Efficient query answering in the presence of DL-LiteR constraints

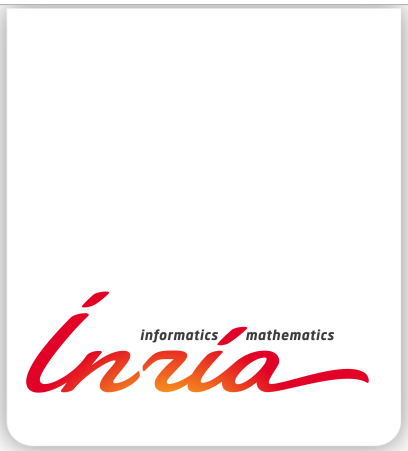
Damian Bursztyn, François Goasdoué, Ioana Manolescu

Efficient query answering in the presence of DL-Lite$_{\mathcal{R}}$ constraints

Damian Bursztyn, François Goasdoué, Ioana Manolescu

RESEARCH REPORT
N° 8714
April 2015
Project-Teams OAK

ISSN 0249-6399 ISRN INRIA/RR--8714--FR+ENG



Efficient query answering in the presence of DL-Lite$_\mathcal{R}$ constraints

Damian Bursztyn, François Goasdoué, Ioana Manolescu

Project-Teams OAK

Research Report n° 8714 — April 2015 — 13 pages

Abstract: We devise a query optimization framework for formalisms enjoying FOL reducibility of query answering, for which it reduces to the evaluation of a FOL query against facts. This framework allows searching within a set of alternative equivalent FOL queries, i.e., FOL reformulations, one with minimal evaluation cost when evaluated through a relational database system. We provide two algorithms, an exhaustive and a greedy, for exploring this space of alternatives. We apply this framework to the DL-Lite$_\mathcal{R}$ description logic underpinning the W3C's OWL2 QL profile; an experimental evaluation validates its interest and applicability.

Keywords: Query answering, query reformulation, query optimization, DL-Lite$_\mathcal{R}$

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Répondre efficacement aux requêtes en présence de contraintes DL-Lite$_{\mathcal{R}}$

Résumé : Nous établissons un cadre d'optimisation de requêtes pour les formalismes où le problème de répondre à une requête est FOL réductible, dans lesquels qu'il se ramène à l'évaluation d'une requête FOL sur des faits. Ce cadre permet de chercher au sein d'un ensemble de requêtes FOL équivalentes, c'est-à-dire des reformulations FOL, une alternative avec un coût d'évaluation minimal lorsqu'évaluée par un système de gestion de bases de données relationnelles. Nous fournissons deux algorithmes, un exhaustif et un glouton, pour explorer cet espace d'alternatives. Nous appliquons ce cadre à la logique de description DL-Lite$_{\mathcal{R}}$ sur laquelle se fonde le profile OWL2QL du W3C ; une évaluation expérimentale valide son intérêt et applicabilité.

Mots-clés : Réponse aux requêtes, reformulation de requête, optimisation de requête, DL-Lite$_{\mathcal{R}}$

1 Introduction

Query answering in the lightweight DL-Lite$_\mathcal{R}$ description logic [1] has received significant attention in the literature, as it provides the foundations of the W3C's OWL2 QL standard for Semantic Web applications. In particular, query answering techniques based on FOL reducibility e.g., [1, 2, 3, 4, 5, 6], which reduce query answering against a knowledge base (KB) to FOL query evaluation against the KB's facts only (a.k.a. ABox) by compiling the KB's domain knowledge (a.k.a. TBox) into the query, hold great potential for performance. This is because FOL queries can be evaluated by an optimized relational database management system (RDBMS) storing the KB's facts.

The fundamental goal of our study is to identify efficient techniques for query answering in description logics (DLs) enjoying FOL reducibility, with a focus on DL-Lite$_\mathcal{R}$. Notably, we reduce query answering to the evaluation of alternative FOL queries, a.k.a. FOL *reformulations*, belonging to richer languages than those considered so far in the literature; in particular, this may allow *several* (equivalent) FOL reformulations for the input query. This contrasts with related works, e.g., the aforementioned ones, which aim at a *single* FOL reformulation (modulo minimization). Allowing a variety of reformulations is crucial for efficiency, as such alternatives, while computing the same answers, may have very different evaluation costs when evaluated through an RDBMS. Therefore, instead of having a single default choice which may perform poorly, we select the one with lowest estimated evaluation cost among possible alternatives. More precisely, our contributions are:

1. For formalisms enjoying FOL reducibility of query answering, we provide a general optimization framework that reduces it to searching among a *set of alternative equivalent* FOL *queries*, i.e., FOL reformulations, one with minimal evaluation cost in an RDBMS (Section 3).
2. We characterize interesting spaces of such alternative equivalent FOL queries for DL-Lite$_\mathcal{R}$ (Section 4), called JUCQ or JUSCQ reformulations.
3. We then optimize query answering against DL-Lite$_\mathcal{R}$ KBs by searching within one of these spaces and picking the reformulation with lowest (estimated) evaluation cost w.r.t. an RDBMS cost model. We provide two algorithms, an exhaustive and a greedy, for selecting a low-cost reformulation (Section 5).
4. Finally, we demonstrate experimentally the effectiveness and the efficiency of our query answering technique for DL-Lite$_\mathcal{R}$, by deploying it on top of the Postgres RDBMS (Section 6).

In the sequel, we recall preliminary notions on DL-Lite$_\mathcal{R}$ in Section 2, then detail the above contributions, and finally we discuss related work and conclude in Section 7.

2 Preliminaries

We recall here the basics of the DL-Lite$_\mathcal{R}$ description logic [1].

DL-Lite knowledge bases. A *knowledge base (KB)* $\mathcal{K}$ consists of a TBox $\mathcal{T}$ (ontology) and an ABox $\mathcal{A}$ (dataset), denoted $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, with $\mathcal{T}$ expressing constraints on $\mathcal{A}$.

The TBox and ABox are constructed from a set N_C of *concept names* (unary predicates), a set N_R of *role names* (binary predicates), and a set N_I of *individuals* (constants). The *ABox* consists of a finite number of *concept assertions* of the form $A(a)$ with $A \in N_C$ and $a \in N_I$, and of *role assertions* of the form $R(a, b)$ with $R \in N_R$ and $a, b \in N_I$. The *TBox* consists of a set of axioms that can be *concept inclusions* of the form $B \sqsubseteq C$, or *role inclusions* of the form $Q \sqsubseteq S$, formed using the following syntax (where $A \in N_C$ and $R \in N_R$):

$$B := A \mid \exists Q,\ C := B \mid \neg B,\ Q := R \mid R^-,\ S := Q \mid \neg Q$$

An *interpretation* has the form $I = (\Delta^I, \cdot^I)$, where Δ^I is a non-empty set and $\cdot^I$ is a function mapping each $a \in N_I$ to $a^I \in \Delta^I$, each $A \in N_C$ to $A^I \subseteq \Delta^I$, and each $R \in N_R$ to $R^I \subseteq \Delta^I \times \Delta^I$. The function $\cdot^I$ is straightforwardly extended to concepts and roles:

- $(\exists Q)^I = \{o_1 \mid \exists o_2\ (o_1, o_2) \in Q^I\}$
- $(R^-)^I = \{(o_1, o_2) \mid (o_2, o_1) \in R^I\}$
- $(\neg B)^I = \Delta^I \setminus B^I$ and $(\neg Q)^I = (\Delta^I \times \Delta^I) \setminus Q^I$

An interpretation I satisfies an inclusion $B \sqsubseteq C$ (resp. $Q \sqsubseteq S$) if $B^I \subseteq C^I$ (resp. if $Q^I \subseteq S^I$); it satisfies $A(a)$ (resp. $R(a, b)$) if $a^I \in A^I$ (resp. $(a^I, b^I) \in R^I$). An interpretation I is a *model* of $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ if I satisfies all inclusions in $\mathcal{T}$ and assertions in $\mathcal{A}$. A KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ is *consistent* if it has a model. In this case, we say that $\mathcal{A}$ is $\mathcal{T}$*-consistent*.

Queries. A First Order Logic (FOL) query is a FOL formula $\phi(\bar{x})$ whose free variables are $\bar{x}$. The *arity* of a query is the number of its free variables, e.g., 0 for a *Boolean* query. Given an interpretation $I = (\Delta^I, .^I)$, the semantics of a Boolean query is defined as true if $\phi()^I = \mathrm{true}$ and false otherwise, while the semantics of a non-Boolean query of arity $n \geq 1$ is the relation of arity n defined on Δ^I as follows: $\{\bar{\mathbf{o}} \in (\Delta^I)^n \mid \phi(\bar{\mathbf{o}})^I = \mathrm{true}\}$. An interpretation that makes true a Boolean FOL query, or non-empty a non-Boolean FOL query, is a *model* of that query.

Given a CQ q asked against a KB $\mathcal{K}$:

- If q is Boolean, the answer set of q over $\mathcal{K}$ is defined as: $ans(q, \mathcal{K}) = \{()\}$ if $\mathcal{K} \models q()$, with $()$ the empty tuple, and $ans(q, \mathcal{K}) = \emptyset$ otherwise.
- If q is a non-Boolean query of arity n, the answer set of q against $\mathcal{K}$ is: $ans(q, \mathcal{K}) = \{\bar{t} \in (N_I)^n \mid \mathcal{K} \models q(\bar{t})\}$ where $\mathcal{K} \models q(\bar{t})$ means as usual that every model of $\mathcal{K}$ is a model of $q(\bar{t})$.

In this paper, we focus on the following FOL query sublanguages. *Conjunctive Queries* (CQs), the core database queries a.k.a. Select-Project-Join queries, are conjunctions of atoms whose bound variables are existentially quantified, and whose atoms are of the forms $A(t)$ or $R(t, t')$ with t, t' variables or individuals. *Semi-Conjunctive Queries* (SCQs) are conjunctions of disjunctions of single-atom CQs with the same arity, where the atom is either of the form $A(t)$ or of the form $R(t, t')$ as above; the bound variables of SCQs are also existentially quantified. *Unions of* CQ*s* (UCQs) are *disjunctions* of CQs with same arity; similarly *Unions of* SCQ*s* (USCQs) are *disjunctions* of SCQs with same arity. Finally, *Joins of* UCQ*s* (JUCQs) are *conjunctions* of UCQs; similarly *Joins of* USCQ*s* (JUSCQs) are *conjunctions* of USCQs. All the abovementioned query languages are easily translated into SQL and thus can be evaluated by an RDBMS.

Notations. We write a FOL query $q(\bar{x}) \leftarrow \phi(\bar{x})$ and we refer to it by its *name* q or its *head* $q(\bar{x})$ when we also need to refer to its free variables; $\phi(\bar{x})$ is the *body* of q. In the queries we consider, existential variables in $\phi(\bar{x})$ are not explicitly quantified; they correspond to those which are not in $\bar{x}$. Further, unless otherwise specified, we systematically use q to refer to the input CQ, $a_1, \ldots, a_n$ to designate the atoms in the body of q, $\mathcal{T}$ to designate a DL-Lite$_\mathcal{R}$ TBox, and $\mathcal{A}$ for the ABox.

FOL-reducibility of data management. DL-Lite$_\mathcal{R}$ has been designed so that the core data management tasks of deciding KB consistency and of query answering are FOL-*reducible* [1]. This property allows reducing a data management task over a KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ to the evaluation against $\mathcal{A}$ *only* of a FOL query computed using $\mathcal{T}$ *only*.

The interest of FOL-reducibility is to perform a data management task in two separate steps: a first reasoning step that begets the FOL query, and a second step which evaluates that query in a purely relational fashion, i.e., within an RDBMS after translation to SQL. These systems benefit from decades of research and development invested in optimizing data stores, and query evaluation engines; thus, they hold great potential for efficient FOL query evaluation.

3 Query answering optimization framework

In this section, we recall the notion of FOL reformulation which underpins FOL reducibility of query answering (Section 3.1). Then, for formalisms enjoying FOL reducibility of query answering, e.g., some DLs or some existential rules and Datalog± fragments, we lay the principles of query answering based on *query covers*, which define a space of alternative equivalent FOL reformulations for a given query (Section 3.2), wherein we can search for a reformulation leading to the minimal evaluation cost in an RDBMS (Section 3.3).

3.1 FOL reducibility of query answering

FOL reducibility of query answering relies on FOL *reformulations* of queries.

Definition 1 (FOL reformulation)**.** *A* FOL reformulation $q^{\texttt{FOL}}$ *of* q *w.r.t.* $\mathcal{T}$ *is a* FOL *query such that* $ans(q, \langle \mathcal{T}, \mathcal{A} \rangle) = ans(q^{\texttt{FOL}}, \langle \emptyset, \mathcal{A} \rangle)$ *for any* $\mathcal{T}$*-consistent ABox* $\mathcal{A}$*.*

Observe that FOL reformulation reduces CQ query answering against a KB to FOL query answering against a database storing the KB's assertions. The latter can hence be delegated to the query evaluation engine of an RDBMS.

3.2 Cover-based query answering

RDBMS query optimizers consider a set of *evaluation alternatives* (a.k.a. logical and physical plans), and select the one minimizing a *cost estimation function*. Since the number of alternatives is in $\mathcal{O}(2^n \times n!)$ for a CQ of n atoms [7], modern optimizers rely on heuristics to explore only a few alternatives; this works (very) well for *small-to-moderate size* CQs. However, FOL reformulations go *beyond* CQ*s* in general, and may be *extremely large* (see Section 6), leading the RDBMS to perform poorly.

To work around this limitation, we introduce the cover-based query answering technique to define a space of equivalent FOL reformulations of a CQ. A *cover* defines how the query is split into subqueries, that may overlap, called *fragment queries*, such that substituting each subquery with its FOL reformulation (obtained from any state-of-the-art technique) and joining the corresponding (reformulated) subqueries, *may* yield a FOL reformulation for the query to answer. Indeed, as we shall see soon, not every cover of a query leads to a FOL reformulation. However, every cover which does, yields an alternative *cover-based* FOL *reformulation* of the original query. Crucially for our problem, *a smart cover choice may lead to a cover-based reformulation whose evaluation is more efficient*. Thus, the cover-based technique amounts to *circumventing* the difficulty of modern RDBMSs to efficiently evaluate FOL reformulations in general.

Definition 2 (CQ cover)**.** *A* cover *of a query* q*, whose atoms are* $\{a_1, \ldots, a_n\}$*, is a set* $C = \{f_1, \ldots, f_m\}$ *of non-empty subsets of atoms of* q*, called* fragments*, such that (i)* $\bigcup_{i=1}^{m} f_i = \{a_1, \ldots, a_n\}$ *and (ii) no fragment is included into another.*

Definition 3 (Fragment queries of a CQ)**.** *Let* $C = \{f_1, \ldots, f_m\}$ *be a cover of* q*. A* fragment query $q_{|f_i, 1 \leq i \leq m}$ *of* q *w.r.t.* C *is the subquery whose body consists of the atoms in* f_i *and whose free variables are the free variables* $\bar{x}$ *of* q *appearing in the atoms of* f_i*,* plus *the existential variables in* f_i *that are shared with another fragment* $f_{j, 1 \leq j \leq m, j \neq i}$*, i.e., on which the two fragments join.*

Definition 4 (Cover-based FOL reformulation)**.** *Let* $C = \{f_1, \ldots, f_m\}$ *be a cover of* q*, and* $q^{\texttt{FOL}}(\bar{x}) \leftarrow \bigwedge_{i=1}^{m} q^{\texttt{FOL}}_{|f_i}$ *a* FOL *query, where* $q^{\texttt{FOL}}_{|f_i}$*, for* $1 \leq i \leq m$*, is a* FOL *reformulation w.r.t.* $\mathcal{T}$ *of the fragment query* $q_{|f_i}$ *of* q*.*

$q^{\texttt{FOL}}$ *is* a cover-based FOL reformulation of q w.r.t. $\mathcal{T}$ and C *if it is a* FOL *reformulation of* q *w.r.t.* $\mathcal{T}$*.*

3.3 Cover-based query answering optimization

We assume given a *query cost estimation function* ϵ which, for any FOL query q, returns the cost of evaluating it through an RDBMS storing the ABox. Thus, ϵ reflects the operations (selections, projections, joins, unions, scans etc) applied on the ABox to compute the results of a q^{FOL} reformulation. The cost estimation ϵ also accounts for the effort needed to join the reformulated fragment query results. Note that any RDBMS hosting the ABox includes such a cost estimation function, accessible for instance using the SQL `explain` directive. While the RDBMS optimizer explores several alternatives for evaluating a given FOL query, ϵ reflects the most efficient alternative found. One can also estimate costs outside the engine, using well-known formulas [8], as in e.g., [9].

Problem 1 (Optimization problem)**.** *Given a* CQ q *and a KB* $\mathcal{K}$*, the* cost-driven cover-based query answering problem *consists of finding a cover-based reformulation of* q *based on* $\mathcal{K}$ *with lowest (estimated) evaluation cost.*

4 Cover-based query answering in DL-Lite

We now apply our general optimization framework to the lightweight DL-Lite$_\mathcal{R}$ description logic, for which not all covers lead to FOL query reformulations. We therefore provide a condition ensuring that a cover is *safe for query answering*, i.e., it leads to a FOL query reformulation.

The literature provides techniques for computing FOL reformulations of a CQ in settings related to DL-Lite$_\mathcal{R}$. They produce (i) a UCQ w.r.t. a DL-Lite$_\mathcal{R}$ TBox, e.g., [1, 2, 3, 5, 6], or extensions thereof using existential rules [10] or Datalog$\pm$ [11, 12], (ii) a USCQ [13] w.r.t. a set of existential rules generalizing a DL-Lite$_\mathcal{R}$ TBox, and (iii) a set of alternative equivalent JUCQs [9] w.r.t. an RDF database [14], whose RDF Schema constraints is the following subset of the DL-Lite$_\mathcal{R}$ TBox ones: $A_1 \sqsubseteq A_2$, $R_1 \sqsubseteq R_2$, $\exists R \sqsubseteq A$ and $\exists R^- \sqsubseteq A$.

Next, we devise a cover-based technique for computing a *set of* JUCQ *or of* JUSCQ *reformulations* of a CQ, when querying DL-Lite$_\mathcal{R}$ KBs. This extends [9] from RDF databases to DL-Lite$_\mathcal{R}$, and builds on the aforementioned state-of-the-art on UCQ and USCQ reformulations.

We first recall the pioneering CQ-to-UCQ reformulation technique for DL-Lite$_\mathcal{R}$ [1], on which we rely on in order to establish our results. These results extend to any other FOL reformulation techniques for DL-Lite$_\mathcal{R}$, e.g., optimized CQ-to-UCQ or CQ-to-USCQ reformulation techniques, because they produce equivalent FOL queries.

The technique of [1] relies on two operations: *the specialization of an atom into another* through backward chaining on a TBox constraint, and *the specialization of two atoms into one* through their most general unifier. The technique consists in exhaustively applying these two operations to the input CQ, each operation generating a CQ subsumed by the input CQ. The same is then recursively applied on the subsumed CQs thus obtained, until reaching a fixpoint for the set of generated CQs. The finite union of the input CQ and generated ones forms the UCQ reformulation of the input CQ w.r.t. the TBox. The next example illustrates this.

Example 1 (CQ-to-UCQ reformulation technique)**.** *Consider the KB* $\mathcal{K} = \langle \mathcal{T} = \{B \sqsubseteq \exists R', R' \sqsubseteq R\}, \mathcal{A} = \{A(a), B(a)\}\rangle$ *and the query* $q(x) \leftarrow A(x) \wedge R(x,y) \wedge R'(z,y)$*, whose only answer is* a*. The* UCQ *reformulation of* q *is*

$$q^{\text{UCQ}}(x) \leftarrow \begin{array}{l} (A(x) \wedge R(x,y) \wedge R'(z,y)) \\ \vee\ (A(x) \wedge R'(x,y) \wedge R'(z,y)) \\ \vee\ (A(x) \wedge R'(x,y)) \\ \vee\ (A(x) \wedge B(x)) \end{array}$$

where the first disjunct is q*'s body, the second is obtained from the first by specializing the atom* $R(x,y)$ *through a backward-chaining application of* $R' \sqsubseteq R \in \mathcal{T}$*, the third is obtained from the second through*

the most general unifier of $R'(x, y)$ and $R'(z, y)$, and the fourth results from the third by specializing $R'(x, y)$ through the backward-chaining application of $B \sqsubseteq \exists R' \in \mathcal{T}$.

As stated in Definition 4, a query cover may not lead to a cover-based query reformulation. This is true even in the lightweight DL-Lite$_\mathcal{R}$ setting, as shown below, while it is not in the simpler RDF setting of [9].

Example 2 (Covers for JUCQ reformulation)**.** *Consider the same TBox $\mathcal{T}$ and query q as in the preceding example. Following Definition 4, the FOL query obtained from q, its cover $C_1 = \{\{A(x), R(x, y)\}, \{R'(z, y)\}\}$, $\mathcal{T}$ and the CQ-to-UCQ reformulation technique is the JUCQ: $q^{\text{JUCQ}}(x) \leftarrow q_1^{\text{UCQ}}(x, y) \wedge q_2^{\text{UCQ}}(y)$, where $q_1^{\text{UCQ}}(x, y) \leftarrow (A(x) \wedge R(x, y)) \vee (A(x) \wedge R'(x, y))$ and $q_2^{\text{UCQ}}(y) \leftarrow R'(z, y)$. $q^{\text{JUCQ}}(x)$ is* not *a reformulation of q w.r.t. $\mathcal{T}$: $ans(q^{\text{JUCQ}}, \langle \emptyset, \mathcal{A} \rangle) = \emptyset \neq \{a\}$.*

The two examples above show that the cover used in Example 2 prevents the unification of $R'(x, y)$ and $R'(z, y)$, as these two atoms are split across two subqueries that are reformulated separately, while this unification is required for the correctness (actually the completeness) of the technique: it is the only way to trigger the use of the constraint $B \sqsubseteq \exists R'$.

More generally, given an input CQ and a TBox, each unification necessary for the correctness of the CQ-to-UCQ technique defines a set of input CQ atoms that cannot be separated within the cover: those which beget this unification.

We therefore provide a sufficient condition for a cover to be *safe for query answering*, i.e., to lead to a cover-based FOL reformulation. The main idea for this condition is to have a cautious approximation of the *query atoms* which are interdependent w.r.t. reformulation, i.e., which (directly or after specialization) unify through the CQ-to-UCQ reformulation technique, and keep them in the same cover fragment.

Only atoms that have the same predicate may unify. Thus, we identify for each *predicate* (i.e., concept or role name) appearing in a query atom, the set of all TBox predicates in which the atom may turn through some sequence of atom specializations, i.e., backward chaining and/or unifications, the two operations applied by the technique of [1]. We say the query atom predicate *depends* on such TBox predicates, and we capture them in a conservative fashion, based on the syntax of the TBox constraints, in the following recursive definition.

Definition 5 (Concept and role dependencies w.r.t. a TBox)**.** *Given a TBox $\mathcal{T}$, a concept or role name N* depends w.r.t. $\mathcal{T}$ on *the set of concept and role names defined as the fixpoint of:*
$dep^0(N) = \{N\}$
$dep^n(N) = dep^{n-1}(N)$
$\quad \cup \{cr(Y) \mid Y \sqsubseteq X \in \mathcal{T}$ *and* $cr(X) \in dep^{n-1}(N)\}$
where the cr function returns the concept or role name involved in any DL-Lite$_\mathcal{R}$ concept or role provided as input.

Definition 6 (Safe cover for query answering)**.** *A cover C of q is* safe for query answering w.r.t. $\mathcal{T}$ *(or* safe *in short) iff it is a partition of q's atoms such that two atoms whose predicates depend on a common concept or role name w.r.t. $\mathcal{T}$ are in a same fragment.*

Note that while Definition 6 requires covers to be partitions, we will relax this restriction in Section 5.1.

Example 3 (Covers for JUCQ reformulation (cont.))**.** *The FOL query obtained with the safe cover $C_2 = \{\{A(x)\}, \{R(x, y), R'(z, y)\}\}$ is the JUCQ* reformulation $q^{\text{JUCQ}}(x) \leftarrow q_1(x) \wedge q_2(x)$*, where $q_1(x) \leftarrow A(x)$ and $(q_2(x) \leftarrow (R(x, y) \wedge R'(z, y)) \vee (R'(x, y) \wedge R'(z, y)) \vee (R'(x, y)) \vee (B(x))$. Observe that $ans(q^{\text{JUCQ}}, \langle \emptyset, \mathcal{A} \rangle) = \{a\} = ans(q, \langle \mathcal{T}, \mathcal{A} \rangle)$.*

Theorem 1 (Cover-based query answering)**.** *Applying Definition 4 on a safe cover C of q w.r.t. $\mathcal{T}$, and any CQ-to-UCQ (resp. CQ-to-USCQ) reformulation technique, yields a cover-based JUCQ (resp. JUSCQ) reformulation q^{FOL} of q w.r.t. $\mathcal{T}$.*

The proof follows from that of the CQ-to-UCQ reformulation technique of [1]. Soundness, i.e., $ans(q^{\texttt{FOL}}, \langle \emptyset, \mathcal{A} \rangle) \subseteq ans(q, \langle \mathcal{T}, \mathcal{A} \rangle)$, directly follows from the construction of $q^{\texttt{FOL}}$, based on Definition 4, together with the soundness of the CQ-to-UCQ reformulation. Completeness, i.e., $ans(q, \langle \mathcal{T}, \mathcal{A} \rangle) \subseteq ans(q^{\texttt{FOL}}, \langle \emptyset, \mathcal{A} \rangle)$, follows from the fact that the CQ-to-UCQ reformulations of the fragment queries induced by a safe cover are independent from each other, since atoms from distinct fragments are never unified by the CQ-to-UCQ reformulation of the input CQ. In turn, this holds because of the definition of a safe cover, allowing atoms in different fragments only if they depend on no common predicate. The proof then relies on the completeness of the CQ-to-UCQ reformulation.

It is worth noting that these results directly *extend* to the use of any CQ-to-UCQ or CQ-to-USCQ reformulation technique for DL-Lite$_\mathcal{R}$ as, for any CQ and TBox, the FOL queries they compute are equivalent to that of [1].

Finally, remark that the trivial one-fragment cover (comprising all query atoms) is *always safe*; in this case, our technique reduces to the CQ-to-UCQ or CQ-to-USCQ one.

5 Cover-based query optimization in DL-Lite

We study next the query answering optimization problem of Section 3.3 for DL-Lite$_\mathcal{R}$. We analyze its optimization space in Section 5.1 and provide search algorithms in Section 5.2.

5.1 Optimization space

We start by identifying a particularly interesting safe cover:

Definition 7 (Root cover)**.** *We term* root cover *for a query q and TBox $\mathcal{T}$ the cover C_{root} obtained as follows. Start with a cover C_1 where each atom is alone in a fragment. Then, for any pair of fragments $f_1, f_2 \in C_1$ and atoms $a_1 \in f_1, a_2 \in f_2$ such that there exists a TBox predicate on which a_1 and a_2 depend, create a fragment $f' = f_1 \cup f_2$ and a new cover $C_2 = (C_1 \setminus \{f_1, f_2\}) \cup \{f'\}$. Repeat the above until the cover is stationary; this is the root cover, denoted C_{root}.*

It is easy to see that C_{root} is safe, and does not depend on the order in which the fragments are considered.

Example 4 (Root cover)**.** *Recall the query and TBox from Example 1. The root cover C_{root} in this case is the cover C_2 from Example 3.*

Proposition 1 (Minimality of C_{root} fragments)**.** *Let C_{root} be the root cover for q and $\mathcal{T}$, and C be another safe cover. For any fragment $f \in C_{\text{root}}$, and atoms $a_i, a_j \in f$, there exists a fragment $f' \in C$ such that $a_i, a_j \in f'$, in other words: any pair of atoms together in C_{root} are also together in C.*
From the above we derive easily:

Theorem 2 (Safe cover space)**.** *Let C be a safe cover and $f \in C$. Then, f is the union of a set of fragments from C_{root}.*

It follows that the set of all safe covers for a query q form a *lattice* $\mathcal{L}_q$ whose precedence relationship is denoted $\prec$, where $C_1 \prec C_2$ if each fragment of C_2 is a union of some fragments of C_1. The lattice has a sink node (the single-fragment partition), and a root node, which is C_{root}.

Beyond Safe Covers. A dependency-rich TBox leads to few, large fragments in C_{root}, and thus to a relatively small lattice. Below, we present a set of covers which are not safe (in particular, they are not partitions), yet from which we still derive FOL reformulations of the query; this enlarges our space of alternatives and thus gives more optimization opportunities.

An *extended fragment* $f\|g$ of q is a pair of atom sets such that $f = \{a_1, \ldots, a_k\}$ is a set of atoms from q, and $g \subseteq f$. When $f = g$, extended fragments coincide with simple fragments, thus we generalize query covers (Definition 2) to consider that they only consist of extended fragments, while preserving that no fragment must be contained in another.

To an extended fragment we associate:

Definition 8 (Extended fragment query of a CQ)**.** *Let $f\|g \in C$ be an extended fragment of q. The* extended fragment query $q_{|f\|g}$ *of q w.r.t. C is the subquery whose body consists of the atoms in f, and whose free variables are the free variables of q appearing in the atoms of g, plus the variables appearing in an atom of g that are shared with some atom in g', for some fragment $f'\|g'$ of C.*

In an extended fragment query, atoms from $f \setminus g$ only *reduce (filter)* the answers, without adding variables to the head.

Given a cover of extended fragments, the *extended cover-based reformulation of a query q* is the FOL query:

$$q^e(\bar{x}) \leftarrow \bigwedge_{i=1}^{m} q^{\texttt{FOL}}_{|f_i\|g_i}$$

Example 5 (Extended cover-based reformulation)**.** *Recall the query and KB from Example 1. Let $f_0 = \{A(x)\}$ and $f_1 = \{R(x,y), R'(z,y)\}$ be the two fragments of the root cover C_{root} for this example, $f_2 = \{A(x), R(x,y)\}$, and the cover $C_4 = \{f_1\|f_1, f_2\|f_0\}$. In this case, $q_{|f_1,\|f_1}(x) \leftarrow R(x,y) \wedge R'(z,y)$, $q_{|f_2\|f_0}(x) = A(x) \wedge R(x,y)$, and the extended cover-based reformulation corresponding to C_4 is: $q^e(x) \leftarrow ((R(x,y) \wedge R'(z,y)) \vee R'(x,y) \vee B(x)) \wedge ((A(x) \wedge R(x,y)) \vee (A(x) \wedge R'(x,y)) \vee (A(x) \wedge B(x)))$ which is indeed a reformulation of q.*

The introduction of extra atoms in extended fragments is reminiscent of the database optimization known as *semijoin reducer* [15], which can be summarized as follows. Consider a CQ $q(x) \leftarrow R(x,y) \wedge S(y,z)$; the standard relational algebra expression used to evaluate q is $R(x,y) \bowtie_y S(y,z)$, where $\bowtie_y$ denotes the join operator (on y) of the two relations. The semijoin reducer technique consists, instead, of evaluating q by the relational algebra expression: $(R(x,y) \ltimes_y \pi_y(S(y,z))) \bowtie_y S(y,z)$ where $\pi_y(S(y,z))$ is the first projection of S, and $\ltimes_y$ denotes the *semijoin* binary relational operator, returning every tuple from the left-hand side input that joins with the right-hand input. Intuitively, the semijoin filters ("reduces") the R relation to only those tuples having a match in S. If there are few distinct values of y in S, $\pi_y(S(y,z))$ is small, thus it can be loaded in memory and the $\ltimes_y$ operator can be evaluated very efficiently. Further, if only few R tuples survive the $\ltimes_y$, the cost of the $\bowtie_y$ is reduced, given that there are less tuple comparisons to be made. While the benefits of semijoins are well-known, there are many ways to introduce them in a given query, further increasing the space of alternative plans (join orders etc) that an optimizer would have to consider. Given that this space is already very large for complex queries such as the FOL reformulations of CQs, in practice, we noted that RDBMS optimizers do not explore semijoin options. *Extended fragments mitigate this problem by introducing semijoin reducers in the* FOL *query given to the RDBMS.*

We now describe an *extended search space* $\mathcal{E}_q$ of covers for a query q. First, any $\mathcal{L}_q$ cover is also in $\mathcal{E}_q$. Second, from any cover $C \in \mathcal{E}_q$ and fragment $f||g \in C$, one can derive a cover C' as follows: (I) pick a query atom $a \notin f$, which shares a variable with an atom in f; (II) build the new extended fragment $f'||g$ defined by $f \cup \{a\}||g$; let $C' = (C \setminus \{f||g\}) \cup \{f'||g\}$. Since the newly added atom was already in a fragment of C_{root}, C' is not a partition.

Theorem 3 ($\mathcal{E}_q$ cover-based query answering)**.** *Applying Definition 4 together with Definition 8, on a* CQ *q, a TBox $\mathcal{T}$, a cover C from $\mathcal{E}_q$, and any* CQ*-to-*UCQ *(resp.* CQ*-to-USCQ) reformulation technique, yields a* JUCQ *(resp.* JUSCQ*) reformulation of q w.r.t. $\mathcal{T}$.*

The proof derives from that of Theorem 1. It relies on the fact that, given a safe cover $C = \{g_1, \ldots, g_m\}$ of q and an extended cover $C' = \{f_1||g_1, \ldots, f_m||g_m\}$ of q, the queries $q(\bar{x}) \leftarrow \bigwedge_{i=1}^{m} q_{|g_i}$ and $q^e(\bar{x}) \leftarrow \bigwedge_{i=1}^{m} q_{|f_i||g_i}$ are equivalent, though each $q_{|g_i}$ subsumes $q_{|f_i||g_i}$. Intuitively, q^e is obtained from q by duplicating atoms already present in q, thus q^e only adds redundancy w.r.t. q, and thus remains equivalent to it.

For the same reason as in the case of simple covers (Section 4), these results directly *extend* to the use of any CQ-to-UCQ or CQ-to-USCQ reformulation technique for DL-Lite$_\mathcal{R}$.

5.2 Cost-based cover search algorithms

Our first algorithm, **EC-DL (Exhaustive Covers)**, starts from C_{root} and builds all $\mathcal{L}_q$ covers by fusing fragments, and all $\mathcal{E}_q$ covers by adding atoms, as explained previously. The second one, **GC-DL (Greedy Covers)**, explores $\mathcal{E}_q$ partially, in greedy fashion. It uses an *explored cover set* initialized with $\{C_{\text{root}}\}$. GC-DL picks from this set a cover C inducing a q^{FOL} reformulation with minimum evaluation cost, and attempts to build from it a cover C', through fragment fusion or atom addition. GC-DL only adds C' to the explored set if $\epsilon(C') < \epsilon(C)$, thus it only explores a small part of the search space. Both algorithms return a cover-based reformulation with the minimum estimated cost w.r.t. the explored space.

When fusing two fragments, ϵ decreases if the resulting fragment is more selective than the two fragments it replaces. Therefore, the RDBMS may find a more efficient way to evaluate the query of this fragment, and/or its result may be smaller, making the evaluation of q^{FOL} based on the new cover C' faster. When adding an atom to an extended fragment, ϵ decreases if the conditions are met for the semijoin reducer to be effective (Section 5.1). Such performance benefits can be very significant, as we show below.

6 Experimental evaluation

We implemented our cover-based approach in Java 7, on top of PostgreSQL v9.3.2, running on an 8-core Intel Xeon (E5506) 2.13 GHz machine with 16GB RAM, using Mandriva Linux r2010.0. We used the LUBM$^{\exists}_{20}$ DL-Lite$_\mathcal{R}$ TBox and associated EUDG data generator [16]: LUBM$^{\exists}_{20}$ consists of 34 roles, 128 concepts and 212 constraints; the generated ABox comprises 15 million facts.

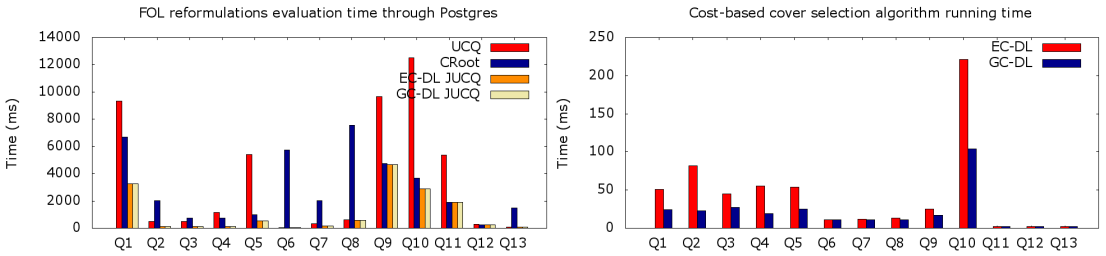
We store the ABox in a binary table for each role and a unary one for each concept; as customary in efficient Semantic Web data management systems, e.g., [17], each identifier or literal occuring in a fact is encoded into an integer prior to storing them in the RDBMS[1]. Also in keeping with the best practices in efficient Semantic Web stores [18, 17], each concept table is indexed, and similarly each role table is indexed by each of its two attributes. The TBox and predicates dependencies are stored in memory. For our experiments, we relied on a CQ-to-UCQ reformulation tool, namely RAPID [5] (recall that any CQ-to-USCQ reformulation technique could have been used instead). For the cost estimation function ϵ, we relied on Postgres' own estimation, obtained using the `explain` directive[2].

We devised a set of 13 CQs against this KB; the queries have between 2 and 10 atoms, with an average of 5.77 atoms. Their UCQ reformulations have 35 to 667 CQs, 290.2 on average. We depict below Q_9, one of the most complex, whose UCQ reformulation is a union of 368 CQs:

$$Q_9(n, e) \leftarrow \text{Student}(x) \wedge \text{TakesCourse}(x, c) \wedge \text{Advisor}(x,a) \wedge \text{MemberOf}(x,\text{"http://dep0.univ0.edu"}) \wedge \text{Phone}(x, \text{"xxx"}) \wedge \text{MemberOf}(a, \text{"http://dep0.univ0.edu"}) \wedge \text{TeacherOf}(p, c) \wedge \text{Email}(p, e) \wedge \text{ResearchInterest}(a, \text{"res7"}) \wedge \text{Name}(c, n)$$

[1] Within an RDBMS, comparisons on fixed-length data items, e.g., integers, are faster than on variable-length items, e.g., strings. Thus, replacing each identifier or constant by an integer and evaluating queries on integer tables is faster, even when including the time to decode the integer results. It also reduces (drastically) the size of the database, since integer codes are very compact.

[2] See http://www.postgresql.org/docs/9.1/static/sql-explain.html.



(a) Evaluation time for JUCQ reformulations. (b) Running time of the cover-based technique.

where quoted strings stand for constants produced by the data generator (shortened for the presentation).

Figure 1a depicts the evaluation time through Postgres, of four FOL reformulations: (i) the UCQ produced by RAPID [5]; (ii) the JUCQ reformulation based on C_{root}; (iii) the JUCQ reformulation corresponding to the best-performing cover found by our algorithm EC-DL, and (iv) the JUCQ reformulation based on the best-performing cover found by GC-DL. First, the figure shows that fixed FOL reformulations are not efficiently evaluated, e.g., UCQ for Q_1, Q_5 and Q_9-Q_{11}, and the one based on C_{root} for Q_6-Q_8 and Q_{13}. This poor performance correlates with the large size of the UCQ reformulation: such very large unions of CQs are very poorly handled by current RDBMS optimizers, which are designed and tuned for small CQs. This finding is confirmed in [9] for two other leading commercial RDBMSs, thus it is not specific to the Postgres system. Second, the reformulation based on the cover returned by EC-DL is always more efficient than UCQ reformulation (more than one order of magnitude for Q_5), respectively, C_{root}-based reformulation (up to a factor of 230 for Q_6). Third, in our experiments, the GC-DL-chosen cover leads to a JUCQ reformulation as efficient as the EC-DL one, demonstrating that even a partial, greedy cover search leads to good performance. (In general, though, the optimality of GC-DL is not guaranteed.)

For Q_9, C_{root} is $\{\{a_5\}, \{a_8\}, \{a_9\}, \{a_{10}\}, \{a_1, a_2, a_3, a_4, a_6, a_7\}\}$ (where the atoms are ordered as shown above), while the cover chosen by EC-DL and GC-DL fuses the largest fragment of C_{root} with $\{a_{10}\}$. For Q_7 and Q_9-Q_{13}, the best cover we found is safe; for all the others, this is not the case, confirming the interest of extended fragments within covers.

Figure 1b depicts the running time of the EC-DL and GC-DL algorithms, which can be seen as the overhead of our cover-based technique. The time is very small, between 2 ms (Q_{11}-Q_{13}, with just 2 atoms) and 221 ms (EC-DL on Q_{10}, of 10 atoms). The time is higher for more complex queries, but these are precisely the cases where our techniques are most beneficial, e.g., for Q_{10}, EC-DL runs in less than 2% of the time to evaluate the UCQ reformulation, while the cover we recommend is more than 4 times faster than UCQ. As expected, GC-DL is faster than EC-DL due to the exploration of less covers. Together, Figure 1a and 1b confirm the benefits and practical interest of our cost-based cover search.

7 Related work and conclusion

We proposed a novel framework for DLs enjoying FOL reducibility of query answering, in order to have a space of alternative FOL reformulations to evaluate through an RDBMS. This space is crucial, as it allows choosing the FOL reformulation most efficiently evaluated by the RDBMS. We applied this framework to the DL-Lite$_\mathcal{R}$ description logic underpinning the W3C's OWL2 QL standard for the Semantic Web, and experimentally demonstrated its performance benefits.

Our approach departs from the literature focused on a *single* FOL query reformulation, where optimization reduces to *producing fast a reformulation as compact as possible*. [3, 5, 6, 10, 11, 12] consider DL-Lite$_\mathcal{R}$, existential rules and Datalog±. [13] studies CQ-to-USCQ reformulation for existential rules encompassing DL-Lite$_\mathcal{R}$; USCQ reformulations are shown to perform overall better than UCQ ones in an RDBMS. We build on this work to devise CQ-to-JUCQ and CQ-to-JUSCQ reformulation techniques, fur-

ther used for cost-based query answering optimization. Our semijoin-inspired technique goes *against* minimization, as it *adds* redundant atoms, but does so with the help of a cost model only when this improves performance. Finally, [9] is an instance of our general framework for the simple case of RDF databases: any cover leads to a reformulation, while it is not the case in general, like in DL-Lite$_\mathcal{R}$ as we have shown. Other approaches for DL-Lite$_\mathcal{R}$ do not rely on FOL reducibility, e.g., [4, 19] produce Datalog reformulations.

We plan to extend our framework to efficient query answering *using views*, i.e., through a set of predefined CQs, which is typical of the *information integration* and *database query optimization* settings [20].

Acknowledgements This work has been partially funded by the *Programme Investissement d'Avenir* Datalyse project and the ANR PAGODA project.

References

[1] 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," *JAR*, vol. 39, no. 3, pp. 385–429, 2007.

[2] N. Abdallah, F. Goasdoué, and M. Rousset, "DL-LITER in the light of propositional logic for decentralized data management," in *IJCAI*, 2009.

[3] H. Pérez-Urbina, I. Horrocks, and B. Motik, "Efficient query answering for OWL 2," in *ISWC*, 2009.

[4] R. Rosati and A. Almatelli, "Improving query answering over DL-Lite ontologies," in *KR*, 2010.

[5] A. Chortaras, D. Trivela, and G. B. Stamou, "Optimized query rewriting for OWL 2 QL," in *CADE*, 2011.

[6] T. Venetis, G. Stoilos, and G. B. Stamou, "Incremental query rewriting for OWL 2 QL," in *Description Logics*, 2012.

[7] K. Ono and G. M. Lohman, "Measuring the complexity of join enumeration in query optimization," in *VLDB*, 1990.

[8] R. Ramakrishnan and J. Gehrke, *Database management systems*. McGraw-Hill, 2003.

[9] D. Bursztyn, F. Goasdoué, and I. Manolescu, "Optimizing reformulation-based query answering in RDF," in *EDBT*, 2015.

[10] M. König, M. Leclère, M. Mugnier, and M. Thomazo, "A sound and complete backward chaining algorithm for existential rules," in *RR*, 2012.

[11] R. D. Virgilio, G. Orsi, L. Tanca, and R. Torlone, "NYAYA: A system supporting the uniform management of large sets of semantic data," in *ICDE*, 2012.

[12] G. Gottlob, G. Orsi, and A. Pieris, "Query rewriting and optimization for ontological databases," *ACM Trans. Database Syst.*, vol. 39, no. 3, p. 25, 2014.

[13] M. Thomazo, "Compact rewriting for existential rules," *IJCAI*, 2013.

[14] F. Goasdoué, I. Manolescu, and A. Roatiş, "Efficient query answering against dynamic RDF databases," in *EDBT*, 2013.

[15] P. A. Bernstein and D. W. Chiu, "Using semi-joins to solve relational queries," *J. ACM*, vol. 28, no. 1, 1981.

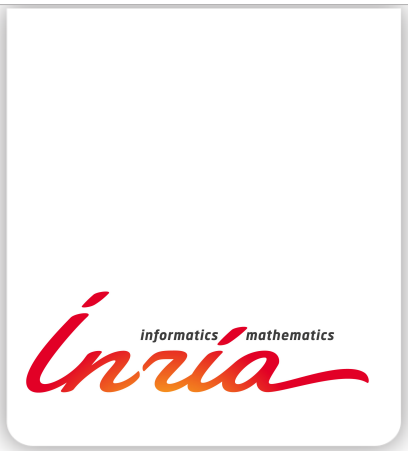
[16] C. Lutz, I. Seylan, D. Toman, and F. Wolter, "The combined approach to OBDA: taming role hierarchies using filters," in *ISWC*, 2013.

[17] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *VLDBJ*, vol. 19, no. 1, pp. 91–113, 2010.

[18] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple indexing for Semantic Web data management," *PVLDB*, 2008.

[19] G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, R. Rosati, M. Ruzzi, and D. F. Savo, "MASTRO: A reasoner for effective ontology-based data access," in *International Workshop on OWL Reasoner Evaluation (ORE-2012)*, 2012.

[20] A. Y. Halevy, "Answering queries using views: A survey," *VLDBJ*, vol. 10, no. 4, pp. 270–294, 2001.



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399