Learning Commonalities in RDF

Sara El Hassad, François Goasdoué(✉), and Hélène Jaudoin

IRISA, Univ. Rennes 1, Lannion, France
{sara.el-hassad,fg,helene.jaudoin}@irisa.fr

Abstract. Finding the commonalities between descriptions of data or knowledge is a foundational reasoning problem of Machine Learning introduced in the 70's, which amounts to computing a *least general generalization (*`lgg`*)* of such descriptions. It has also started receiving consideration in Knowledge Representation from the 90's, and recently in the Semantic Web field. We revisit this problem in the popular Resource Description Framework (RDF) of W3C, where descriptions are RDF graphs, i.e., a mix of data *and* knowledge. Notably, and in contrast to the literature, our solution to this problem holds for the *entire* RDF standard, i.e., we do not restrict RDF graphs in any way (neither their structure nor their semantics based on RDF entailment, i.e., inference) and, further, our algorithms can compute `lgg`s of *small-to-huge* RDF graphs.

Keywords: RDF · RDFS · RDF entailment · Least general generalization

1 Introduction

Finding the commonalities between descriptions of data or knowledge is a foundational reasoning problem of Machine Learning, which was formalized in the early 70's as computing a *least general generalization* (`lgg`) of such descriptions [25,26]. Since the early 90's, this problem has also received consideration in the Knowledge Representation field, where least general generalizations were rebaptized *least common subsumers* [12], in Description Logics, e.g., [9,12,19,33] and in Conceptual Graphs [11].

In this paper, we revisit this old reasoning problem, from both the theoretical and the algorithmic viewpoints, in the *Resource Description Framework (RDF)*: the prominent Semantic Web data model by W3C. In this setting, the problem amounts to computing the `lgg`s of RDF graphs, i.e., a mix of data *and* knowledge, the semantics of which is defined through RDF entailment, i.e., inference using entailment rules from the RDF standard.

To the best of our knowledge, the only proposal in that direction is the recent work [13,14], which brings a limited solution to the problem. It allows finding the commonalities between *single entities* extracted from RDF graphs (e.g., users in a social network), *ignoring* RDF entailment. In contrast, we further aim at considering the problem in all its generality, i.e., finding the commonalities between *general* RDF graphs, hence modeling *multiple interrelated entities* (e.g., social networks of users), accurately w.r.t. their standard semantics.

E. Blomqvist et al. (Eds.): ESWC 2017, Part I, LNCS 10249, pp. 502–517, 2017.
DOI: 10.1007/978-3-319-58068-5_31

More precisely, we bring the following contributions:

1. We define and study the problem of computing an `lgg` of RDF graphs in the *entire RDF standard*: we do not restrict RDF graphs in any way, i.e., neither their structure nor their semantics defined upon RDF entailment.
2. We provide three algorithms for our solution to this problem, which allow computing `lgg`s of *small-to-huge general RDF graphs* (i.e., that fit either in memory, in data management systems or in MapReduce clusters) w.r.t. *any set of entailment rules* from the RDF standard.

The paper is organized as follows. First, we introduce the RDF data model in Sect. 2. Then, in Sect. 3, we define and study the problem of computing an `lgg` of RDF graphs, for which we provide algorithms in Sect. 4. Finally, we discuss related work and conclude in Sect. 5.

Proofs of our technical results are available in the online research report [16].

2 The Resource Description Framework (RDF)

RDF Graphs. The RDF data model allows specifying *RDF graphs*. An RDF graph is a set of *triples* of the form $(\mathtt{s}, \mathtt{p}, \mathtt{o})$. A triple states that its *subject* `s` has the *property* `p`, the value of which is the *object* `o`. Triples are built using three pairwise disjoint sets: a set $\mathcal{U}$ of *uniform resources identifiers (URIs)*, a set $\mathcal{L}$ of *literals* (constants), and a set $\mathcal{B}$ of *blank nodes* allowing to support *incomplete information*. Blank nodes are identifiers for missing values in an RDF graph (unknown URIs or literals). *Well-formed triples*, as per the RDF specification [31], belong to $(\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{B})$; we only consider such triples hereafter.

Notations. We use `s`, `p`, `o` in triples as placeholders. We note $\mathtt{Val}(\mathcal{G})$ the set of *values* occurring in an RDF graph $\mathcal{G}$, i.e., the URIs, literals and blank nodes; we note $\mathtt{Bl}(\mathcal{G})$ the set of blank nodes occurring in $\mathcal{G}$. A blank node is written b possibly with a subscript, and a literal is a string between quotes. For instance, the triples $(b, \text{hasTitle}, \text{“LGG in RDF”})$ and $(b, \text{hasContactAuthor}, b_1)$ state that *something (b) entitled “LGG in RDF” has somebody (b_1) as contact author*.

A triple models an assertion, either for a *class* (unary relation) or for a *property* (binary relation). Table 1 (top) shows the use of triples to state such assertions. The RDF standard [31] provides built-in classes and properties, as URIs within the `rdf` and `rdfs` pre-defined namespaces, e.g., rdf:type which can be used to state that the above b is a conference paper with the triple $(b, \text{rdf:type}, \text{ConfPaper})$.

Adding Ontological Knowledge to RDF Graphs. An essential feature of RDF is the possibility to enhance the descriptions in RDF graphs by declaring *ontological constraints* between the classes and properties they use. This is achieved with *RDF Schema (RDFS)* statements, which are triples using particular built-in properties. Table 1 (bottom) lists the allowed constraints and the triples to state them; *domain* and *range* denote respectively the first and second attribute of every property. For example, the

triple (ConfPaper, rdfs:subClassOf, Publication) states that *conference papers are publications*, the triple (hasContactAuthor, rdfs:subPropertyOf, hasAuthor) states that *having a contact author is having an author*, the triple (hasAuthor, rdfs:domain, Publication) states that *only publications may have authors*, and the triple (hasAuthor, rdfs:range, Researcher) states that *only researchers may be authors of something.*

Notations. For conciseness, we use the following shorthands for built-in properties: τ for rdf:type, $\prec_{\text{sc}}$ for rdfs:subClassOf, $\prec_{\text{sp}}$ for rdfs:subPropertyOf, $\hookleftarrow_d$ for rdfs:domain, and $\hookrightarrow_r$ for rdfs:range.

Figure 1 displays the usual representation of the RDF graph $\mathcal{G}$ made of the seven above-mentioned triples, which are called the *explicit triples* of $\mathcal{G}$. A triple $(\mathtt{s}, \mathtt{p}, \mathtt{o})$ corresponds to the $\mathtt{p}$-labeled directed edge from the $\mathtt{s}$ node to the $\mathtt{o}$ node. Explicit triples are shown as solid edges, while the *implicit ones*, which are derived using ontological constraints (see below), are shown as dashed edges.

Importantly, it is worth noticing the deductive nature of ontological constraints, which begets implicit triples within an RDF graph. For instance, in Fig. 1, the constraint $(\text{hasContactAuthor}, \prec_{\text{sp}}, \text{hasAuthor})$ together with the triple $(b, \text{hasContactAuthor}, b_1)$ implies the implicit triple $(b, \text{hasAuthor}, b_1)$, which, further, with the constraint $(\text{hasAuthor}, \hookrightarrow_r, \text{Researcher})$ yields another implicit triple $(b_1, \tau, \text{Researcher})$.

Deriving the Implicit Triples of an RDF Graph. The RDF standard defines a set of *entailment rules* in order to derive automatically *all* the triples that are implicit to an RDF graph. Table 2 shows the strict subset of these rules that we will use to illustrate important notions as well as our contributions in the next sections; importantly, our contributions hold for the whole set of entailment rules of the RDF standard, and any subset of thereof. The rules in Table 2 concern the derivation of implicit triples using ontological constraints (i.e., *RDFS statements*). They encode the *propagation* of assertions through constraints (`rdfs2`, `rdfs3`, `rdfs7`, `rdfs9`), the *transitivity* of the $\prec_{\text{sp}}$ and $\prec_{\text{sc}}$ constraints (`rdfs5`, `rdfs11`), the *complementation* of domains or ranges through $\prec_{\text{sc}}$ (`ext1`, `ext2`), and the *inheritance* of domains and of ranges through $\prec_{\text{sp}}$ (`ext3`, `ext4`).

Table 1. RDF & RDFS statements.

RDF statement	Triple
Class assertion	$(\mathtt{s}, \text{rdf:type}, \mathtt{o})$
Property assertion	$(\mathtt{s}, \mathtt{p}, \mathtt{o})$ with $\mathtt{p} \neq \text{rdf:type}$

RDFS statement	Triple
Subclass	$(\mathtt{s}, \text{rdfs:subClassOf}, \mathtt{o})$
Subproperty	$(\mathtt{s}, \text{rdfs:subPropertyOf}, \mathtt{o})$
Domain typing	$(\mathtt{s}, \text{rdfs:domain}, \mathtt{o})$
Range typing	$(\mathtt{s}, \text{rdfs:range}, \mathtt{o})$

Table 2. Sample RDF entailment rules.

Rule [32]	Entailment rule
`rdfs2`	$(\mathtt{p}, \hookleftarrow_d, \mathtt{o}), (\mathtt{s}_1, \mathtt{p}, \mathtt{o}_1) \rightarrow (\mathtt{s}_1, \tau, \mathtt{o})$
`rdfs3`	$(\mathtt{p}, \hookrightarrow_r, \mathtt{o}), (\mathtt{s}_1, \mathtt{p}, \mathtt{o}_1) \rightarrow (\mathtt{o}_1, \tau, \mathtt{o})$
`rdfs5`	$(\mathtt{p}_1, \prec_{\text{sp}}, \mathtt{p}_2), (\mathtt{p}_2, \prec_{\text{sp}}, \mathtt{p}_3) \rightarrow (\mathtt{p}_1, \prec_{\text{sp}}, \mathtt{p}_3)$
`rdfs7`	$(\mathtt{p}_1, \prec_{\text{sp}}, \mathtt{p}_2), (\mathtt{s}, \mathtt{p}_1, \mathtt{o}) \rightarrow (\mathtt{s}, \mathtt{p}_2, \mathtt{o})$
`rdfs9`	$(\mathtt{s}, \prec_{\text{sc}}, \mathtt{o}), (\mathtt{s}_1, \tau, \mathtt{s}) \rightarrow (\mathtt{s}_1, \tau, \mathtt{o})$
`rdfs11`	$(\mathtt{s}, \prec_{\text{sc}}, \mathtt{o}), (\mathtt{o}, \prec_{\text{sc}}, \mathtt{o}_1) \rightarrow (\mathtt{s}, \prec_{\text{sc}}, \mathtt{o}_1)$
`ext1`	$(\mathtt{p}, \hookleftarrow_d, \mathtt{o}), (\mathtt{o}, \prec_{\text{sc}}, \mathtt{o}_1) \rightarrow (\mathtt{p}, \hookleftarrow_d, \mathtt{o}_1)$
`ext2`	$(\mathtt{p}, \hookrightarrow_r, \mathtt{o}), (\mathtt{o}, \prec_{\text{sc}}, \mathtt{o}_1) \rightarrow (\mathtt{p}, \hookrightarrow_r, \mathtt{o}_1)$
`ext3`	$(\mathtt{p}, \prec_{\text{sp}}, \mathtt{p}_1), (\mathtt{p}_1, \hookleftarrow_d, \mathtt{o}) \rightarrow (\mathtt{p}, \hookleftarrow_d, \mathtt{o})$
`ext4`	$(\mathtt{p}, \prec_{\text{sp}}, \mathtt{p}_1), (\mathtt{p}_1, \hookrightarrow_r, \mathtt{o}) \rightarrow (\mathtt{p}, \hookrightarrow_r, \mathtt{o})$

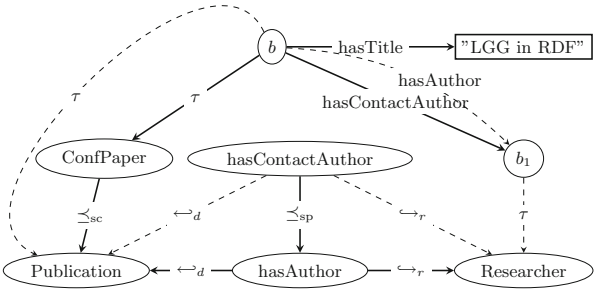

Fig. 1. Sample RDF graph $\mathcal{G}$.

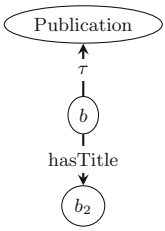

Fig. 2. Sample RDF graph $\mathcal{G}'$.

The *saturation (a.k.a. closure)* of an RDF graph $\mathcal{G}$ w.r.t. a set $\mathcal{R}$ of RDF entailment rules, is the RDF graph $\mathcal{G}^\infty$ obtained by adding to $\mathcal{G}$ all the implicit triples that can be derived from $\mathcal{G}$ using $\mathcal{R}$. Roughly speaking, the saturation $\mathcal{G}^\infty$ *materializes* the semantics of $\mathcal{G}$. It corresponds to the fixpoint obtained by repeatedly applying the rules in $\mathcal{R}$ to $\mathcal{G}$ in a forward-chaining fashion. In RDF, the saturation is *always* finite and unique (up to blank node renaming), and does not contain implicit triples [31,32].

The saturation of the RDF graph $\mathcal{G}$ shown in Fig. 1 corresponds to the RDF graph $\mathcal{G}^\infty$ in which all the $\mathcal{G}$ implicit triples have been made explicit. It is worth noting how, starting from $\mathcal{G}$, applying RDF entailment rules *mechanizes* the construction of $\mathcal{G}^\infty$. For instance, recall the reasoning sketched above for deriving the triple $(b_1, \tau, \text{Researcher})$. This is automated by the following sequence of applications of RDF entailment rules: $(\text{hasContactAuthor}, \preceq_{\text{sp}}, \text{hasAuthor})$ and $(b, \text{hasContactAuthor}, b_1)$ trigger `rdfs7` that adds $(b, \text{hasAuthor}, b_1)$ to the RDF graph. In turn, this new triple together with $(\text{hasAuthor}, \hookrightarrow_r, \text{Researcher})$ triggers `rdfs3` that adds $(b_1, \tau, \text{Researcher})$.

Comparing RDF Graphs. The RDF standard defines a generalization/specialization relationship between two RDF graphs, called *entailment between graphs*. Roughly speaking, an RDF graph $\mathcal{G}$ is more specific than another RDF graph $\mathcal{G}'$, or equivalently $\mathcal{G}'$ is more general than $\mathcal{G}$, whenever there is an embedding of $\mathcal{G}'$ into the *saturation* of $\mathcal{G}$, i.e., the complete set of triples that $\mathcal{G}$ models.

More formally, given any subset $\mathcal{R}$ of RDF entailment rules, an RDF graph $\mathcal{G}$ *entails* an RDF graph $\mathcal{G}'$, denoted $\mathcal{G} \models_{\mathcal{R}} \mathcal{G}'$, iff there exists an homomorphism ϕ from $\texttt{Bl}(\mathcal{G}')$ to $\texttt{Val}(\mathcal{G}^\infty)$ such that $[\mathcal{G}']_\phi \subseteq \mathcal{G}^\infty$, where $[\mathcal{G}']_\phi$ is the RDF graph obtained from $\mathcal{G}'$ by replacing every blank node b by its image $\phi(b)$.

Figure 2 shows an RDF graph $\mathcal{G}'$ entailed by the RDF graph $\mathcal{G}$ in Fig. 1 w.r.t. the RDF entailment rules displayed in Table 2. In particular, $\mathcal{G} \models_{\mathcal{R}} \mathcal{G}'$ holds for the homomorphism ϕ such that: $\phi(b) = b$ and $\phi(b_2) =$ "LGG in RDF". By contrast, when $\mathcal{R}$ is empty, this is not the case (i.e., $\mathcal{G} \not\models_{\mathcal{R}} \mathcal{G}'$), as the dashed edges in $\mathcal{G}$ are not materialized by saturation, hence the $\mathcal{G}'$ triple $(b, \tau, \text{Publication})$ cannot have an image in $\mathcal{G}$ through some homomorphism.

Notations. When RDF entailment rules are disregarded, i.e., $\mathcal{R} = \emptyset$, we note the entailment relation $\models$ (i.e., without indicating the rule set at hand).

Importantly, some remarkable properties follow directly from the definition of entailment between two RDF graphs [31,32]:

1. $\mathcal{G}$ and $\mathcal{G}^\infty$ are equivalent, noted $\mathcal{G} \equiv_\mathcal{R} \mathcal{G}^\infty$, since clearly $\mathcal{G} \models_\mathcal{R} \mathcal{G}^\infty$ and $\mathcal{G}^\infty \models_\mathcal{R} \mathcal{G}$ hold,
2. $\mathcal{G} \models_\mathcal{R} \mathcal{G}'$ holds iff $\mathcal{G}^\infty \models \mathcal{G}'$ holds.

In particular, the second above property points out that checking $\mathcal{G} \models_\mathcal{R} \mathcal{G}'$ can be done in two steps: a reasoning step that computes the saturation $\mathcal{G}^\infty$ of $\mathcal{G}$, followed by a standard graph homomorphism step that checks if $\mathcal{G}^\infty \models \mathcal{G}'$ holds.

3 Finding Commonalities Between RDF Graphs

In Sect. 3.1, we define the largest set of commonalities between RDF graphs as a particular RDF graph representing their *least general generalization* (`lgg` for short). Then, we devise a technique for computing such an `lgg` in Sect. 3.2.

3.1 Defining the `lgg` of RDF Graphs

A *least general generalization* of n descriptions $d_1, \ldots, d_n$ is a most specific description d generalizing every $d_{1 \leq i \leq n}$ for some generalization/specialization relation between descriptions [25,26]. In RDF, we use RDF graphs as descriptions and entailment between RDF graphs as relation for generalization/specialization:

Definition 1 (`lgg` of RDF graphs). *Let $\mathcal{G}_1, \ldots, \mathcal{G}_n$ be RDF graphs and $\mathcal{R}$ a set of RDF entailment rules.*

- *A* generalization *of $\mathcal{G}_1, \ldots, \mathcal{G}_n$ is an RDF graph $\mathcal{G}_g$ such that $\mathcal{G}_i \models_\mathcal{R} \mathcal{G}_g$ holds for $1 \leq i \leq n$.*
- *A* least general generalization *(`lgg`) of $\mathcal{G}_1, \ldots, \mathcal{G}_n$ is a generalization $\mathcal{G}_{\texttt{lgg}}$ of $\mathcal{G}_1, \ldots, \mathcal{G}_n$ such that for any other generalization $\mathcal{G}_g$ of $\mathcal{G}_1, \ldots, \mathcal{G}_n$, $\mathcal{G}_{\texttt{lgg}} \models_\mathcal{R} \mathcal{G}_g$ holds.*

Importantly, in the RDF setting, the following holds:

Theorem 1. *An* `lgg` *of RDF graphs always exists; it is* unique *up to entailment.*

Intuitively, we can always construct a (possibly empty) RDF graph that is the `lgg` of RDF graphs, in particular *the cover graph of RDF graphs* devised in the next Sect. 3.2. Further, an `lgg` is unique up to entailment (since $\mathcal{G}_{\texttt{lgg}} \models_\mathcal{R} \mathcal{G}_g$ holds for any $\mathcal{G}_g$ in Definition 1): if it were that many `lgg`s exist, pairwise incomparable w.r.t. entailment, then their merge[1] would be a single strictly more specific `lgg`, a contradiction.

[1] The merge of RDF graphs is their union *after renaming* their blank nodes, so that these RDF graphs do not join on such values which are *local* to them (Sect. 2).

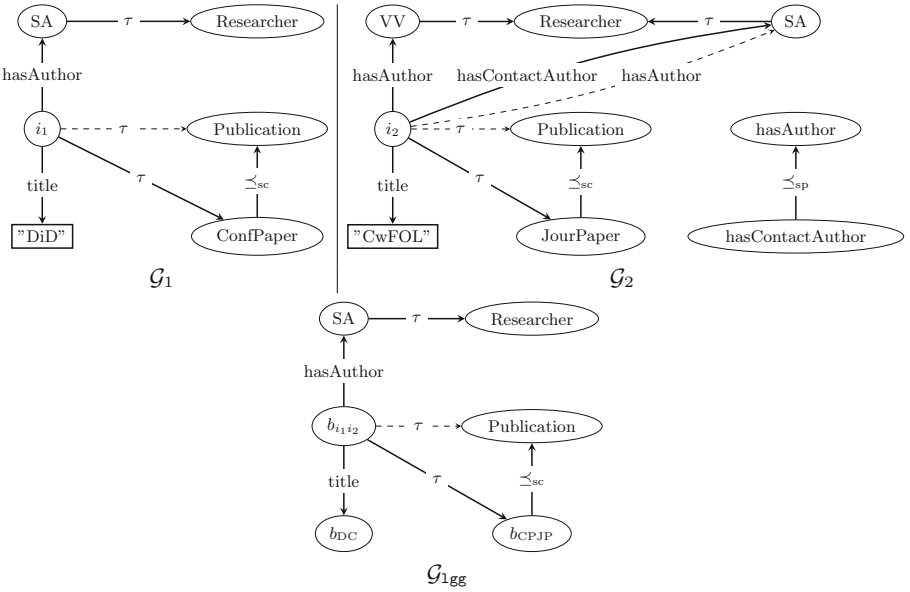

Fig. 3. Sample RDF graphs $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_{\texttt{lgg}}$, with $\mathcal{G}_{\texttt{lgg}}$ the minimal `lgg` of $\mathcal{G}_1$ and $\mathcal{G}_2$; their implicit triples (i.e., derived by the rules in Table 2) are shown as dashed edges.

Figure 3 displays two RDF graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, as well as their minimal `lgg` (with lowest number of triples) when we consider the RDF entailment rules shown in Table 2: $\mathcal{G}_{\texttt{lgg}}$. $\mathcal{G}_1$ describes a conference paper i_1 with title "Disaggregations in Databases" and author Serge Abiteboul, who is a researcher; also conference papers are publications. $\mathcal{G}_2$ describes a journal paper i_2 with title "Computing with First-Order Logic", contact author Serge Abiteboul and author Victor Vianu, who are researchers; moreover, journal papers are publications and having a contact author is having an author. $\mathcal{G}_{\texttt{lgg}}$ states that their common information comprises the existence of a resource ($b_{i_1 i_2}$) having some type ($b_{\mathrm{C(onf)P(aper)J(our)P(aper)}}$), which is a particular case of publication, with some title ($b_{\mathrm{D(iD)C(wFOL)}}$) and author Serge Abiteboul, who is a researcher.

Though unique up to entailment (i.e., semantically unique), an `lgg` may have many syntactical forms due to *redundant* triples. Such triples can be either explicit ones that could have been left implicit if the set of RDF entailment rules at hand allows deriving them from the remaining triples (e.g., materializing the only $\mathcal{G}_{\texttt{lgg}}$ implicit triple in Fig. 3 would make it redundant if we consider the entailment rules in Table 2) or triples generalizing others without needing RDF entailment rules, i.e., w.r.t. $\models_\emptyset$ (e.g., adding the triple $(b, \mathrm{hasAuthor}, b')$ to $\mathcal{G}_{\texttt{lgg}}$ in Fig. 3 would be redundant w.r.t. $(b_{i_1 i_2}, \mathrm{hasAuthor}, SA)$). Also, an `lgg` may have *several minimal* syntactical variants obtained by pruning out redundant triples. For example, think of a minimal `lgg` comprising the triples $(A, \prec_{sc}, B)$, $(B, \prec_{sc}, A)$ and (b, τ, A), i.e., there exists an instance of the class A, which is equivalent to class B. Clearly, an equivalent and minimal variant of this `lgg` is the RDF graph comprising the triples $(A, \prec_{sc}, B)$, $(B, \prec_{sc}, A)$ and (b, τ, B).

Importantly, the above discussion is not specific to lggs of RDF graphs, since any RDF graph may feature redundancy. The detection and elimination of RDF graph redundancy has been studied in the literature, e.g., [21,24], hence we focus in this work on computing *some* lgg of RDF graphs.

The proposition below states that an lgg of n RDF graphs, with $n \geq 3$, can be defined (hence computed) as a sequence of $n-1$ lggs of *two* RDF graphs. Intuitively, assuming that $\ell_{k\geq 2}$ is an operator computing an lgg of k input RDF graphs, the next proposition establishes that:

$$\begin{array}{lll} \ell_3(\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3) & \equiv_{\mathcal{R}} & \ell_2(\ell_2(\mathcal{G}_1, \mathcal{G}_2), \mathcal{G}_3) \\ \dots & & \dots \\ \ell_n(\mathcal{G}_1, \dots, \mathcal{G}_n) & \equiv_{\mathcal{R}} & \ell_2(\ell_{n-1}(\mathcal{G}_1, \dots, \mathcal{G}_{n-1}), \mathcal{G}_n) \\ & \equiv_{\mathcal{R}} & \ell_2(\ell_2(\cdots \ell_2(\ell_2(\mathcal{G}_1, \mathcal{G}_2), \mathcal{G}_3) \cdots, \mathcal{G}_{n-1}), \mathcal{G}_n) \end{array}$$

Proposition 1. *Let $\mathcal{G}_1, \dots, \mathcal{G}_{n\geq 3}$ be n RDF graphs and $\mathcal{R}$ a set of RDF entailment rules. $\mathcal{G}_{\texttt{lgg}}$ is an* lgg *of $\mathcal{G}_1, \dots, \mathcal{G}_n$ iff $\mathcal{G}_{\texttt{lgg}}$ is an* lgg *of an* lgg *of $\mathcal{G}_1, \dots, \mathcal{G}_{n-1}$ and $\mathcal{G}_n$.*

Based on the above result, without loss of generality, we focus in the next section on the following problem:

Problem 1. Given *two* RDF graphs $\mathcal{G}_1, \mathcal{G}_2$ and a set $\mathcal{R}$ of RDF entailment rules, we want to compute *some* lgg of $\mathcal{G}_1$ and $\mathcal{G}_2$.

3.2 Computing an lgg of RDF Graphs

We first devise the *cover graph* of two RDF graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ (to be defined shortly, Definition 2 below), which is central to our technique for computing an lgg of $\mathcal{G}_1$ and $\mathcal{G}_2$. We indeed show (Theorem 2) that this particular RDF graph corresponds to an lgg of $\mathcal{G}_1$ and $\mathcal{G}_2$ when considering their explicit triples *only*, i.e., ignoring RDF entailment rules. Then, we show the main result of this section (Theorem 3): an lgg of $\mathcal{G}_1$ and $\mathcal{G}_2$, for *any set* $\mathcal{R}$ of RDF entailment rules, is the cover graph of their saturations w.r.t. $\mathcal{R}$. We also provide the worst-case size of cover graph-based lggs, as well as the worst-case time to compute them.

Definition 2 (Cover graph). *The* cover graph $\mathcal{G}$ *of two RDF graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ is the RDF graph, which may be empty, such that for every property p in both $\mathcal{G}_1$ and $\mathcal{G}_2$:*

$$(t_1, p, t_2) \in \mathcal{G}_1 \text{ and } (t_3, p, t_4) \in \mathcal{G}_2 \text{ iff } (t_5, p, t_6) \in \mathcal{G}$$

with $t_5 = t_1$ if $t_1 = t_3$ and $t_1 \in \mathcal{U} \cup \mathcal{L}$, else t_5 is the blank node $b_{t_1 t_3}$, and, similarly $t_6 = t_2$ if $t_2 = t_4$ and $t_2 \in \mathcal{U} \cup \mathcal{L}$, else t_6 is the blank node $b_{t_2 t_4}$.

The cover graph is a *generalization* of $\mathcal{G}_1$ and $\mathcal{G}_2$ (first item in Definition 1) as each of its triple (t_5, p, t_6) is a *least general anti-unifier* of a triple (t_1, p, t_2) from $\mathcal{G}_1$ and a triple (t_3, p, t_4) from $\mathcal{G}_2$. The notion of *least general anti-unifier* [25,26,29] is dual to the well-known notion of *most general unifier* [28,29]. Observe that $\mathcal{G}$'s triples result from anti-unifications of $\mathcal{G}_1$ and $\mathcal{G}_2$ triples with *same* property

URI. Indeed, anti-unifying triples of the form $(\mathtt{s}_1, p, \mathtt{o}_1)$ and $(\mathtt{s}_2, p', \mathtt{o}_2)$, with $p \neq p'$, would lead to a non-well-formed triples of the form $(\mathtt{s}, b_{pp'}, \mathtt{o})$ (recall that property values *must* be URIs in RDF graphs), where $b_{pp'}$ is the blank node required to generalize the distinct values p and p'.

Further, the cover graph is an `lgg` for the explicit triples in $\mathcal{G}_1$ and those in $\mathcal{G}_2$ (second item in Definition 1) since, intuitively, we capture their *common structures* by consistently naming, across all the anti-unifications begetting $\mathcal{G}$, the blank nodes used to generalize pairs of distinct subject values or of object values: each time the distinct values t from $\mathcal{G}_1$ and t' from $\mathcal{G}_2$ are generalized by a blank node while anti-unifying two triples, it is *always* by the same blank node $b_{tt'}$ in $\mathcal{G}$. This way, we establish *joins* between $\mathcal{G}$ triples, which reflect the common join structure on t within $\mathcal{G}_1$ and on t' within $\mathcal{G}_2$. For instance in Fig. 3, the *explicit* triples $(i_1, \tau, \text{ConfPaper})$, $(\text{ConfPaper}, \preceq_{\text{sc}}, \text{Publication})$, $(i_1, \text{title}, \text{“DiD”})$ in $\mathcal{G}_1$, and $(i_2, \tau, \text{JourPaper})$, $(\text{JourPaper}, \preceq_{\text{sc}}, \text{Publication})$, $(i_2, \text{title}, \text{“CwFOL”})$ in $\mathcal{G}_2$, lead to the triples $(b_{i_1 i_2}, \tau, b_{\text{CPJP}})$, $(b_{\text{CPJP}}, \preceq_{\text{sc}}, \text{Publication})$, $(b_{i_1 i_2}, \text{title}, b_{\text{DC}})$ in the cover graph of $\mathcal{G}_1$ and $\mathcal{G}_2$ shown in Fig. 4 (top). The first above-mentioned $\mathcal{G}$ triple results from anti-unifying i_1 and i_2 into $b_{i_1 i_2}$, and, ConfPaper and JourPaper into b_{CPJP}. The second results from anti-unifying *again* ConfPaper and JourPaper into b_{CPJP}, and, Publication and Publication into Publication (as a constant is its own least general generalization). Finally, the third results from anti-unifying *again* i_1 and i_2 into $b_{i_1 i_2}$, and, “DiD” and “CwFOL” into b_{DC}. By reusing consistently the same blank node name $b_{i_1 i_2}$ for each anti-unification of the constants i_1 and i_2 (resp. b_{CPJP} for ConfPaper and JourPaper)), the cover graph triples join on $b_{i_1 i_2}$ (resp. b_{CPJP}) in order to reflect that, in $\mathcal{G}_1$ and in $\mathcal{G}_2$, there exists a particular case of publication (i_1 in $\mathcal{G}_1$ and i_2 in $\mathcal{G}_2$) with some title (“DiD” in $\mathcal{G}_1$ and “CwFOL” in $\mathcal{G}_2$).

The next theorem formalizes the above discussion by stating that the cover graph of two RDF graphs is an `lgg` of them, *just in case of an empty set of RDF entailment rules.*

Theorem 2. *The* cover graph $\mathcal{G}$ *of the RDF graphs* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *exists and is an* `lgg` *of them for the empty set* $\mathcal{R}$ *of RDF entailment rules (i.e.,* $\mathcal{R} = \emptyset$*).*

We provide below worst-case bounds for the time to compute a cover graph and for its size; these bounds are met when *all the triples of the two input graphs use the same property URI* (i.e., every pair of $\mathcal{G}_1$ and $\mathcal{G}_2$ triples begets a $\mathcal{G}$ triple).

Proposition 2. *The cover graph of two RDF graphs* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *can be computed in* $O(|\mathcal{G}_1| \times |\mathcal{G}_2|)$*; its size is bounded by* $|\mathcal{G}_1| \times |\mathcal{G}_2|$*.*

The main theorem below generalizes Theorem 2 in order to take into account any set of entailment rules from the RDF standard. It states that it is sufficient to compute the cover graph of the saturations of the input RDF graphs, instead of the input RDF graphs themselves.

Theorem 3. *Let* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *be two RDF graphs, and* $\mathcal{R}$ *a set of RDF entailment rules. The* cover graph $\mathcal{G}$ *of* $\mathcal{G}_1^\infty$ *and* $\mathcal{G}_2^\infty$ *exists and is an* `lgg` *of* $\mathcal{G}_1$ *and* $\mathcal{G}_2$*.*

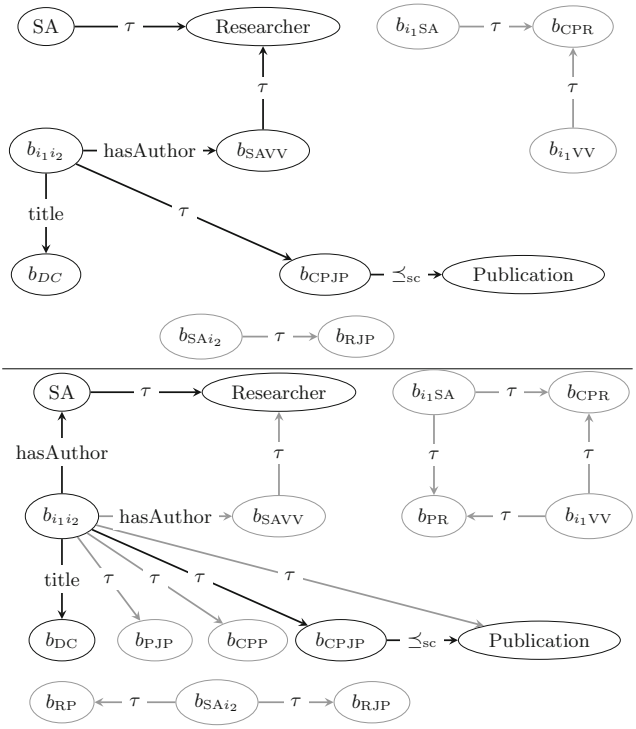

Fig. 4. Cover graphs of $\mathcal{G}_1$ and $\mathcal{G}_2$ in Fig. 3 (top) and of their saturations w.r.t. the entailment rules in Table 2 (bottom). Triples shown in gray are redundant w.r.t. those shown in black: they are part of the graph, while implicit to the triples shown in black.

As an immediate consequence of the above results, we get the following worst-case bounds for the time to compute a cover graph-based `lgg` of two RDF graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, and for its size. Here, we assume given the saturation $\mathcal{G}_1^\infty$ and $\mathcal{G}_2^\infty$, as the times to compute them and their sizes depend on the RDF entailment rules at hand.

Corollary 1. *An* `lgg` *of two RDF graphs* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *can be computed in* $O(|\mathcal{G}_1^\infty| \times |\mathcal{G}_2^\infty|)$ *and its size is bounded by* $|\mathcal{G}_1^\infty| \times |\mathcal{G}_2^\infty|$.

Remark that computing naively the cover graph-based `lgg` of n RDF graphs of size M based on Proposition 1 may lead to an `lgg` of size M^n, in the unlikely worst-case where all the triples of all the RDF graphs use the same property URI. However, removing the redundant triples from the intermediate and final cover graph-based `lgg`s limits their size to at most M.

Figure 4 (bottom) displays an `lgg` of the RDF graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ in Fig. 3 w.r.t. the entailment rules shown in Table 2. In contrast to Fig. 4 (top), which shows an `lgg` of the same RDF graphs when RDF entailment rules are ignored, we further learn that Serge Abiteboul is an author of some particular publication (i_1 in $\mathcal{G}_1$ and i_2 in $\mathcal{G}_2$). Moreover, removing the redundant triples (the gray ones) yields precisely the `lgg` $\mathcal{G}_{\texttt{lgg}}$ of $\mathcal{G}_1$ and $\mathcal{G}_2$ shown in Fig. 3.

4 Algorithms

We provide algorithms to compute `lgg`s of RDF graphs based on the results obtained in the preceding Section. In Sect. 4.1, we present an algorithm for computing the least general anti-unifiers of triples. Then, in Sect. 4.2, we give three algorithms to compute a cover graph-based `lgg` of RDF graphs, which allow handling RDF graphs of increasing size, i.e., when the input and output RDF graphs fit in memory, in data management systems or in MapReduce clusters. Also, to choose between these algorithms, we show how the exact size of a cover graph-based `lgg` they produce can be calculated, *without computing this* `lgg`.

4.1 Least General Anti-unifier of Triples

Algorithm 1, called `lgau`, computes a least general anti-unifier (t_1^T, t_2^T, t_3^T) of two triples (t_1, t_2, t_3) and (t'_1, t'_2, t'_3). This is achieved by setting the i^{th} value t_i^T of the output triple to the least general generalization of the values found at the i^{th} positions of the two input triples: t_i and t'_i. Recall that a pair of a same constant is generalized by this constant itself, otherwise the generalization leads to a blank node (Sect. 3.2). Crucially, such a blank node uses the consistent naming scheme devised in Sect. 3.2, which allows us preserving the common structure of input RDF graphs across the anti-unifications of their triples.

Algorithm 1. Least general anti-unification: `lgau`

In: triples $T_1 = (t_1, t_2, t_3)$ and $T_2 = (t'_1, t'_2, t'_3)$
Out: least general anti-unification T of T_1 and T_2

1: **for** $i = 1$ **to** 3 **do** ▷ *for each pair of* T_1 *and* T_2 i^{th} *values*
2: **if** $t_i = t'_i$ and $t_i \in \mathcal{U} \cup \mathcal{L}$ **then**
3: $t_i^T \leftarrow t_i$ ▷ *generalization of a same constant by itself*
4: **else**
5: $t_i^T \leftarrow b_{t_i t'_i}$ ▷ *otherwise generalization by a blank node*
6: **return** (t_1^T, t_2^T, t_3^T)

4.2 `lgg` of RDF Graphs

Following Definition 2, Algorithm 2, called `lgg4g`, computes the cover graph $\mathcal{G}$ of two input RDF graphs $\mathcal{G}_1$ and $\mathcal{G}_2$: $\mathcal{G}$ comprises the least general anti-unifier of every pair of $\mathcal{G}_1$ and $\mathcal{G}_2$ triples with *same* property. Therefore, given two RDF graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, a call `lgg4g`$(\mathcal{G}_1, \mathcal{G}_2)$ produces the cover graph-based `lgg` of $\mathcal{G}_1$ and $\mathcal{G}_2$ ignoring RDF entailment (Theorem 2), while a call `lgg4g`$(\mathcal{G}_1^\infty, \mathcal{G}_2^\infty)$ produces the cover graph-based `lgg` of $\mathcal{G}_1$ and $\mathcal{G}_2$ taking into account the set of RDF entailment rules at hand (Theorem 3). In the latter case, the input RDF graphs can be saturated using standard algorithms implemented in RDF reasoners or data management systems, like Jena [3] and Virtuoso [8].

Importantly, `lgg4g` assumes that the input RDF graphs, as well as their output cover graph, fit in memory. Checking whether this is the case for the input RDF graphs under consideration can be done as follows.

Algorithm 2. Cover graph of two RDF graphs: `lgg4g`

In: RDF graphs $\mathcal{G}_1$ and $\mathcal{G}_2$
Out: $\mathcal{G}$ is the cover graph of $\mathcal{G}_1$ and $\mathcal{G}_2$
1: $\mathcal{G} \leftarrow \emptyset$
2: **for all** $T_1 = (\mathtt{s}_1, \mathtt{p}_1, \mathtt{o}_1) \in \mathcal{G}_1$ **do**
3: **for all** $T_2 = (\mathtt{s}_2, \mathtt{p}_2, \mathtt{o}_2) \in \mathcal{G}_2$ with $\mathtt{p}_1 = \mathtt{p}_2$ **do**
4: $\mathcal{G} \leftarrow \mathcal{G} \cup \{\mathtt{lgau}(T_1, T_2)\}$ ▷ *add to* $\mathcal{G}$ *the least general anti-unifier of* T_1 *and* T_2
5: **return** $\mathcal{G}$

The size of the input RDF graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ can be computed with the following SPARQL queries counting how many triples each of them holds: `SELECT count(*) as` $?size$ `FROM` $\mathcal{G}_i$ with $i \in [1, 2]$. Recall that the worst-case size of the output cover graph is $|\mathcal{G}| = |\mathcal{G}_1| \times |\mathcal{G}_2|$ in the unlikely case where *all* the $\mathcal{G}_1$ and $\mathcal{G}_2$ triples use the same property (Proposition 2 and Corollary 1).

The precise size of the output cover graph $\mathcal{G}$ can be computed, *without* computing $\mathcal{G}$, with SPARQL queries. First, we calculate for each input RDF graph $\mathcal{G}_i$, with $i \in [1, 2]$, how many triples it holds per distinct property p:

$$S_{\mathcal{G}_i} = \{(p, n_i) \mid |\{(\mathtt{s}, p, \mathtt{o}) \in \mathcal{G}_i\}| = n_i\}$$

This can be computed with the SPARQL query: `SELECT` $?p$ `count(*) as` $?n_i$ `FROM` $\mathcal{G}_i$ `WHERE` $\{(?s, ?p, ?o)\}$ `GROUP BY` $?p$. Then, since every $\mathcal{G}_1$ triple with property p anti-unifies with every $\mathcal{G}_2$ triple with same property p in order to beget $\mathcal{G}$, the size of $\mathcal{G}$ is: $|\mathcal{G}| = \sum_{(p,n_1) \in S_{\mathcal{G}_1}, (p,n_2) \in S_{\mathcal{G}_2}} n_1 \times n_2$.

This can be computed with the SPARQL query: `SELECT SUM(`$?n_1$`*`$?n_2$`) as` $?size$ `WHERE` $\{\{S_{\mathcal{G}_1}\}\{S_{\mathcal{G}_2}\}\}$ with $S_{\mathcal{G}_1}$ and $S_{\mathcal{G}_2}$ denoting the above SPARQL queries computing these two sets, which join on their common answer variable $?p$.

When the input RDF graphs or their output cover graph cannot fit in memory, we propose variants of `lgg4g` that either assume that RDF graphs are stored in data management systems (DMSs, in short) or in a MapReduce cluster.

Handling Large RDF Graphs Using DMSs. Algorithm 3, called `lgg4g-dms`, is an adaptation of `lgg4g`, which assumes that the input RDF graphs (already saturated if needed) and their cover graph are all stored in one or several DMSs. It further assumes that the system(s) storing the input RDF graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ feature(s) the well-known database mechanism of *cursor* [17,27]. This is for instance the case for RDF graphs stored in relational servers like DB2 [1], MySQL [4], Oracle [5] and PostgreSQL [6], or in RDF servers like Jena-TDB [3] and Virtuoso [8]. Roughly speaking, a cursor is a pointer or iterator on tuples held in a DMS (e.g., stored as relation or computed as the results to a query) that can be used to access these tuples. In particular, a cursor can be used by an application to iteratively traverse all the tuples by fetching n of them at a time.

`lgg4g-dms` uses cursors to proceed similarly to `lgg4g` (remark that lines 5–7 in Algorithm 3 are almost the same as lines 2–4 in Algorithm 2) on pairs of n-triples subsets of $\mathcal{G}_1$ and of $\mathcal{G}_2$, instead of on the whole RDF graphs themselves. It follows that the worst-case number of triples kept in memory by `lgg4g-dms` is $M = (2 \times n) + 1$ at line 7 (i.e., n for B_1, n for B_2, and the anti-unifier triple output by `lgau`), with:

Algorithm 3. Cover graph of two RDF graphs: `lgg4g-dms`

In: cursor c_1 on RDF graph $\mathcal{G}_1$, cursor c_2 on RDF graph $\mathcal{G}_2$, data access path d to an empty RDF graph $\mathcal{G}$, integer n
Out: $\mathcal{G}$ is the cover graph of $\mathcal{G}_1$ and $\mathcal{G}_2$

1: c_1.`init`() ▷ *c_1 at beginning of $\mathcal{G}_1$ triples set*
2: **while** $B_1 = c_1$.`next`(n) **do** ▷ *fetch the next block B_1 of n $\mathcal{G}_1$ triples*
3: c_2.`init`() ▷ *c_2 at beginning of $\mathcal{G}_2$ triples set*
4: **while** $B_2 = c_2$.`next`(n) **do** ▷ *fetch the next block B_2 of n $\mathcal{G}_2$ triples*
5: **for all** $T_1 = (\mathtt{s}_1, \mathtt{p}_1, \mathtt{o}_1) \in B_1$ **do**
6: **for all** $T_2 = (\mathtt{s}_2, \mathtt{p}_2, \mathtt{o}_2) \in B_2$ with $\mathtt{p}_1 = \mathtt{p}_2$ **do**
7: d.`insert`(`lgau`(T_1, T_2))

$$3 \leq M \leq |\mathcal{G}_1| + |\mathcal{G}_2| + 1$$

The above lower bound is met for n set to 1, while the upper one is met for n set to $\max(|\mathcal{G}_1|, |\mathcal{G}_2|)$. Importantly, `lgg4g-dms` allows *choosing* the value of n in order to reflect the memory devoted to handling triples. For instance, if one wants to use 4 GB of RAM for triples, assuming that any triple fits in less one 1 KB (this value is much less when using dictionary encoding [22], i.e., when triples values are mapped to integers), the value of n can be set to 2M.

This clearly contrasts with the worst-case number of triples kept in memory by `lgg4g`: $M = |\mathcal{G}_1| + |\mathcal{G}_2| + |\mathcal{G}|$ at line 5, with:

$$|\mathcal{G}_1| + |\mathcal{G}_2| \leq M \leq |\mathcal{G}_1| + |\mathcal{G}_2| + (|\mathcal{G}_1| \times |\mathcal{G}_2|)$$

The above lower bound is met when $\mathcal{G}_1$ and $\mathcal{G}_2$ have no property in common in their triples (i.e., $|\mathcal{G}| = 0$), while the upper one is met in the unlikely case where $\mathcal{G}_1$ and $\mathcal{G}_2$ use a same property in all their triples (i.e., $|\mathcal{G}| = |\mathcal{G}_1| \times |\mathcal{G}_2|$).

Handling Huge RDF Graphs Using MapReduce. Algorithm 4, called `lgg4g-mr`, is a MapReduce (MR) variant of `lgg4g`. MR is a popular massively parallel programming framework [15], implemented by large-scale data processing systems, like Hadoop [2] and Spark [7], which orchestrate clusters of compute nodes.

Algorithm 4. Cover graph of two RDF graphs: `lgg4g-mr`

In: file G_1 for RDF graph $\mathcal{G}_1$, file G_2 for RDF graph $\mathcal{G}_2$
Out: $\mathcal{G}$ is the cover graph of $\mathcal{G}_1$ and $\mathcal{G}_2$, stored in G-$*$ files

`Map`(key: file G_i, value: triple $T_i = (\mathtt{s}_i, \mathtt{p}_i, \mathtt{o}_i)$)
1: `emit`$(\langle \mathtt{p}_i, (G_i, T_i) \rangle)$

`Reduce`(key: p, values: set $\mathcal{V}$ of values emitted for key p)
1: $f \leftarrow$ `open`$(G$-$p)$
2: **for all** $(G_1, T_1 = (\mathtt{s}_1, p, \mathtt{o}_1)) \in \mathcal{V}$ **do**
3: **for all** $(G_2, T_2 = (\mathtt{s}_2, p, \mathtt{o}_2)) \in \mathcal{V}$ **do**
4: $f.write$(`lgau`(T_1, T_2))
5: `close`(f)

A MR program is organized in successive *jobs*, each of which comprises a *Map task* followed by a *Reduce task*. The Map task consists in reading some input data from the distributed file system[2] of the cluster, so as to partition the data into $\langle \mathtt{k}, \mathtt{v} \rangle$ key-value pairs. Importantly, an MR engine transparently processes the Map task by running *Mapper processes* in parallel on cluster nodes, each process taking care of partitioning a portion of the input data by applying a `Map`(key: file, value: data unit) function on every data unit of a given input file. Key-value pairs thus produced are shuffled across the network, so that *all* pairs with *same* key $\langle \mathtt{k}, \mathtt{v}_1 \rangle \cdots \langle \mathtt{k}, \mathtt{v}_n \rangle$ are shipped to a same compute node. The Reduce task consists in running *Reducer processes* in parallel, for every distinct key `k` received by every compute node. Each process takes care of the set $\mathcal{V}$ of values $\{\mathtt{v}_1, \ldots, \mathtt{v}_n\}$ emitted with key `k`, by applying a `Reduce`(key: k, values: $\mathcal{V}$) function, and writing its results in a file. The result of an MR job comprises the data, stored in a distributed fashion, in all the files output by Reducers.

In `lgg4g-mr`, the `Map` function applies to every $(\mathtt{s}_i, \mathtt{p}_i, \mathtt{o}_i)$ triple of the input RDF graph $\mathcal{G}_i$ stored in file G_i, and produces the corresponding key-value pair $\langle \mathtt{p}_i, (G_i, (\mathtt{s}_i, \mathtt{p}_i, \mathtt{o}_i)) \rangle$, for $i \in [1, 2]$. Hence, all the $\mathcal{G}_1$ and $\mathcal{G}_2$ triples with a same key/property p are shipped to the same cluster node. Then, similarly to `lgg4g` at lines 2–4, the `Reduce` functions process, on each node, the set $\mathcal{V}$ of values emitted for every received key p. The least general anti-unifier triples obtained at line 4 are stored in the output file G-p. At the end of the MR job, the `lgg` $\mathcal{G}$ of $\mathcal{G}_1$ and $\mathcal{G}_2$ is stored in the G-$*$ files of the distributed file system, where $*$ denotes any key/property p.

A `Map` function holds at most a single $\mathcal{G}_1$ or $\mathcal{G}_2$ triple in memory. In contrast, the worst-case number of triples handled by a `Reduce` function for a given key p is: $M = |\mathcal{G}_1| + |\mathcal{G}_2| + 1$ at line 4. This upper bound is met in the unlikely case where $\mathcal{G}_1$ and $\mathcal{G}_2$ use the same property p in all their triples. Similarly to `lgg4g-dms`, this upper bound can set to $M = (2 \times n) + 1$, with $3 \leq M \leq |\mathcal{G}_1| + |\mathcal{G}_2| + 1$, by first splitting the input RDF graphs in k_i files of n $\mathcal{G}_i$ triples (files $G_i^1, \ldots, G_i^{k_i}$), and then by processing every pair of such files with an MR job (i.e., with $k_1 \times k_2$ jobs), instead of a single MR job for the entire two input RDF graphs.

Finally, to take into account RDF entailment, input RDF graphs can be saturated before being stored in the MR cluster using standard (centralized) techniques, or within the MR cluster using MR-based saturation techniques [30]. Also, it is worth noting that RDF graphs, hence `lgg`s of them, stored in an MR cluster can be queried with MR-based SPARQL engines [18,23].

5 Related Work and Conclusion

We revisited the Machine Learning problem of computing a *least general generalization (`lgg`) of some descriptions* in the setting of RDF; it was introduced to generalize First Order Logic clauses w.r.t. θ-subsumption [25,26], a non-standard specialization/generalization relation widely used in Machine Learning.

[2] We assume w.l.o.g. that input and output data of an MR job is stored on disk, like in Hadoop, while it can also reside in in-memory shared data structures, like in Spark.

This problem has also been investigated in Knowledge Representation, for formalisms whose expressivity overlaps with our RDF setting, notably Description Logics (DLs) [9,19,33] and Conceptual Graphs (CGs) [11]. Finally, recently, this problem has started receiving attention in the Semantic Web field [13,14,20].

In DLs, computing an `lgg` of concepts (formulae) has been studied for $\mathcal{EL}$ and extensions thereof [9,19,33]. The $\mathcal{EL}$ setting translates into *particular tree-shaped* RDF graphs, which may feature RDFS subclass and domain constraints, and for which RDF entailment is limited to the use of these two constraints only[3]. In these equivalent RDF and $\mathcal{EL}$ fragments, the $\mathcal{EL}$ technique that computes an `lgg` of $\mathcal{EL}$ concepts, which is an $\mathcal{EL}$ concept, provides *only a (non least general) generalization* of their corresponding tree-shaped RDF graphs w.r.t. the problem we study: the (minimal) cover graph-based `lgg` of tree-shaped RDF graphs is clearly a forest-shaped RDF graph in general. In CGs, the so-called *simple CGs with unary and binary relations* correspond to *particular* RDF graphs (e.g., a property URI in a triple cannot be the subject or object of another triple, a class - URI or blank node - in a τ triple cannot be the subject of another τ triple nor the subject or object of another non-τ triple, etc.), which may feature the four RDFS constraints, and for which RDF entailment is limited to the use of these RDFS constraints only [10]. In these equivalent RDF and CG fragments, we may interchangeably compute `lgg`s with the CG technique in [11] or ours.

In RDF, computing an `lgg` has been studied for *particular* RDF graphs, called *r-graphs*, *ignoring RDF entailment* [13,14]. An r-graph is an *extracted subgraph* of an RDF graph $\mathcal{G}$, *rooted* in the $\mathcal{G}$ value r and comprising the $\mathcal{G}$ triples *reachable from r through directed* paths of length at most n. Such a rooted and directed r-graph can be defined recursively as $\mathcal{S}(\mathcal{G}, r, n)$, with:

$$\mathcal{S}(\mathcal{G}, v, 0) = \emptyset \mid \mathcal{S}(\mathcal{G}, v, n) = \bigcup_{(v,p,v') \in \mathcal{G}} \{(v, p, v')\} \cup \mathcal{S}(\mathcal{G}, v', n-1) \cup \mathcal{S}(\mathcal{G}, p, n-1)$$

Intuitively, this purely structural definition of r-graph attempts carrying $\mathcal{G}$'s knowledge about r. `lgg`s of r-graphs allow finding the commonalities between *single root entities*, while with general RDF graphs we further allow finding the commonalities between *sets of multiple interrelated entities*. The technique for computing an `lgg` of two r-graphs exploits their rooted and directed structure: it starts from their respective root and traverses them simultaneously considering triples reachable through directed paths of increasing size, while incrementally constructing an r-graph `lgg`. In contrast, the general RDF graphs we consider are unstructured; our technique blindly traverses the input RDF graphs to anti-unify their triples with same property, and captures their common structure across these anti-unifications thanks to the consistent naming scheme we devised

[3] An $\mathcal{EL}$ concept C recursively translates into the RDF graph rooted in the blank node b_r returned by the call $\mathcal{G}(C, b_r)$, with: $\mathcal{G}(\top, b) = \emptyset$ for the universal $\mathcal{EL}$ concept $\top$, $\mathcal{G}(A, b) = \{(b, \tau, A)\}$ for an atomic $\mathcal{EL}$ concept A, $\mathcal{G}(\exists r.C, b) = \{(b, r, b')\} \cup \mathcal{G}(C, b')$, with b' a fresh blank node, for an $\mathcal{EL}$ existential restriction $\exists r.C$, and $\mathcal{G}(C_1 \sqcap C_2, b) = \mathcal{G}(C_1, b) \cup \mathcal{G}(C_2, b)$ for an $\mathcal{EL}$ conjunction $C_1 \sqcap C_2$; the $\mathcal{EL}$ constraints $A_1 \sqsubseteq A_2$ and $\exists r.\top \sqsubseteq A$ correspond to $(A_1, \preceq_{\text{sc}}, A_2)$ and $(r, \hookrightarrow_d, A)$ resp.

for the blank nodes they generate. The r-graph technique that computes an `lgg` of r-graphs, which is an r-graph, gives *only a (non least general) generalization* of them w.r.t. the problem we study: the (minimal) cover graph-based `lgg` of r-graphs is clearly a general RDF graph.

In SPARQL, computing an `lgg` has been considered for *unary tree-shaped conjunctive queries (UTCQ)* [20]; a UTCQ `lgg` is computed by a simultaneous root-to-leaves traversal of the input queries. UTCQs are tree-shaped RDF graphs, when variables are viewed as blank nodes, for which RDF entailment is ignored [13]. The UTCQ technique that computes an `lgg` of UTCQs, which is a UTCQ, yields *only a (non least general) generalization* of their corresponding tree-shaped RDF graphs w.r.t. the problem we study: the (minimal) cover graph-based `lgg` of tree-shaped RDF graphs is clearly a forest-shaped RDF graph.

Our work significantly extends the state of the art on computing `lgg`s of RDF graphs by considering the *entire* RDF standard of W3C. Crucially, we neither restrict RDF graphs nor RDF entailment in any way, while related works consider *particular* RDF graphs and, further, *ignore* RDF entailment, hence do not accurately capture the semantics of RDF graphs. Also, we provide a set of algorithms that allows computing `lgg`s of small-to-huge general RDF graphs.

As future work, we want to study heuristics in order to efficiently prune out as much as possible redundant triples, while computing `lgg`s. Indeed, as for instance Fig. 4 shows, our cover graph technique does produce redundant triples. This would allow having more compact `lgg`s, and reducing the a posteriori elimination effort of redundant triples using standard technique from the literature.

References

1. DB2. www.ibm.com/analytics/us/en/technology/db2
2. Hadoop. hadoop.apache.org
3. Jena. jena.apache.org
4. MySQL. www.mysql.com
5. Oracle. www.oracle.com/database
6. PostgreSQL. www.postgresql.org
7. Spark. spark.apache.org
8. Virtuoso. virtuoso.openlinksw.com
9. Baader, F., Sertkaya, B., Turhan, A.Y.: Computing the least common subsumer w.r.t. a background terminology. J. Appl. Logic **5**(3), 392–420 (2007)
10. Baget, J., Croitoru, M., Gutierrez, A., Leclère, M., Mugnier, M.: Translations between RDF(S) and conceptual graphs. In: ICCS (2010)
11. Chein, M., Mugnier, M.: Graph-Based Knowledge Representation - Computational Foundations of Conceptual Graphs. Springer, London (2009)
12. Cohen, W.W., Borgida, A., Hirsh, H.: Computing least common subsumers in description logics. In: AAAI (1992)
13. Colucci, S., Donini, F., Giannini, S., Sciascio, E.D.: Defining and computing least common subsumers in RDF. J. Web Semant. **39**, 62–80 (2016)
14. Colucci, S., Donini, F.M., Sciascio, E.D.: Common subsumbers in RDF. In: AI*IA (2013)

15. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI (2004)
16. El Hassad, S., Goasdoué, F., Jaudoin, H.: Learning commonalities in RDF and SPARQL (research report) (2016). https://hal.inria.fr/hal-01386237
17. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems - The Complete Book. Pearson Education, Harlow (2009)
18. Goasdoué, F., Kaoudi, Z., Manolescu, I., Quiané-Ruiz, J., Zampetakis, S.: Cliquesquare: flat plans for massively parallel RDF queries. In: ICDE (2015)
19. Küsters, R.: Non-standard Inferences in Description Logics. LNCS, vol. 2100. Springer, Heidelberg (2001)
20. Lehmann, J., Bühmann, L.: AutoSPARQL: let users query your knowledge base. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., Leenheer, P., Pan, J. (eds.) ESWC 2011. LNCS, vol. 6643, pp. 63–79. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21034-1_5
21. Meier, M.: Towards rule-based minimization of RDF graphs under constraints. In: Calvanese, D., Lausen, G. (eds.) RR 2008. LNCS, vol. 5341, pp. 89–103. Springer, Heidelberg (2008). doi:10.1007/978-3-540-88737-9_8
22. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. **19**(1), 91–113 (2010)
23. Papailiou, N., Tsoumakos, D., Konstantinou, I., Karras, P., Koziris, N.: H_2rdf+: an efficient data management system for big RDF graphs. In: SIGMOD (2014)
24. Pichler, R., Polleres, A., Skritek, S., Woltran, S.: Complexity of redundancy detection on RDF graphs in the presence of rules, constraints, and queries. Semant. Web **4**(4), 351–393 (2013)
25. Plotkin, G.D.: A note on inductive generalization. Mach. Intell. **5**, 153–163 (1970)
26. Plotkin, G.D.: A further note on inductive generalization. Mach. Intell. **6**, 101–124 (1971)
27. Ramakrishnan, R., Gehrke, J.: Database Management Systems. McGraw-Hill, New York (2003)
28. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965)
29. Robinson, J.A., Voronkov, A. (eds.): Handbook of Automated Reasoning. Elsevier and MIT Press, Weidenbach (2001)
30. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.E.: WebPIE: a web-scale parallel inference engine using MapReduce. J. Web Semant. **10**, 59–75 (2012)
31. Resource description framework 1.1. https://www.w3.org/TR/rdf11-concepts
32. RDF 1.1 semantics. https://www.w3.org/TR/rdf11-mt/
33. Zarrieß, B., Turhan, A.: Most specific generalizations w.r.t. general EL-TBoxes. In: IJCAI (2013)