For instance, the CWA interpretation of a constraint of the form $R_1 \subseteq R_2$ is: any tuple in the relation $R_1$ *must* also be in the relation $R_2$ *in the database*, otherwise the database is inconsistent. On the contrary, under OWA, some facts may hold even though they are *not in the database*. For instance, the OWA interpretation of the same example is: any tuple $t$ in the relation $R_1$ *is considered as being also in the relation* $R_2$ (the inclusion constraint *propagates* $t$ *to* $R_2$).

The RDF data model – and accordingly, the present work – is based on OWA, and this is how we interpret all the constraints in Figure 4. For instance, if the triples $\text{hasFriend} \hookleftarrow_{d} \text{Person}$ and $\text{Anne hasFriend Marie}$ hold in the graph, then so does the triple $\text{Anne} \tau \text{Person}$. The latter is due to the $\hookleftarrow_{d}$ constraint in Figure 4.

**EXAMPLE 2 (CONTINUED).** *Consider next to the above graph* G*, a schema stating that poster papers together with the unknown class* $_{:}b_0$ *of which* $\text{doi}_1$ *is an instance, are subclasses of conference papers, which are scientific papers. Moreover, titles, authors, and contact authors (themselves a particular case of authors) are used to describe papers, which are connected to the conferences in whose proceedings they appear. Finally, names describe conferences, and creators describe resources. The extended graph* $\mathtt{G}'$ *of* G *corresponding to this schema is depicted in Figure 5.*

**Entailment** Our discussion about OWA above illustrated an important RDF feature: *implicit triples*, considered to be part of the graph even though they are not explicitly present in it. The W3C names *entailment* the mechanism through which, based on the set of explicit triples and some *entailment rules* (to be described soon), implicit RDF triples are derived. We denote by $\vdash^{i}_{\text{RDF}}$ *immediate entailment*, i.e., the process of deriving new triples through a single application of an entailment rule. Then, (full) *RDF entailment* can be defined as follows: a triple $\mathtt{s\ p\ o}$ is entailed by a graph G, denoted $\mathtt{G} \vdash_{\text{RDF}} \mathtt{s\ p\ o}$, if and only if there is a sequence of applications of immediate entailment rules that leads from G to $\mathtt{s\ p\ o}$ (where at each step of the entailment sequence, the triples previously entailed are also taken into account).

**Graph saturation** The immediate entailment rules allow defining the finite *saturation* (a.k.a. *closure*) of a graph G, which is the graph, denoted $\mathtt{G}^{\infty}$, defined as the fixpoint of:

- $\mathtt{G}^0 = \mathtt{G}$
- $\mathtt{G}^{\alpha} = \mathtt{G}^{\alpha-1} \cup \{\mathtt{s\ p\ o} \mid \mathtt{G}^{\alpha-1} \vdash^{i}_{\text{RDF}} \mathtt{s\ p\ o}\}$

The saturation of a graph is unique (up to blank node renaming), and does not contain any implicit triples (they have all been made explicit by saturation). An obvious connection holds between the triples entailed by a graph G and its saturation: $\mathtt{G} \vdash_{\text{RDF}} \mathtt{s\ p\ o}$ if and only if $\mathtt{s\ p\ o} \in \mathtt{G}^{\infty}$. *Entailment is part of the RDF specification itself, and therefore the semantics of a graph is its saturation*, that is: any graph G is equivalent to, and models, its saturation $\mathtt{G}^{\infty}$.

**Immediate entailment rules** We give here an overview of the different kinds of immediate entailment rules upon which RDF entailment relies.

A first kind of rule generalizes triples using blank nodes. For instance:

$$\mathtt{s} \tau \mathtt{o} \vdash_{\text{RDF}} _{:}b \tau \mathtt{o}$$

Indeed, if s is an instance of o, then there exists an instance of o, namely the blank node $_{:}b$.

A second kind of rule derives entailed triples from the semantics of built-in classes and properties. E.g., RDF provides *http://www.w3.org/2000/01/rdf-schema#Class*, denoted $\mathcal{C}$, whose semantics is the set of all built-in and user-defined classes. Thus, if a resource is of type o, then o is a class:

$$\mathtt{s} \tau \mathtt{o} \vdash_{\text{RDF}} \mathtt{o} \tau \mathcal{C}$$

Finally, the third kind of rule derives entailed triples from the constraints modeled in an RDF Schema. Some rules derive entailed RDFS statements, through the transitivity of class and property inclusions, and from inheritance of domain and range typing. Using our running example:

$$\{_{:}b_0 \prec_{sc} \text{confP}, \text{confP} \prec_{sc} \text{paper}\} \vdash_{\text{RDF}} _{:}b_0 \prec_{sc} \text{paper}$$

and similarly:

$$\{\text{hasContactA} \prec_{sp} \text{hasAuthor}, \text{hasAuthor} \hookleftarrow_{d} \text{paper}\} \vdash_{\text{RDF}} \text{hasContactA} \hookleftarrow_{d} \text{paper}$$

Some other rules derive entailed RDF statements, through the propagation of values (URIs, blank nodes, and literals) from subclasses and sub-properties to their super-classes and super-properties, and from properties to classes typing their domains and ranges. Using our running example:

$$\{\text{hasContactA} \prec_{sp} \text{hasAuthor}, \text{doi}_1\ \text{hasContactA}\ _{:}b_1\} \vdash_{\text{RDF}} \text{doi}_1\ \text{hasAuthor}\ _{:}b_1, \text{ and moreover:}$$
$$\{\text{doi}_1\ \text{hasAuthor}\ _{:}b_1, \text{hasAuthor} \hookleftarrow_{d} \text{paper}\} \vdash_{\text{RDF}} \text{doi}_1 \tau \text{paper}$$

**Restricted rule sets** While these families of rules are part of the W3C specification [1], they are not all of equal interest. For instance, consider the generalization to blank nodes: it is probably more interesting to know that $\mathtt{s} \tau \mathtt{o}$ than to know that *some unknown* $_{:}b$ has that type. Similarly, the fact that $\mathcal{C}$ is an instance of itself is of limited interest. Other rules, such as those stating that, e.g., confP is a subclass of paper are comparatively much more useful. Accordingly, formally identified RDF fragments [10, 11, 12, 13] each consider only a useful subset of the existing rules. This is also our approach, as we will explain in Section 4.

## 2.2 BGP queries

We consider the well-known subset of SPARQL consisting of (unions of) basic graph pattern (BGP) queries.

A BGP is a *set of triple patterns*, or triples in short. Each triple has a subject, property and object. Subjects and properties can be URIs, blank nodes or variables; objects can also be literals.

We focus on the boolean BGP queries of the form `ASK WHERE` $\{t_1, \ldots, t_\alpha\}$, and on the non-boolean BGP queries of the form `SELECT` $\bar{x}$ `WHERE` $\{t_1, \ldots, t_\alpha\}$, where $\{t_1, \ldots, t_\alpha\}$ is a BGP, i.e., a set of triples modeling their conjunction; the variables $\bar{x}$ in the head of the query are called *distinguished variables*, and are a subset of the variables occurring in $t_1, \ldots, t_\alpha$.

**Notations** Without loss of generality, in the following we will use the conjunctive query notation $q(\bar{x})$:- $t_1, \ldots, t_\alpha$ for both `ASK` and `SELECT` queries (for boolean queries, $\bar{x}$ is empty). We use $x$, $y$, and $z$ (possibly with subscripts) to denote variables in queries. We denote by $\texttt{VarBl}(q)$ the set of variables *and* blank nodes occurring in the query $q$. The set of values (URIs, blank nodes, literals) of a graph G is denoted $\texttt{Val}(\texttt{G})$.

**Query evaluation** Given a query $q$ and a graph G, the *evaluation of q against* G is:

$$q(\texttt{G}) = \{\bar{x}_\mu \mid \mu : \texttt{VarBl}(q) \to \texttt{Val}(\texttt{G}) \text{ is a total assignment s.t. } (t_1, \ldots, t_\alpha)_\mu \subseteq \texttt{G}\}.$$

where for a given triple (or triple set) $O$, we denote by $O_\mu$ the result of replacing every occurrence of a variable or blank node $e \in \texttt{VarBl}(q)$ in $O$, by the value $\mu(e) \in \texttt{Val}(\texttt{G})$. If $q$ is boolean, the empty answer set encodes false, while the non-empty answer set made of the empty tuple $\emptyset_\mu = \langle\rangle$ encodes true.

Notice that (normative) query evaluation *treats the blank nodes in a query as non-distinguished variables*. That is, one could consider equivalently queries without blank nodes or queries without non-distinguished variables.

**Query answering** It is important to keep in mind the distinction between query *evaluation* and query *answering*. The evaluation of $q$ against G only uses G's explicit triples, thus may lead to an incomplete answer set. The (complete) *answer set* of $q$ against G is obtained by the evaluation of $q$ against $\texttt{G}^\infty$, denoted by $q(\texttt{G}^\infty)$.

**EXAMPLE 3 (CONTINUED).** *The following query asks for the authors of papers published in the proceedings of a conference somehow related to PODS'98:*

$$q(x) \text{:- } y_1 \text{ hasAuthor } x,\ y_1 \text{ inProceedingsOf } y_2,\ y_2\ y_3\ \text{“}PODS'98\text{”}$$

*The answer set of q against the graph* $\texttt{G}'$ *from Figure 5 is:* $q(\texttt{G}'^\infty) = \{\langle\text{“}SA\text{”}\rangle, \langle _{:}b_1\rangle\}$. *The answer “SA” results from the assignment:*

$$\mu = \{y_1 \to \text{doi}_1, x \to \text{“}SA\text{”}, y_2 \to _{:}b_2, y_3 \to \text{hasName}\}$$

*while the answer* $_{:}b_1$ *results from* $\texttt{G}' \vdash_{\text{RDF}} \text{doi}_1$ hasAuthor $_{:}b_1$ *and the assignment:*

$$\mu = \{y_1 \to \text{doi}_1, x \to _{:}b_1, y_2 \to _{:}b_2, y_3 \to \text{hasName}\}$$

*The evaluation of q against* $\texttt{G}'$ *leads to the incomplete answer set* $q(\texttt{G}') = \{\langle\text{“}SA\text{”}\rangle\}$, *which is a strict subset of* $q(\texttt{G}'^\infty)$.

# 3. RDF MEETS RDBMS

RDF graphs turn out to be a special case of incomplete relational databases based on *V-tables* [14, 15], which allow using variables in their tuples. Note that using a variable multiple times in a V-table allows expressing joins on unknown values.

An important result on V-table querying is that the standard relational evaluation (which sees variables in V-tables as constants) computes the exact answer set of any conjunctive query [14, 15]. From a practical viewpoint, this provides a possible way of answering conjunctive queries against V-tables using standard relational database management systems (RDBMSs, in short). We use the same observation to obtain complete answer sets to BGP queries using RDBMS evaluation, as follows.

Given a graph G, we encode it into the V-table $\texttt{Triple}(\texttt{s}, \texttt{p}, \texttt{o})$ storing the triples of G as tuples, in which blank nodes become variables. Then, given a BGP query $q(\bar{x})$:- $\texttt{s}_1\ \texttt{p}_1\ \texttt{o}_1, \ldots, \texttt{s}_n\ \texttt{p}_n\ \texttt{o}_n$, in which blank nodes have been equivalently replaced by fresh non-distinguished variables, the SPARQL evaluation $q(\texttt{G})$ of $q$ against G is obtained by the relational evaluation of the conjunctive query:

$$Q(\bar{x}) \text{:- } \textstyle\bigwedge_{i=1}^{n} \texttt{Triple}(\texttt{s}_i, \texttt{p}_i, \texttt{o}_i)$$

against the `Triple` table. Indeed, *SPARQL and relational evaluations coincide with the above encoding*, as relational evaluation amounts to finding all the total assignments from the variables of the query to the values (constants and variables) in the `Triple` table, so that the query becomes a subset of that `Triple` table.

It follows that evaluating $Q(\bar{x})$:- $\bigwedge_{i=1}^{n} \texttt{Triple}(\texttt{s}_i, \texttt{p}_i, \texttt{o}_i)$ against the `Triple` table containing the saturation of G, instead of G itself, computes the answer set of $q$ against G. In other words, BGP query answering boils down to conjunctive query evaluation on a saturated database.

Conceptually, the above observation is the reason why the simple RDF query answering aproach taken in previous works such as [3, 5, 16, 17, 18] is sound. Our saturation-based query answering algorithm (Section 5) also immediately follows from this observation. In contrast, as we will show, our reformulation-based query answering technique (Section 6) requires a quite subtler approach.

| Rule name [1] | Triples | Entailed triple ($\vdash^i_{\text{RDF}}$) |
|---|---|---|
| rdfs11 | $\texttt{s} \prec_{sc} \texttt{s}_1, \texttt{s}_1 \prec_{sc} \texttt{s}_2$ | $\texttt{s} \prec_{sc} \texttt{s}_2$ |
| rdfs5 | $\texttt{p} \prec_{sp} \texttt{p}_1, \texttt{p}_1 \prec_{sp} \texttt{p}_2$ | $\texttt{p} \prec_{sp} \texttt{p}_2$ |
| ext1 | $\texttt{p} \hookleftarrow_d \texttt{s}_1, \texttt{s}_1 \prec_{sc} \texttt{s}$ | $\texttt{p} \hookleftarrow_d \texttt{s}$ |
| ext2 | $\texttt{p} \hookrightarrow_r \texttt{s}_1, \texttt{s}_1 \prec_{sc} \texttt{s}$ | $\texttt{p} \hookrightarrow_r \texttt{s}$ |
| ext3 | $\texttt{p} \prec_{sp} \texttt{p}_1, \texttt{p}_1 \hookleftarrow_d \texttt{s}$ | $\texttt{p} \hookleftarrow_d \texttt{s}$ |
| ext4 | $\texttt{p} \prec_{sp} \texttt{p}_1, \texttt{p}_1 \hookrightarrow_r \texttt{s}$ | $\texttt{p} \hookrightarrow_r \texttt{s}$ |

**Figure 6: Schema-level entailment from two schema triples.**

| Rule name [1] | Triples | Entailed triple ($\vdash^i_{\text{RDF}}$) |
|---|---|---|
| rdfs9 | $\texttt{s}_1 \prec_{sc} \texttt{s}_2, \texttt{s}\ \tau\ \texttt{s}_1$ | $\texttt{s}\ \tau\ \texttt{s}_2$ |
| rdfs7 | $\texttt{p}_1 \prec_{sp} \texttt{p}_2, \texttt{s}\ \texttt{p}_1\ \texttt{o}$ | $\texttt{s}\ \texttt{p}_2\ \texttt{o}$ |
| rdfs2 | $\texttt{p} \hookleftarrow_d \texttt{s}, \texttt{s}_1\ \texttt{p}\ \texttt{o}_1$ | $\texttt{s}_1\ \tau\ \texttt{s}$ |
| rdfs3 | $\texttt{p} \hookrightarrow_r \texttt{s}, \texttt{s}_1\ \texttt{p}\ \texttt{o}_1$ | $\texttt{o}_1\ \tau\ \texttt{s}$ |

**Figure 7: Instance-level entailment from combining schema and instance triples.**

# 4. QUERYING THE DATABASE FRAGMENT OF RDF

We are now ready to set the stage for our main algorithmic contributions. We introduce our database fragment of RDF in Section 4.1, and then state an important result on answering BGP queries within the DB fragment in Section 4.2.

## 4.1 The DB fragment of RDF

We define a restriction of RDF that we call its *database (DB) fragment*, aiming simultaneously at an expressive fragment, and at one for which saturation- and reformulation-based query *answering* can be efficiently implemented on top of any conjunctive query *evaluation* engine, be it an RDBMS or a reasoning-agnostic RDF engine. This DB fragment is obtained by:

- *restricting RDF entailment to the rules dedicated to RDF Schema* only (a.k.a. RDFS entailment). The main ones are

shown in Figures 6–7, while the others derive tautologies (e.g., , a class is a subclass of itself etc.) and can be found in our technical report [19];

- *not restricting graphs in any way*. In other words, any triple allowed by the RDF specification is also allowed in the DB fragment.

We call a graph belonging to our DB fragment a *database*. A database db is a pair $\langle \mathtt{S}, \mathtt{D} \rangle$, where S and D are two disjoint sets of triples. S triples can only be RDFS statements such as shown in Figure 4. We call these triples the *schema-level* of db. The other triples, of the forms listed in Figure 2, belong to D, and are called the *instance-level* of db. Observe that S and D provide a way to partition any graph (any triple belongs to exactly one of them).

The saturation of a database db with this aforementioned entailment rule set is denoted $\mathtt{db}^{\infty}$, thus $\mathtt{db}^{\infty} \subseteq \mathtt{db}^{\infty\infty}$.

**BGP queries on the DB fragment** We use the BGP queries introduced in Section 2.2 to query the DB fragment of RDF. The *evaluation of a query q against a database* db is exactly the evaluation of $q$ against the graph db, i.e., $q(\mathtt{db})$, and the *answer set of q against* db is $q(\mathtt{db}^{\infty})$, thus $q(\mathtt{db}^{\infty}) \subseteq q(\mathtt{db}^{\infty\infty})$.

User queries may traverse both the schema- and instance-level of the database. In our running example, one can ask for the ranges of the properties describing conference papers:

$$\text{ClassRelatedToConfPaper}(x) \text{:- } y_1 \ \tau \ \text{confP}, y_1 \ y_2 \ y_3, y_2 \ \hookrightarrow_r \ x$$

In some settings, however, the separation between schema and data, which corresponds to many users' intuitive comprehension of the database, may lead them to specify that their queries be evaluated *only against the instance-level* or *only against the schema-level* database.

From a database perspective, queries whose evaluation is asked against the (saturated) instance-level database only are the most familiar. While schema triples are not returned by such queries, they do impact their answer, because the saturation of the instance-level database (necessary in order to return complete answers) relies on the schema-level triples. For instance, an instance-level query returning the city of EDBT 2013 is:

$$\text{EDBTCity}(y) \text{:- } x \text{ name “EDBT2013”}, x \text{ city } y$$

Instance-level queries can also return classes and properties associated to specific values. For instance, one can ask for the classes to which a given value $v$ belongs, i.e., $\text{ClassFinding}(x)$:- $v \ \tau \ x$

From a knowledge representation perspective, a class of interesting queries can be evaluated over the schema-level database alone. Such queries offer a convenient means to *explore the relationships between the classes and properties of a schema, including the implied relationships*. For instance, one can ask whether a given class is a subclass of another:

$$\text{SubclassChecking}() \text{:- posterCP} \ \prec_{sc} \ \text{paper}$$

or, what are the classes typing the domain of a given property:

$$\text{DomainFinding}(x) \text{:- inProceedingsOf} \ \hookleftarrow_d \ x$$

Another schema exploration query is:

$$\text{Schema}(x, y, z) \text{:- } x \ y \ z$$

returning all triples of the database. By restricting the query to only the schema-level database, the user retrieves all direct or entailed relationships among classes and properties.

The above discussion shows that our setting is general enough to integrate both database-style instance-level querying and knowledge representation-style schema-level querying, while also allowing a smooth integration of both levels through queries on both database components.

In the following, given the overwhelming practical impact of querying only the instance-level (implicit and explicit) data, we focus on efficient query answering algorithms for this problem.

$$\frac{\{\mathbf{c}_1 \ \prec_{sc} \ \mathbf{c}_2, \mathtt{s} \ \tau \ \mathbf{c}_1\} \subseteq \mathtt{db}}{\mathtt{db} = \mathtt{db} \cup \{\mathtt{s} \ \tau \ \mathbf{c}_2\}} \quad (1)$$

$$\frac{\{\mathbf{p} \ \hookleftarrow_d \ \mathbf{c}, \mathtt{s} \ \mathbf{p} \ \mathtt{o}\} \subseteq \mathtt{db}}{\mathtt{db} = \mathtt{db} \cup \{\mathtt{s} \ \tau \ \mathbf{c}\}} \quad (2)$$

$$\frac{\{\mathbf{p} \ \hookrightarrow_r \ \mathbf{c}, \mathtt{s} \ \mathbf{p} \ \mathtt{o}\} \subseteq \mathtt{db}}{\mathtt{db} = \mathtt{db} \cup \{\mathtt{o} \ \tau \ \mathbf{c}\}} \quad (3)$$

$$\frac{\{\mathbf{p}_1 \ \prec_{sp} \ \mathbf{p}_2, \mathtt{s} \ \mathbf{p}_1 \ \mathtt{o}\} \subseteq \mathtt{db}}{\mathtt{db} = \mathtt{db} \cup \{\mathtt{s} \ \mathbf{p}_2 \ \mathtt{o}\}} \quad (4)$$

**Figure 8: Saturation rules for an RDF database db.**

## 4.2 Query answering in RDF databases

We investigate two query answering techniques against RDF databases: *saturation-* and *reformulation-based*. Each technique performs a specific *pre-processing* step, either on the database or on the queries, to deal with entailed triples, after which query answering is reduced to query evaluation.

*Saturation-based query answering* is rather straightforward: the saturation of the database is computed using the allowed entailment rules; then, the answer set of every query against the (original) database is obtained by query evaluation against the saturation. The advantage of this approach is that it is easy to implement. Its disadvantages are that database saturation needs time to be computed and space to store all the entailed triples; moreover, the saturation must be somehow recomputed upon every database update.

*Reformulation-based query answering* reformulates a query $q$ w.r.t. a database db into another query $q'$ (using the immediate entailment rules), so that the evaluation of $q'$ against the (original) database db, denoted $q'(\mathtt{db})$, is exactly the answer set of $q$ against db (i.e., $q(\mathtt{db}^{\infty})$). The advantage of reformulation is that the database saturation does not need to be (re)computed. The disadvantage is that every incoming query must be reformulated, which often results in a more complex query.

We focus on saturation- and reformulation-based query answering only for *instance-level queries*. The following theorem shows that to answer such queries, among our DB fragment's rules shown in Figures 6–7 and [19], it suffices to consider only the entailment rules in Figure 7:

THEOREM 1. *Let* db *be a database,* $t_1$ *be a triple of the form* $\mathtt{s} \ \tau \ \mathtt{o}$*, and* $t_2$ *be a triple of the form* $\mathtt{s} \ \mathtt{p} \ \mathtt{o}$*.* $t_1 \in \mathtt{db}^{\infty}$ *(respectively,* $t_2 \in \mathtt{db}^{\infty}$*) iff there exists a sequence of application of the rules in Figure 7 leading from* db *to* $t_1$ *(respectively* $t_2$*), assuming that each entailment step relies on* db *and all triples previously entailed.*

The proofs of all the theorems of this paper can be found in [19].

# 5. SATURATION-BASED QUERY ANSWERING

Our first and simplest query answering technique resembles those previously discussed from the literature [3, 5, 16, 17, 18]: compute the instance-level saturation of a given database, then evaluate the original query against it (Section 5.1). Our main contribution in the area of saturation-based query answering is to show how to efficiently handle *database changes at the instance- or schema-level*: we provide a novel incremental algorithm for saturation maintenance (Section 5.2) and formally establish its correctness (Section 5.3).

## 5.1 Database saturation

Our Saturate query answering algorithm relies on the saturation rules in Figure 8, which are a direct implementation of the entailment rules in Figure 7. In Figure 8 and in the sequel, the bold symbols (possibly with subscripts) $\mathbf{c}$ for a class, and $\mathbf{p}$ for a property, denote some unspecified values.

The rules in Figure 8 define a set of database transformations of the form $\frac{input}{output}$, where both the input and the output are databases. Intuitively, given a database db, $\mathtt{Saturate}(\mathtt{db})$ applies exhaustively the rules in Figure 8, on db plus all the gradually generated triples.

The output of $\mathtt{Saturate}(\mathtt{db})$ is defined as the fixpoint $\mathtt{Saturate}^{\infty}(\mathtt{db})$, where:

$\mathtt{Saturate}^0(\mathtt{db}) = \mathtt{db}$
$\mathtt{Saturate}^{k+1}(\mathtt{db}) = \mathtt{Saturate}^k(\mathtt{db}) \cup \{t_3 \mid \exists i \in [1,4]$ s.t.
applying rule $(i)$ on some $\{t_1, t_2\} \subseteq \mathtt{Saturate}^k(\mathtt{db})$
produces $t_3$, where $t_2 \notin \mathtt{Saturate}^{k-1}(\mathtt{db})$ when defined$\}$

**EXAMPLE 4 (CONTINUED).** *The saturation of* db*, containing the previously described graph* $\mathtt{G}'$*, is shown in Figure 9.*

Theorem 2 shows that our saturation algorithm terminates. It also provides upper bounds for the size of the output saturation and its computation time.

THEOREM 2. *For a database* db*, the size (number of triples) of the output of* $\mathtt{Saturate}(\mathtt{db})$ *is in* $O(\#\mathtt{db}^2)$ *and the time to compute it is in* $O(\#\mathtt{db}^3)$*, with* $\#\mathtt{db}$ *the size (number of triples) of* db*.*

Our experiments (Table 1 in Section 7) show that in practical cases, $\mathtt{Saturate}(\mathtt{db})$ has a more moderate size, but it can still be significantly larger than db; moreover, in practical databases, the theoretical time complexity is far from being reached (Figure 14).

## 5.2 Saturation maintenance upon updates

Saturation-based query answering is efficient at query time, since one only has to evaluate the original query. However, the saturation must be somehow recomputed to reflect the impact of updates.

In this section, we study the problem of efficiently maintaining the database saturation upon two kinds of updates: triple *insertion* and *deletion*. Taking inspiration from the rich literature on incremental view maintenance in databases [20], our purpose is to devise *incremental* algorithms, which do not re-compute the saturation but just modify it to reflect the update.

An important issue is to keep track of the *multiple ways in which a triple was entailed (i.e., derived)*. This is significant when considering both implicit data and updates: for a given update, we must decide whether this adds/removes one *reason why a triple belongs to the saturation*. When this count reaches 0, the implied triple should be removed. A naïve implementation would record the inference paths of each implied triple, that is: all sequences of reasoning rules that have lead to that triple being present in the saturation. However, as shown in [21], the volume of such justification grows very fast and thus the approach does not scale. Instead, we chose to keep track of the *number of reasons* why a triple has been inferred, and provide maintenance algorithms which rely only on this (much more compact) information.

In order to explain our algorithms, we extend the previous notion of database saturation, so that it becomes a *multiset* in which a triple appears as many times as it can be entailed. Formally, given a database db, the saturation is now defined as the fixpoint $\mathtt{Saturate}_+^{\infty}(\mathtt{db})$ obtained from the following $\mathtt{Saturate}_+$ algorithm, where $\uplus$ is the union operator for multisets.

$\mathtt{Saturate}^0(\mathtt{db}) = \mathtt{db}$
$\mathtt{Saturate}^1(\mathtt{db}) = \mathtt{Saturate}^0(\mathtt{db}) \cup \{\mathrm{doi}_1\ \tau\ \mathrm{confP}\ ,$
$\mathrm{doi}_1\ \tau\ \mathrm{paper}\ ,\ \ \mathrm{doi}_1\ \mathrm{hasAuthor}\ _{:}b_1\ ,$
$_{:}b_2\ \tau\ \mathrm{conference}\}$
$\mathtt{Saturate}^2(\mathtt{db}) = \mathtt{Saturate}^1(\mathtt{db})$

**Figure 9: Sample saturation of a database** db**.**

$\mathtt{Saturate}_+^0(\mathtt{db}) = \mathtt{db}$
$\mathtt{Saturate}_+^1(\mathtt{db}) = \mathtt{Saturate}_+^0(\mathtt{db}) \uplus \{\mathrm{doi}_1\ \tau\ \mathrm{confP}\ ,$
$\mathrm{doi}_1\ \tau\ \mathrm{paper}\ ,\ \ \mathrm{doi}_1\ \tau\ \mathrm{paper}\ ,$
$\mathrm{doi}_1\ \mathrm{hasAuthor}\ _{:}b_1\ ,\ \ \mathrm{doi}_1\ \tau\ \mathrm{confP}\ ,$
$_{:}b_2\ \tau\ \mathrm{conference}\ ,\ \ _{:}b_2\ \tau\ \mathrm{conference}\}$
$\mathtt{Saturate}_+^2(\mathtt{db}) = \mathtt{Saturate}_+^1(\mathtt{db}) \uplus \{\mathrm{doi}_1\ \tau\ \mathrm{paper}\ ,$
$\mathrm{doi}_1\ \tau\ \mathrm{paper}\ ,\ \ \mathrm{doi}_1\ \tau\ \mathrm{paper}\}$
$\mathtt{Saturate}_+^3(\mathtt{db}) = \mathtt{Saturate}_+^2(\mathtt{db})$

**Figure 10: Sample multiset-based saturation.**

$\mathtt{Saturate}_+^0(\mathtt{db}) = \mathtt{db}$
$\mathtt{Saturate}_+^{k+1}(\mathtt{db}) = \mathtt{Saturate}_+^k(\mathtt{db}) \biguplus \{t_3 \mid \exists i \in [1,4]$ s.t.
applying rule $(i)$ on some $\{t_1, t_2\} \subseteq \mathtt{Saturate}_+^k(\mathtt{db})$
produces $t_3$ with $t_2 \notin \mathtt{Saturate}_+^{j<k}(\mathtt{db})$ when defined$\}$

The following Property expresses the obvious relationship between the set-based saturation and the multiset-based saturation of a database; $\mathtt{set}(\cdot)$ returns the set of (distinct) elements occurring in a given multiset.

PROPERTY 1. $\mathtt{Saturate}(\mathtt{db}) = \mathtt{set}(\mathtt{Saturate}_+(\mathtt{db}))$ *holds for any RDF database* db*.*

**EXAMPLE 5 (CONTINUED).** *Consider again the previously introduced database* db*. Its multiset-based saturation is shown in Figure 10.*

With the multiset-based saturation and Property 1 in place, Theorem 3 shows how saturation can be *incrementally* maintained upon update; $\uplus$ is again multiset union, while $\setminus_+$ is multiset difference.

THEOREM 3. *Let* $\mathtt{db} = \langle \mathtt{S}, \mathtt{D} \rangle$ *be a database.*
*Insertion:* $\mathtt{Saturate}_+(\mathtt{db} \cup \{t\}) =$

- $\mathtt{Saturate}_+(\mathtt{db})$ *if* $t \in \mathtt{db}$*. Otherwise,* $t \notin \mathtt{db}$ *and:*
- $\mathtt{Saturate}_+(\mathtt{db}) \uplus [\mathtt{Saturate}_+(\langle \mathtt{S}, \{t\} \rangle) \setminus_+ \mathtt{S}]$ *if* $t$ *is an instance-level triple;*
- $\mathtt{Saturate}_+(\mathtt{db}) \uplus \{t\} \uplus \biguplus_{t' \in \mathtt{D}'} [\mathtt{Saturate}_+(\langle \mathtt{S}, \{t'\} \rangle) \setminus_+ \mathtt{S}]$*, where the multiset* $\mathtt{D}'$ *is* $\{t_3 \mid \exists i \in [1,4]$ *such that applying rule (i) on* $\{t, t_2\}$ *with* $t_2 \in \mathtt{Saturate}_+(\mathtt{db})$ *yields* $t_3\}$*, if* $t$ *is a schema-level triple.*

*Deletion:* $\mathtt{Saturate}_+(\mathtt{db} \setminus \{t\}) =$

- $\mathtt{Saturate}_+(\mathtt{db})$ *if* $t \notin \mathtt{db}$*. Otherwise,* $t \in \mathtt{db}$ *and:*
- $\mathtt{Saturate}_+(\mathtt{db}) \setminus_+ [\mathtt{Saturate}_+(\langle \mathtt{S}, \{t\} \rangle) \setminus_+ \mathtt{S}]$ *if* $t$ *is an instance-level triple;*
- $\mathtt{Saturate}_+(\mathtt{db}) \setminus_+ \{t\} \setminus_+ \biguplus_{t' \in \mathtt{D}'} [\mathtt{Saturate}_+(\langle \mathtt{S}, \{t'\} \rangle) \setminus_+ \mathtt{S}]$ *where the multiset* $\mathtt{D}'$ *is* $\{t_3 \mid \exists i \in [1,4]$ *such that applying rule (i) on* $\{t, t_2\}$ *with* $t_2 \in \mathtt{Saturate}_+(\mathtt{db})$ *yields* $t_3\}$*, if* $t$ *is a schema-level triple.*

Theorem 3 reads as follows. Inserting a triple already in the database, or deleting a triple that is not in the database, does not require any work. Otherwise, inserting (deleting) a given instance

$\texttt{Saturate}_+(\texttt{db}) = \texttt{S} \uplus \{(\text{doi}_1\ \tau\ _{:}b_0\ \text{T}\ 1)\ ,$
$(\text{doi}_1\ \text{hasTitle}\ \text{“}CAQUMV\text{”}\ \text{T}\ 1)\ ,$
$(\text{doi}_1\ \text{hasAuthor}\ \text{“}SA\text{”}\ \text{T}\ 1)\ ,$
$(\text{doi}_1\ \text{hasContactA}\ _{:}b_1\ \text{T}\ 1)\ ,$
$(\text{doi}_1\ \text{inProceedingsOf}\ _{:}b_2\ \text{T}\ 1)\ ,$
$(\text{hasName createdBy}\ \text{“}John\ Doe\text{”}\ \text{T}\ 1)\ ,$
$(\text{edbt2013}\ \tau\ \text{conference}\ \text{T}\ 1)\ ,$
$(_{:}b_2\ \text{hasName}\ \text{“}PODS'98\text{”}\ \text{T}\ 1)\ ,$
$(\text{doi}_1\ \tau\ \text{confP}\ \text{F}\ 2)\ , (\text{doi}_1\ \tau\ \text{paper}\ \text{F}\ 5)\ ,$
$(\text{doi}_1\ \text{hasAuthor}\ _{:}b_1\ \text{F}\ 1)\ , (_{:}b_2\ \tau\ \text{conference}\ \text{F}\ 2)\}$

**Figure 11: Sample compact representation of $\texttt{Saturate}_+(\texttt{db})$.**

or schema triple also adds to (removes from) the current saturation all instance-level triples whose derivation uses this given triple.

**Optimized implementation** From a practical viewpoint, the multiset $\texttt{Saturate}_+(\texttt{db})$ can be compactly stored (Figure 11) as the set $\texttt{Saturate}(\texttt{db})$, for which every triple is tagged with:

$(i)$ a boolean indicating whether it *belongs to* db (T) or is *only entailed by* db (F), and

$(ii)$ an integer indicating how many times it appears in $\texttt{Saturate}_+(\texttt{db})$.

The above provides a single lightweight representation for db, $\texttt{Saturate}(\texttt{db})$, and $\texttt{Saturate}_+(\texttt{db})$.

**EXAMPLE 6 (CONTINUED).** *Given $\texttt{Saturate}_+(\texttt{db})$ for our running example (Figures 10 and 11), we exemplify below the saturation maintenance process for an instance insertion and a schema deletion.*

*(i) The insertion of $t = \text{doi}_2$ inProceedingsOf edbt2013, entails ($\texttt{Saturate}_+(\langle \texttt{S}, \{t\}\rangle) \setminus_+ \texttt{S}$) three new triples, one of which already exists explicitly in the saturation:*

$\texttt{Saturate}_+(\texttt{db}) = \{\ \ldots\ , (\text{edbt2013}\ \tau\ \text{conference}\ \text{T}\ \mathbf{2})\ ,$
$\ldots\ , (\text{doi}_2\ \text{inProceedingsOf edbt2013}\ \text{T}\ 1)\ ,$
$(\text{doi}_2\ \tau\ \text{confP}\ \text{F}\ 1)\ , (\text{doi}_2\ \tau\ \text{paper}\ \text{F}\ 1)\ \}$

*(ii) Deleting the schema triple $t = \text{hasContactA} \prec_{sp} \text{hasAuthor}$ causes its removal from $\texttt{Saturate}_+(\texttt{db})$, together with all the instance triples entailed by it and the database instance: $\texttt{D}' = \{\text{doi}_1\ \text{hasAuthor}\ _{:}b_1\}$, and the instance triples entailed by $\texttt{D}'$ and the schema: $\biguplus_{t' \in \texttt{D}'}[\texttt{Saturate}_+(\langle \texttt{S}, \{t'\}\rangle) \setminus_+ \texttt{S}] = \{\text{doi}_1\ \tau\ \text{paper}\}$. Notice that only one instance of $\text{doi}_1\ \tau\ \text{paper}$ is removed (the count is decreased to **4**), as it is still entailed by other facts.*

## 5.3 From database saturation to saturation-based query answering

Based on the above notion of saturation, we state our saturation-based query answering technique:

THEOREM 4. *Given a BGP query q and a database db, the following holds:*

$$q(\texttt{db}^\infty) = q(\texttt{Saturate}(\texttt{db})) = q(\texttt{set}(\texttt{Saturate}_+(\texttt{db}))).$$

The proof for Theorem 4 trivially follows from Theorem 1, the definition of our Saturate algorithm, and Property 1.

From a practical viewpoint, and based on our observation from Section 3, saturation-based query answering can be delegated to an RDBMS by $(i)$ storing either $\texttt{Saturate}(\texttt{db})$ or the aforementioned compact representation of $\texttt{Saturate}_+(\texttt{db})$ in the Triple table, and $(ii)$ evaluating queries using the RDBMS engine.

**EXAMPLE 7 (CONTINUED).** *Consider the query $q(x, y)$:- $x\ \tau\ y$ asking for all resources and the classes to which they belong. The answer set of q against the previous database db is:*

$$q(\texttt{Saturate}(\texttt{db})) = q(\texttt{set}(\texttt{Saturate}_+(\texttt{db}))) =$$
$$\{\langle \text{doi}_1, _{:}b_0\rangle, \langle \text{doi}_1, \text{confP}\rangle,$$
$$\langle \text{edbt2013}, \text{conference}\rangle,$$
$$\langle \text{doi}_1, \text{paper}\rangle, \langle _{:}b_2, \text{conference}\rangle\}.$$

# 6. REFORMULATION-BASED QUERY ANSWERING

We now turn to query answering through reformulation. Given a query $q$ and a database db, Section 6.1 describes our algorithm for reformulating queries in our DB fragment. The expressive power of our RDF DB fragment, however, widens the gap between query answering and conjunctive query evaluation: in our setting, simply evaluating the queries resulting from reformulation does not suffice to compute the correct result. To bridge this gap, Section 6.2 introduces our novel *non-standard query evaluation*, which, applied on reformulated queries, computes (still relying on an RDF-agnostic conjunctive query processor, e.g., an RDBMS) the sound and complete answer sets in our expressive DB fragment.

## 6.1 Query reformulation

Our Reformulate algorithm exhaustively applies the set of rules shown in Figure 12, starting from a query $q$ and a database db. Each rule defines a transformation of the form $\frac{input}{output}$, where the input is of the form $\langle$logical condition on db, logical condition on $q\rangle$ and the output is a query $q'$. Each but not both of the conditions in the input may be empty. Intuitively, each rule produces a new query when the rule's input conditions are satisfied, one by the database db, and the other by some query (either the original query $q$ or a query $q'$ produced by a previous application of a rule). The set of all queries produced by applying the rules is the result of the reformulation of $q$ w.r.t. db.

A key concept for our reformulation-based query answering are:

**Partially instantiated queries** Let $q(\bar{x})$:- $t_1, \ldots, t_\alpha$ be a query and $\sigma$ be a mapping from a subset of $q$'s variables and blank nodes, to some values (URIs, blank nodes, or literals).

The *partially instantiated query* $q_\sigma$ is a query $q_\sigma(\bar{x}_\sigma)$:- $(t_1, \ldots, t_\alpha)_\sigma$ where $\sigma$ has been applied both on $q$'s head variables $\bar{x}$ and on $q$'s body. Observe that, in non-standard fashion, some distinguished (head) variables of $q_\sigma$ can be bound! If $\sigma = \emptyset$, then $q_\sigma$ coincides with the original (non-instantiated) query $q$.

By allowing constants in the head, partially instantiated queries go outside the reach of our definition of BGP queries. Accordingly, a slight extension is required to the notions of BGP query evaluation and answer sets, introduced in Section 2.2 for graphs and in Section 4 for databases, as follows.

Given a database db whose set of values (URIs, blank nodes, literals) is $\texttt{Val}(\texttt{db})$ and a query $q_\sigma(\bar{x}_\sigma)$:- $(t_1, \ldots, t_\alpha)_\sigma$ whose set of variables and blank nodes is $\texttt{VarBl}(q_\sigma)$, the *evaluation* of $q_\sigma$ against db is:

$$q_\sigma(\texttt{db}) = \{(\bar{x}_\sigma)_\mu \mid \mu : \texttt{VarBl}(q_\sigma) \to \texttt{Val}(\texttt{db}) \text{ is a total assignment } ((t_1, \ldots, t_\alpha)_\sigma)_\mu \subseteq \texttt{db}\}$$

The *answer set* of $q_\sigma$ against db is the evaluation of $q_\sigma$ against $\texttt{db}^\infty$, denoted $q_\sigma(\texttt{db}^\infty)$.

**Reformulation rules** The rules (5)–(13) reformulate queries by binding one of their variables, either to the built-in property $\tau$ or to a class or property name picked in the database. The other rules (14)–(17) replace some query triple with another, based on schema-level triples.

Consider for instance rule (5). The rule says: if a triple of the form s $y$ o, i.e., having any kind of subject or object, but having

$$\frac{\langle \mathtt{s}\ y\ \mathtt{o} \in q_\sigma \rangle}{q_{\sigma\cup\nu=\{y\to\tau\}}} \quad (5)$$

$$\frac{\langle \mathtt{s}_1\ \mathbf{p}\ \mathtt{o}_1 \in \mathtt{db}, \mathtt{s}\ y\ \mathtt{o} \in q_\sigma \rangle}{q_{\sigma\cup\nu=\{y\to\mathbf{p}\}}} \quad (6)$$

$$\frac{\langle \mathtt{s}_1\ \prec_{sp}\ \mathbf{p} \in \mathtt{db}, \mathtt{s}\ y\ \mathtt{o} \in q_\sigma \rangle}{q_{\sigma\cup\nu=\{y\to\mathbf{p}\}}} \quad (7)$$

$$\frac{\langle \mathbf{p}\ \prec_{sp}\ \mathtt{o}_1 \in \mathtt{db}, \mathtt{s}\ y\ \mathtt{o} \in q_\sigma \rangle}{q_{\sigma\cup\nu=\{y\to\mathbf{p}\}}} \quad (8)$$

$$\frac{\langle \mathtt{s}_1\ \tau\ \mathbf{c} \in \mathtt{db}, \mathtt{s}\ \tau\ z \in q_\sigma \rangle}{q_{\sigma\cup\nu=\{z\to\mathbf{c}\}}} \quad (9)$$

$$\frac{\langle \mathtt{s}_1\ \prec_{sc}\ \mathbf{c} \in \mathtt{db}, \mathtt{s}\ \tau\ z \in q_\sigma \rangle}{q_{\sigma\cup\nu=\{z\to\mathbf{c}\}}} \quad (10)$$

$$\frac{\langle \mathbf{c}\ \prec_{sc}\ \mathtt{o} \in \mathtt{db}, \mathtt{s}\ \tau\ z \in q_\sigma \rangle}{q_{\sigma\cup\nu=\{z\to\mathbf{c}\}}} \quad (11)$$

$$\frac{\langle \mathtt{s}_1\ \hookleftarrow_d\ \mathbf{c} \in \mathtt{db}, \mathtt{s}\ \tau\ z \in q_\sigma \rangle}{q_{\sigma\cup\nu=\{z\to\mathbf{c}\}}} \quad (12)$$

$$\frac{\langle \mathtt{s}_1\ \hookrightarrow_r\ \mathbf{c} \in \mathtt{db}, \mathtt{s}\ \tau\ z \in q_\sigma \rangle}{q_{\sigma\cup\nu=\{z\to\mathbf{c}\}}} \quad (13)$$

$$\frac{\langle \mathbf{c}_1\ \prec_{sc}\ \mathbf{c}_2 \in \mathtt{db}, \mathtt{s}\ \tau\ \mathbf{c}_2 \in q_\sigma \rangle}{q_{\sigma[\mathtt{s}\ \tau\ \mathbf{c}_2/\mathtt{s}\ \tau\ \mathbf{c}_1]}} \quad (14)$$

$$\frac{\langle \mathbf{p}\ \hookleftarrow_d\ \mathbf{c} \in \mathtt{db}, \mathtt{s}\ \tau\ \mathbf{c} \in q_\sigma \rangle}{q_{\sigma[\mathtt{s}\ \tau\ \mathbf{c}/\mathtt{s}\ \mathbf{p}\ y]}} \quad (15)$$

$$\frac{\langle \mathbf{p}\ \hookrightarrow_r\ \mathbf{c} \in \mathtt{db}, \mathtt{s}\ \tau\ \mathbf{c} \in q_\sigma \rangle}{q_{\sigma[\mathtt{s}\ \tau\ \mathbf{c}/y\ \mathbf{p}\ \mathtt{s}]}} \quad (16)$$

$$\frac{\langle \mathbf{p}_1\ \prec_{sp}\ \mathbf{p}_2 \in \mathtt{db}, \mathtt{s}\ \mathbf{p}_2\ \mathtt{o} \in q_\sigma \rangle}{q_{\sigma[\mathtt{s}\ \mathbf{p}_1\ \mathtt{o}/\mathtt{s}\ \mathbf{p}_2\ \mathtt{o}]}} \quad (17)$$

**Figure 12: Reformulation rules for a partially instantiated query $q_\sigma$ w.r.t. a database db.**

$\mathtt{Reformulate}^0(q, \mathtt{db}) = \{q(x,y)\text{:- } x\ \tau\ y\}$
$\mathtt{Reformulate}^1(q, \mathtt{db}) = \mathtt{Reformulate}^0(q, \mathtt{db}) \cup$
$\{q(x, \text{confP})\text{:- } x\ \tau\ \text{confP},$
$q(x, \text{posterCP})\text{:- } x\ \tau\ \text{posterCP},\ q(x, _{:}b_0)\text{:- } x\ \tau\ _{:}b_0,$
$q(x, \text{paper})\text{:- } x\ \tau\ \text{paper}$
$q(x, \text{conference})\text{:- } x\ \tau\ \text{conference}\}$
$\mathtt{Reformulate}^2(q, \mathtt{db}) = \mathtt{Reformulate}^1(q, \mathtt{db}) \cup$
$\{q(x, \text{confP})\text{:- } x\ \tau\ \text{posterCP},\ q(x, \text{confP})\text{:- } x\ \tau\ _{:}b_0,$
$q(x, \text{confP})\text{:- } x\ \text{inProceedingsOf}\ z,$
$q(x, \text{paper})\text{:- } x\ \tau\ \text{confP},\ q(x, \text{paper})\text{:- } x\ \text{hasTitle}\ z,$
$q(x, \text{paper})\text{:- } x\ \text{hasAuthor}\ z,$
$q(x, \text{conference})\text{:- } z\ \text{inProceedingsOf}\ x$
$q(x, \text{conference})\text{:- } x\ \text{hasName}\ z\}$
$\mathtt{Reformulate}^3(q, \mathtt{db}) = \mathtt{Reformulate}^2(q, \mathtt{db}) \cup$
$\{q(x, \text{paper})\text{:- } x\ \tau\ \text{posterCP},\ q(x, \text{paper})\text{:- } x\ \tau\ _{:}b_0,$
$q(x, \text{paper})\text{:- } x\ \text{inProceedingsOf}\ z,$
$q(x, \text{paper})\text{:- } x\ \text{hasContactA}\ z\}$
$\mathtt{Reformulate}^4(q, \mathtt{db}) = \mathtt{Reformulate}^3(q, \mathtt{db})$

**Figure 13: Sample reformulation of a query $q$ w.r.t. the database of our running example.**

a variable in the property position, appears in $q_\sigma$, then create the new query $q_{\sigma\cup\nu}$, which binds $y$ to the built-in property $\tau$. Observe that if $y$ was a distinguished variable in $q_\sigma$, a head variable in $q_{\sigma\cup\nu}$ will be bound after the rule application. Now consider rule (6) on some query $q_\sigma$. If $q_\sigma$ contains a triple of the same form $\mathtt{s}\ y\ \mathtt{o}$, and the database db contains a triple with any $\mathbf{p}$ in the property position, the rule creates the new query $q_{\sigma\cup\nu}$ where $y$ is bound to $\mathbf{p}$. Rules (7) and (8) instantiate query variables appearing in the property position, to values appearing in a $\prec_{sp}$ statement of db. The intuition is that both the subject and the object of a $\prec_{sp}$ statements are properties, therefore they can be used to instantiate the property variable $y$.

Rules (9)–(13) instantiate the variable $z$ in a query triple of the form $\mathtt{s}\ \tau\ z$. The RDF meta-model specifies that the values of the $\tau$ property are classes. Therefore, the rules bind $z$ to db values of which it can be inferred that they are classes, i.e., those appearing in specific positions in schema-level triples. For instance, if $\mathbf{s}_1\ \tau\ \mathbf{c} \in \mathtt{db}$, then $\mathbf{c}$ is a class and $z$ in rule (9) can be instantiated to $\mathbf{c}$. Similarly, the subject and object of a $\prec_{sc}$ statements are used in rules (10) and (11). Finally, Rules (14)–(17) use schema triples to replace (denoted *old triple / new triple*) a triple in the input query with a new triple. Rule (14) exploits $\prec_{sc}$ statements: if the query $q_\sigma$ asks for instances of class $\mathbf{c}_2$ and $\mathbf{c}_1$ is a subclass of $\mathbf{c}_2$, then instances of $\mathbf{c}_1$ should also be returned, and this is what the output query of this rule does. The last three rules are similar.

**EXAMPLE 8 (CONTINUED).** *Consider the previously introduced database* db *and query $q(x,y,z)$:- $x\ y\ z$ asking for the triples of* db *(including the entailed ones). We show how some of the above rules can be used to reformulate $q$ w.r.t.* db.

- $(i)$ *Using $q$ as input for rule (5) produces the query: $q_{\{y\to\tau\}}$, i.e., $q(x,\tau,z)$:- $x\ \tau\ z$.*
- $(ii)$ *Using $q_{\{y\to\tau\}}$ as input for rule (11) can lead to: $q_{\{y\to\tau, z\to\text{confP}\}}$, i.e., $q(x,\tau,\text{confP})$:- $x\ \tau$ confP.*
- $(iii)$ *Finally, using $q_{\{y\to\tau, z\to\text{confP}\}}$ as input for rule (14) can lead to: $q(x,\tau,\text{confP})$:- $x\ \tau\ _{:}b_0$.*

**Query reformulation algorithm** For a query $q$ and a database db, the output of $\mathtt{Reformulate}(q, \mathtt{db})$ is defined as the fixpoint $\mathtt{Reformulate}^\infty(q, \mathtt{db})$, where:

$\mathtt{Reformulate}^0(q, \mathtt{db}) = \{q\}$
$\mathtt{Reformulate}^{k+1}(q, \mathtt{db}) = \mathtt{Reformulate}^k(q, \mathtt{db}) \cup$
$\{q''_{\sigma''} \mid \exists i \in [5, \ldots, 17]$ s.t. applying rule $(i)$ on db
and some query $q'_{\sigma'} \in \mathtt{Reformulate}^k(q, \mathtt{db})$
yields the query $q''_{\sigma''}\}$

Theorem 5 shows that our reformulation algorithm terminates and provides an upper bound for the size of its output.

THEOREM 5. *Given a BGP query $q$ and a database* db*, the size (number of queries) of the output of* $\mathtt{Reformulate}(q, \mathtt{db})$ *is in $O((6 * \#\mathtt{db}^2)^{\#q})$, with $\#$db and $\#q$ the sizes (number of triples) of* db *and $q$ respectively.*

In practice, the size of a reformulated query is much smaller than the theoretical upper bound, but it may still be in the hundreds, depending on the query and the schema-level triples. However, these queries have many atoms in common with each other, therefore important performance benefits can be achieved by evaluating each sub-expression common to several such queries, only once. In our experiments, the off-the-shelf Postgres optimizer was able to recognize such cases and handle them quite well (see Section 7).

Clearly, improving the conjunctive query processor's capability to recognize and factorize common sub-expressions may further speed up the evaluation of reformulated queries.

**EXAMPLE 9 (CONTINUED).** *The reformulation of the query $q(x,y)$:- $x\ \tau\ y$ w.r.t.* db*, asking for all resources and the classes to which they belong, is in Figure 13.*

## 6.2 From query reformulation to reformulation-based query answering

It turns out that by handing the result of reformulating a query as explained above, directly to a conjunctive query processor for evaluation, may introduce erroneous answers:

**EXAMPLE 10 (CONTINUED).** *Consider again the database* db *and the query* $q(x, y)$*:-* $x\ \tau\ y$*. The queries in* $\texttt{Reformulate}(q, \texttt{db})$ *are shown in Figure 13. Evaluating these queries, and in particular* $q(x, \text{confP})$*:-* $x\ \tau\ _{:}b_0$ *in* $\texttt{Reformulate}^2(q, \texttt{db})$*, with the assignment* $\mu = \{x \to \text{edbt2013}, _{:}b_0 \to \text{conference}\}$*, leads to the answer tuple* $\langle \text{edbt2013}, \text{confP} \rangle$*. Or, this tuple does not belong to the correct answer (presented in Example 7). Thus, the tuple is an erroneous answer.*

As the above example suggests, the issue is due to blank nodes. The semantics of blank nodes in BGP queries does not match the purpose for which they are brought into query reformulation by the Reformulate algorithm. Remember that the semantics of a blank node in a BGP query against an RDF graph or database is that of a non-distinguished variable. However, when our Reformulate algorithm brings a blank node in a query through a variable binding or a triple replacement, it refers precisely to *that particular blank node* in the database (as opposed to: any value which matches an existential variable). In the above example, during the reformulation of $q$, when we use the rule (14) to reformulate

$$q(x, \text{confP}) \text{:-}\ x\ \tau\ \text{confP}$$

using the schema-level triple:

$$_{:}b_0\ \prec_{sc}\ \text{confP} \in \texttt{db}$$

into

$$q(x, \text{confP}) \text{:-}\ x\ \tau\ _{:}b_0$$

the goal is indeed to find conference paper values for $x$ from the anonymous (blank-node) subclass $_{:}b_0$ of confP.

**Non-standard evaluation and answer set of a query against a database** To overcome the above issue, we introduce *alternate notions of evaluation and of answer set of a partially instantiated query* against a database. The difference between the standard definitions from Section 6.1 and the non-standard ones concerns blank nodes. Standard evaluation is based on binding $\texttt{VarBl}(q)$, all the query variables and blank nodes, to database values. In contrast, the non-standard definition only seeks bindings for the query variables; *blank nodes are left untouched*, just like URIs and literals.

Formally, given a database db whose set of values (URIs, blank nodes, literals) is $\texttt{Val}(\texttt{db})$ and a query $q_\sigma(\bar{x}_\sigma)$*:-* $(t_1, \ldots, t_\alpha)_\sigma$ whose set of variables (no blank nodes) is $\texttt{Var}(q_\sigma)$, the *non-standard evaluation* of $q_\sigma$ against db is defined as:

$$\tilde{q}_\sigma(\texttt{db}) = \{(\bar{x}_\sigma)_\mu \mid \mu : \texttt{Var}(q_\sigma) \to \texttt{Val}(\texttt{db}) \text{ is a total assignment s.t. } ((t_1, \ldots, t_\alpha)_\sigma)_\mu \subseteq \texttt{db}\}$$

The *non-standard answer set* of $q_\sigma$ against db is obtained by the non-standard evaluation of $q_\sigma$ against $\texttt{db}^\infty$, which using our notation is denoted $\tilde{q}_\sigma(\texttt{db}^\infty)$.

The next property shows how standard and non-standard definitions of query evaluation and answer set are related. It follows directly from the fact that the assignments $\mu$ involved in non-standard evaluations, defined on $\texttt{Var}(q)$ only, are a subset of those allowed in standard evaluations, defined on $\texttt{VarBl}(q)$, as non-standard evaluations implicitly assign any URI, blank node, or literal to itself.

PROPERTY 2. *Let* db *be a database and* $q_\sigma$ *a (partially instantiated) query against* db*.*

1. $\tilde{q}_\sigma(\texttt{db}) \subseteq q_\sigma(\texttt{db})$ *and* $\tilde{q}_\sigma(\texttt{db}^\infty) \subseteq q_\sigma(\texttt{db}^\infty)$ *hold.*
2. *If* $q_\sigma$ *does not contain blank nodes then* $\tilde{q}_\sigma(\texttt{db}) = q_\sigma(\texttt{db})$ *and* $\tilde{q}_\sigma(\texttt{db}^\infty) = q_\sigma(\texttt{db}^\infty)$*.*

With the above notion of non-standard evaluation in place, our reformulation-based query answering technique is:

THEOREM 6. *Given a BGP query* $q$ *without blank nodes and a database* db*, the following holds:*

$$q(\texttt{db}^\infty) = \bigcup_{q'_{\sigma'} \in \texttt{Reformulate}(q, \texttt{db})} \tilde{q}'_{\sigma'}(\texttt{db}).$$

Note that Theorem 6 considers queries without blank nodes. This assumption is made *without loss of generality*, since blank nodes *from the original query* can be immediately replaced with non-distinguished variables in BGP queries, without impacting the answer set in any way (as explained in Section 2.2). We only make the assumption in order to prevent confusion between the *original blank nodes* (i.e., non-distinguished variables) and those introduced by the reformulation steps, and which required the introduction of non-standard evaluation in order to avoid erroneous answers.

**Implementing (non-)standard evaluation** While our alternate definitions are *non-standard* from an RDF perspective, they are just as easy to implement using e.g. an RDBMS, as the "standard" definitions. For "standard" RDF evaluation of a conjunctive BGP query $q$, translate $q$ into SQL, taking care to replace each blank node with the respective relation attribute name; for "non-standard" evaluation, translate $q$ into SQL by enclosing the blank nodes within quotes, so that the RDBMS treats each as a constant, to be matched only by the exact same value in the database.

From a practical perspective, Theorem 6 states: to answer a query $q$ against a database db, it suffices to $(i)$ reformulate $q$ w.r.t. db and $(ii)$ evaluate each reformulated query on the *original* database db, using the *non-standard* evaluation. In other words, query reformulation (based on db) followed by non-standard evaluation of partially instantiated queries computes the exact answer set, and does not require saturating the database. Moreover (and importantly), these steps only require standard conjunctive query evaluation capabilities from the underlying system.

**EXAMPLE 11 (CONTINUED).** *Consider again the query* $q(x, y)$*:-* $x\ \tau\ y$*. The answer set of* $q$ *against the previous database* db *is:*

$$\bigcup_{q'_{\sigma'} \in \texttt{Reformulate}(q, \texttt{db})} \tilde{q}'_{\sigma'}(\texttt{db}) = \{\langle \text{doi}_1, _{:}b_0 \rangle, \langle \text{doi}_1, \text{confP} \rangle, \langle \text{edbt2013}, \text{conference} \rangle, \langle \text{doi}_1, \text{paper} \rangle, \langle _{:}b_2, \text{conference} \rangle\}.$$

*Note that this answer set coincides with the one obtained by saturation-based query answering in Example 7.*

# 7. EXPERIMENTAL EVALUATION

We now describe experiments on our query answering strategies within the DB fragment of RDF. We describe the settings in Section 7.1, before discussing our results on saturation in Section 7.2, query answering in Section 7.3 and updates in Section 7.4.

## 7.1 Settings

We implemented our Saturate and Reformulate algorithms in Java 1.6 and deployed them on top of PostgreSQL v8.5 (using standard default settings). *Instance-level* triples are stored in a $\texttt{Triple}(\texttt{s}, \texttt{p}, \texttt{o})$ table, the set-based saturation in a $\texttt{Sat}(\texttt{s}, \texttt{p}, \texttt{o})$ table, while the multiset-based saturation (required for incrementally maintaining the saturation) is efficiently stored, as explained

| Graph | #Schema | #Instance | #Saturation | Saturation increase (%) | #Multiset | Multiset increase (%) | $t_{sat}$ (s) | $t_{sat+}$ (s) |
|---|---|---|---|---|---|---|---|---|
| Barton [22] | 101 | 33,912,142 | 38,969,023 | 14.91 | 73,551,358 | 116.89 | 4,294 | 4,586 |
| DBpedia [23] | 5666 | 26,988,098 | 29,861,597 | 10.65 | 66,029,147 | 227.37 | 2,742 | 2,977 |
| DBLP [24] | 41 | 8,424,216 | 11,882,409 | 41.05 | 18,699,232 | 121.97 | 748 | 799 |

**Table 1: Graph characteristics and saturation times.**

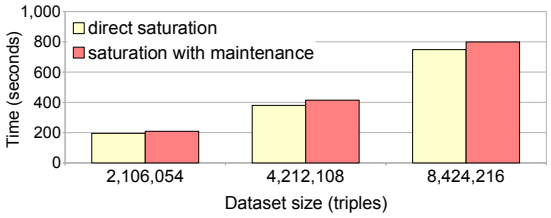



**Figure 14: Scalability of our saturation algorithms.**




**Figure 16: Update times on the DBLP graph.**

in Section 5.2, in a table $\texttt{SatM}(\texttt{s}, \texttt{p}, \texttt{o}, \texttt{isExplicit}, \texttt{count})$. Each table is indexed by all permutations of the (s, p, o) columns, leading to a total of 6 indexes; the spo index is clustering. We adopted this indexing choice (inspired by [25]) to give PostgreSQL efficient query evaluation opportunities. *Schema-level* triples are kept in memory. *All measured times are averaged over* 10 *executions.*

Previous works have used *dictionary encoding* when storing an RDF database: to avoid storing and joining string-encoded RDF attributes, a dictionary is built associating an integer to each distinct URI or blank node, queries are *encoded* by replacing constants with the respective integers, evaluated on the integer-encoded data, and their results *decoded* at the end of execution. We have tested our experiments with and without a dictionary; we present results *with* this dictionary encoding.

We compare our query answering techniques using the well-established Barton, DBpedia, and DBLP RDF data sets, whose main characteristics are summarized in Table 1.

## 7.2 Saturation

We denote $\mathbf{t_{sat}}$ the time to saturate a given database by Algorithm Saturate (Section 5.1), and $\mathbf{t_{sat+}}$ the time to saturate it by Algorithm $\texttt{Saturate}_+$ (Section 5.2).

As Table 1 shows, saturation added between 10% and 41% to the database size. Table 1 also shows $\mathbf{t_{sat}}$ and $\mathbf{t_{sat+}}$ for each graph. As expected, Saturate is (slightly) faster than $\texttt{Saturate}_+$. However, if the graph is updated, one can maintain the saturation only if $\texttt{Saturate}_+$ was used, as explained in Section 5.2. For space reasons, we only show here results on the DBLP [24] graph, whose instance contains blank nodes; more results can be found in [19].

**Saturation process scalability** We now study the scalability of our saturation algorithms, with and without the provisions needed for incremental maintenance of the saturation. Figure 14 shows the running times of our Saturate and $\texttt{Saturate}_+$ algorithms when saturating the $\frac{1}{4}$, $\frac{1}{2}$, and finally the whole DBLP instance-level data with the DBLP schema-level triples. The figure shows that the saturation time grows almost linearly as the data size increases, although its worst-case complexity is $O(\#\texttt{db}^3)$ (Theorem 2).

**Comparison with other saturation methods** Our Saturate algorithm is quite straightforward and its performance is comparable to others from the literature (modulo the specific set of rules used). Our incremental $\texttt{Saturate}_+$ algorithm, on the other hand, is novel and outperforms existing saturation-based query answering techniques, relying on saturation maintenance. These either scale poorly [21] or perform more costly maintenance operations [26]. Indeed, the maintenance method in [21], implemented in Sesame, uses a truth maintenance algorithm relying on managing the *justifications* (i.e., the proofs) for every entailed triple. Maintaining this set of justifications is problematic even for relatively small graphs (300.000 triples); maintenance after deletions is more costly than re-saturating the graph from scratch!

The maintenance method in [26], implemented in the well-known BigOWLIM commercial system, improves over the maintenance-upon-delete method of [21] by computing justifications only at maintenance time, and only for the triples which may be impacted, instead of systematically computing and storing all entailed triples justifications. Still, this computes and stores much more data than our integer derivation counts, while achieving the same goal of correctly maintaining the saturation when the database changes. Furthermore, our algorithm is not tailored for a specific system and can be plugged on top of any RDBMS.

## 7.3 Query answering times

We hand-picked a set of 26 queries for the DBLP graph, aiming at a variety of behavior when reformulated against the DBLP schema. The queries, detailed in [19], have between 1 and 10 triple patterns (6 on average). We obtained similar query answering times on the two saturation tables, Sat and SatM, therefore from this point further only the SatM results will be discussed.

For a query $q$, we denote by $\mathbf{t^{sat}}(q)$ the time to answer $q$ against the already saturated database SatM, and $\mathbf{t^{ref}}(q)$ the time to answer $q$ by reformulating $q$ and the (non-standard) evaluation of its reformulation against the Triple table.

The graph in Figure 15 shows the query answer times, grouped in the decreasing order of $\mathbf{t^{ref}}$ and divided in four groups by value of $\mathbf{t^{ref}}$ compared with the thresholds: 100s, 10s, and 1s. The first group contains two extreme-case queries: $Q_{01}$ returns the whole saturation of the database ($11 \times 10^6$ tuples), while $Q_{02}$ returns all (DBLP publication, attribute) pairs ($5 \times 10^6$ tuples). The second group contains mostly general queries using upper-level classes or properties of the schema (entailed triples, thus RDF reasoning, strongly contribute to answering such queries). Finally, the third and fourth groups contains mostly specific queries using lower-level classes or properties of the schema (thus, these queries' results are less impacted by reasoning).

Figure 15 shows, for each query: the number of union terms in the reformulated query (in parentheses after the query name); the time $\mathbf{t^{sat}}(q)$; the time $\mathbf{t^{ref}}(q)$; the sum $\mathbf{t^{sat}}(q) + \mathbf{t_{sat+}}$. As expected, $\mathbf{t^{sat}}(q)$ is significantly smaller than $\mathbf{t^{ref}}(q)$ for queries with large reformulations. However, if one factors in the saturation time $\mathbf{t_{sat+}}$, the saturation-based approach becomes expensive. Obviously, saturation costs are paid only once, not for each query; we

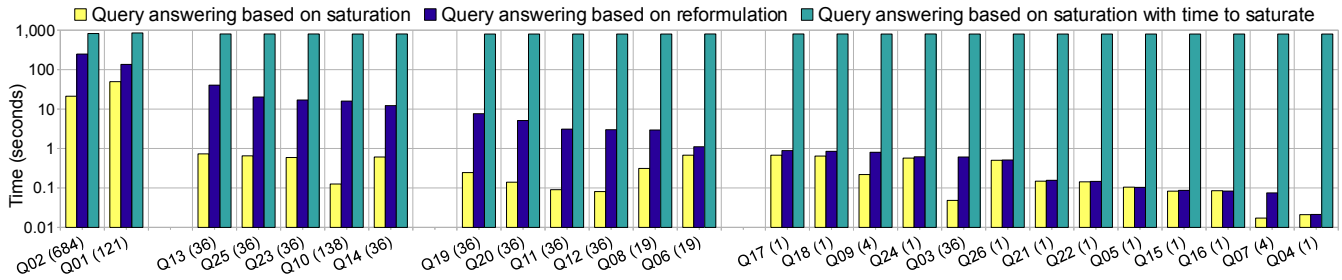



**Figure 15: Query answering times on the DBLP graph.**

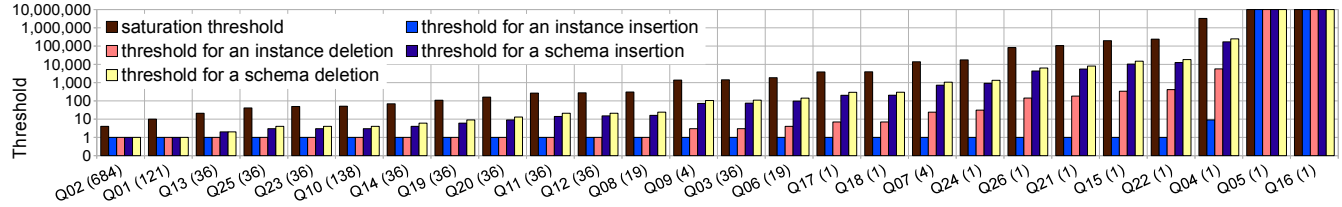



**Figure 17: Saturation thresholds for DBLP queries.**

deepen this analysis below when discussing thresholds. Inspecting the results, we also found small-result queries have small $\mathbf{t}^{\mathbf{sat}}$ and $\mathbf{t}^{\mathbf{ref}}$, an encouraging sign that PostgreSQL's optimizer handled correctly both the original and the reformulated queries.

## 7.4 Comparison of our query answering strategies in the presence of updates

We now compare saturation and reformulation upon graph updates. Updates have no impact on reformulation, but saturation needs to maintain `SatM`. To measure this overhead, we performed updates of one triple on the data and the schema (detailed in [19]). The graph in Figure 16 shows: the average time to insert into (resp. delete from) the `Triple` table; the average time to insert into (resp. delete from) *and maintain* the `SatM` table; the average time to *maintain* the `SatM` table after an insertion (resp. deletion) made on the schema. We see that handling `SatM` is slower than updating `Triple`, by two orders of magnitude for instance deletions. This shows that while the algorithm $\texttt{Saturate}_+$ and the `SatM` table are required in order to avoid saturating from scratch, saturation maintenance may get costly due to the recursive nature of entailment. In particular, in the case of schema updates, maintaining the saturation may sometimes be more costly than re-saturating.

We now study *when saturation pays off* over multiple query runs. We call the *saturation threshold* of a query $q$, or $\mathbf{st}(q)$, the smallest integer $n$ such that:

$$n \times \mathbf{t}^{\mathbf{ref}}(q) > n \times \mathbf{t}^{\mathbf{sat}}(q) + \mathbf{t}_{\mathbf{sat}+}$$

In other words, $n$ is the minimum number of times one needs to run $q$ in order for the whole saturation cost to amortize.

Similarly, we study how many times $q$ should run in order for the *maintenance overhead due to one instance or schema update* to pay off. We formalize this as follows.

Concerning instance updates, let $\mathbf{t}^+_{\texttt{Triple}}$ be the time to insert one statement in `Triple`, and $\mathbf{t}^+_{\texttt{SatM}}$ be the time to propagate the insertion of one triple to the `SatM` relation. Then, the *saturation threshold for an instance insertion*, denoted $\mathbf{st}^+_{\mathbf{i}}(q)$, is the smallest $n$ for which:

$$n \times \mathbf{t}^{\mathbf{ref}}(q) + \mathbf{t}^+_{\texttt{Triple}} > n \times \mathbf{t}^{\mathbf{sat}}(q) + \mathbf{t}^+_{\texttt{SatM}}$$

In other words, $\mathbf{st}^+_{\mathbf{i}}(q)$ is the minimum number of times one needs to run $q$ in order for the maintenance overhead due to the insertion of one triple (recall Figure 16) to amortize. We similarly define the *saturation threshold for an instance deletion*, denoted $\mathbf{st}^-_{\mathbf{i}}(q)$.

Schema updates do not affect the `Triple` table, since the schema is kept in memory, but they can have a big impact on `SatM`. Similar to $\mathbf{st}(q)$, we define the *saturation threshold for a schema insertion* $\mathbf{st}^+_{\mathbf{s}}(q)$ and *deletion* $\mathbf{st}^-_{\mathbf{s}}(q)$, as the minimum number of times one needs to run $q$ in order for the schema update cost to amortize.

Figure 17 shows the 5 saturation thresholds for our queries. The vertical (log-scale) axis is limited to $10^7$, for readability. The thresholds follow a similar trend, strongly determined by the size of the reformulated query (shown in parentheses on the $x$ axis). The larger the reformulated query, the lower the threshold: saturation pays off faster when reformulation is expensive, and this tends to happen when the queries are syntactically complex.

Concerning $\mathbf{st}(q)$, we see that it varies from 2 for $Q_{02}$ to more than $10^5$ for $Q_{05}$ or $Q_{22}$; for queries such as $Q_{04}$, $Q_{05}$ etc., which are left unchanged by reformulation, the saturation cost can only be compensated after $10^6 - 10^7$ runs. This shows that saturation is not always the most efficient way to go. While it is true saturation can be performed off-line, one needs to also keep in mind that saturation may require quite complex maintenance algorithms.

Comparing the thresholds among themselves, we notice that $\mathbf{st}$ is always higher than the update thresholds, which is expected since $\mathbf{st}$ runs need to offset *the complete saturation cost*, whereas $\mathbf{st}^+_{\mathbf{i}}$, $\mathbf{st}^-_{\mathbf{i}}$, $\mathbf{st}^+_{\mathbf{s}}$ and $\mathbf{st}^-_{\mathbf{s}}$ need to offset the cost of maintaining saturation for just one triple added or deleted. Finally, $\mathbf{st}^+_{\mathbf{i}}$ is lower than $\mathbf{st}^-_{\mathbf{i}}$, and $\mathbf{st}^+_{\mathbf{s}}$ is lower than $\mathbf{st}^-_{\mathbf{s}}$, meaning that saturation costs particularly penalize scenarios where deletions are frequent.

## 7.5 Experiment conclusions

Our experiments showed that `Saturate` and `Reformulate` can be used to *process BGP queries* efficiently by exploiting an off-the-shelf RDBMS. However, they perform very differently depending on the query selectivity and the impact of the schema through reasoning: saturation is best for large-reformulation queries, while reformulation is efficient for small-to-moderate reformulation.

With respect to *updates*, we showed that saturation can be maintained at a reasonable cost for instance-level updates, while schema-level updates are much more expensive. Updates, however, have a small impact on reformulation making it appropriate for high update rates. When considering also *repeated query runs*, we highlighted a number of thresholds determining when saturation pays off; these thresholds are strongly impacted by the query reformulation size and selectivity. While saturation is the default in many

RDF platforms, our experiments demonstrate the practical interest of *reformulation-based BGP query answering*.

# 8. RELATED WORK

Many well-known SPARQL compliant RDF platforms, e.g., 3store [27], Jena [28], OWLIM [29], Sesame [30], or Virtuoso [31], or research prototypes, e.g., Hexastore [5] or RDF-3X [4], either $(i)$ ignore entailed triples or $(ii)$ provide saturation-based query answering, based on (a subset of) RDF entailment rules.

The drawbacks of saturation w.r.t. updates have been pointed out in [21], which proposes a *truth maintenance* technique implemented in Sesame. It relies on the storage and management of the *justifications* of entailed triples (which triples beget them). While efficient on graphs with few entailed triples, the technique is pegged by the high overhead of handling justifications when their number and size grow. Therefore, [26] proposes to compute only the relevant justifications w.r.t. an update, at maintenance time. This technique is implemented in OWLIM, however [29] points out that schema-level deletions can lead to poor performance. In contrast, our saturation maintenance technique (Section 5.2) is based on the *number of times* triples are entailed; this is easier to store and manipulate. Our technique performs well for instance updates, and acceptably on schema updates. On average, it is worth maintaining the saturation, though in some (rare) cases, when the updates affect upper-level classes or properties of the schema, it may be more costly than re-saturating (up to five times in our experiments). A distinct yet related problem is finding which triples to delete from an RDF graph, so that an implicit triple no longer holds [32].

Reformulation-based query answering has been investigated in RDF fragments ranging from the Description Logic (DL) [6] one [7, 8, 9], i.e., modeling simple DL knowledge bases, to a slight extension thereof allowing values to be used both as constants and classes/properties [10, 11, 12, 13]. These two fragments of RDF impose restrictions on triples (no blank nodes) and on entailment (only the RDFS entailment rules are considered). Our DB fragment is *strictly more expressive* since it supports blank nodes.

Reformulation-based query answering in the DL fragment of RDF has been investigated for relational conjunctive queries [7, 8, 9], while the slight extension thereof considered in [10, 11, 12, 13] has been investigated for one-triple BGP queries [12, 13], BGP queries [11], and SPARQL queries [10]. Relational conjunctive queries are *strictly less expressive* than BGP queries, since the former only allow triples of the form $\mathtt{s}\ \tau\ c$ and $\mathtt{s}\ p\ \mathtt{o}$, ruling out the possibility to put variables in place of classes or properties.

The query reformulation algorithms of [7, 11] are restrictions of our `Reformulate`. The same holds for the algorithms in [8, 9] when restricted to the DL fragment of RDF, whereas they are capable of handling complex DLs. Algorithms in [7, 8, 9] consider only our rules (14)–(17) to reformulate relational conjunctive queries, while the algorithm in [11] needs two additional rules for BGP queries. These two rules actually correspond to our rules (5)–(13), *under the simplifying assumption* that part of the information needed for reformulation have been pre-computed. In [12, 13], atomic BGP queries are reformulated using a standard backward-chaining algorithm [33] on first order encodings of the entailment rules dedicated to RDFS. In [10], SPARQL queries are reformulated into *nested* SPARQL, i.e., an extension of SPARQL in which properties in triples can be nested regular expressions. While such nested reformulated queries are more compact, the queries we produce are more practical, since their evaluation can be directly delegated to *any* off-the-shelf RDBMS, or to an RDF engine such as RDF-3X [4] even if it is unaware of reasoning.

# 9. CONCLUSION

In this paper, we extended the state of the art in BGP query answering against RDF graphs with updates. We devised novel saturation- and reformulation-based query answering techniques robust to instance and schema updates, for an RDF fragment extending those known in the literature; we compared thoroughly their performance and identified the factors impacting the comparison. Notably, our techniques can be directly deployed on top of any off-the-shelf RDBMS. An automated strategy to choose between the two techniques is part of our future work.

# 10. REFERENCES


[1] "Resource description framework," http://www.w3.org/RDF.

[2] W3C, "SPARQL protocol and RDF query language," http://www.w3.org/TR/rdf-sparql-query.

[3] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *VLDB*, 2007.

[4] T. Neumann and G. Weikum, "x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases," *PVLDB*, vol. 3, no. 1, 2010.

[5] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for Semantic Web data management," *PVLDB*, vol. 1, no. 1, 2008.

[6] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, Eds., *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[7] P. Adjiman, F. Goasdoué, and M.-C. Rousset, "SomeRDFS in the semantic web," *JODS*, vol. 8, 2007.

[8] D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, "Tractable reasoning and efficient query answering in description logics: The DL-Lite family," *Journal of Automated Reasoning (JAR)*, vol. 39, no. 3, 2007.

[9] G. Gottlob, G. Orsi, and A. Pieris, "Ontological queries: Rewriting and optimization," in *ICDE*, 2011, keynote.

[10] M. Arenas, C. Gutierrez, and J. Pérez, "Foundations of RDF databases," in *Reasoning Web*, 2009.

[11] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu, "View selection in semantic web databases," *PVLDB*, 2011.

[12] Z. Kaoudi, I. Miliaraki, and M. Koubarakis, "RDFS reasoning and query answering on DHTs," in *ISWC*, 2008.

[13] J. Urbani, F. van Harmelen, S. Schlobach, and H. Bal, "QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases," in *ISWC*, 2011.

[14] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[15] T. Imielinski and W. L. Jr., "Incomplete information in relational databases," *JACM*, vol. 31, no. 4, 1984.

[16] T. Neumann and G. Weikum, "Scalable join processing on very large RDF graphs," in *SIGMOD*, 2009.

[17] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold, "Column-store support for RDF data management: not all swans are white," vol. 1, no. 2, 2008.

[18] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *PVLDB*, vol. 1, no. 1, 2008.

[19] "Extended version (Inria report no. 8018)," http://hal.inria.fr/hal-00719641/, 2012.

[20] A. Gupta and I. S. M. (editors), *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.

[21] J. Broekstra and A. Kampman, "Inferencing and truth maintenance in RDF Schema: Exploring a naive practical approach," in *PSSS Workshop*, 2003.
[22] "Barton," http://simile.mit.edu/rdf-test-data/barton.
[23] "DBpedia 3.7," http://wiki.dbpedia.org/Downloads37.
[24] http://kdl.cs.umass.edu/data/dblp/dblp-info.html.
[25] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *VLDB J.*, 2010.
[26] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov, "OWLIM: A family of scalable semantic repositories," *Semantic Web*, vol. 2, no. 1, 2011.
[27] "3store," http://www.aktors.org/technologies/3store.
[28] "Jena," http://jena.sourceforge.net.
[29] "Owlim," http://owlim.ontotext.com.
[30] "Sesame," http://www.openrdf.org.
[31] "Virtuoso," http://virtuoso.openlinksw.com.
[32] C. Gutierrez, C. A. Hurtado, and A. A. Vaisman, "RDFS update: From theory to practice," in *ESWC*, 2011.
[33] S. J. Russell and P. Norvig, *Artificial Intelligence - A Modern Approach*. Pearson Education, 2010.