HAL
open science

Efficient query answering in the presence of DL-LiteR constraints

Damian Bursztyn, François Goasdoué, Ioana Manolescu

To cite this version:

Damian Bursztyn, François Goasdoué, Ioana Manolescu. Efficient query answering in the presence of DL-LiteR constraints. [Research Report] RR-8714, INRIA Saclay; INRIA. 2016. hal-01143498v4

HAL Id: hal-01143498

https://inria.hal.science/hal-01143498v4

Submitted on 22 Aug 2016

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

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

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

Damian Bursztyn, François Goasdoué, Ioana Manolescu

RESEARCH REPORT
N° 8714
Août 2016
Project-Teams CEDAR

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 CEDAR

Research Report n° 8714 — version 4 — initial version Août 2016 — revised version Août 2016 — 43 pages

Abstract: In the presence of an ontology, query answers must reflect not only data explicitly present in the database, but also implicit data, which holds due to the ontology, even though it is not present in the database. A large and useful set of ontology languages enjoys FOL *reducibility of query answering*, that is: answering a query can be reduced to evaluating a certain first-order logic (FOL) formula (obtained from the query and ontology) against only the explicit facts.
We present a *novel query optimization framework for ontology-based data access settings enjoying* FOL *reducibility*. Our framework is based on 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 apply this framework to the DL-Lite$_{\mathcal{R}}$ Description Logic underpinning the W3C's OWL2 QL ontology language, and demonstrate through experiments its performance benefits when two leading SQL systems, one open-source and one commercial, are used for evaluating the FOL query reformulations.

Key-words: Query answering, DL-lite, query optimization

RESEARCH CENTRE SACLAY – ÎLE-DE-FRANCE

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

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

Résumé : En présence d'une ontologie, les réponses aux requêtes doivent refléter non seulement les données explicitement présentes dans la base de données, mais également les données implicites dues à l'ontologie, qui elles ne sont pas présentes dans la base. Un large éventaille de langages d'ontologie permet de réduire le problème de répondre à une requête à l'évaluation d'une requête de la logique du premier ordre appelée reformulation : répondre à une requête peut se réduire à l'évaluation d'une reformulation (obtenue à partir de la requête et de l'ontologie) uniquement sur les données explicites.

Nous présentons une nouveau cadre d'optimisation de requête dans ce contexte d'accès aux données en présence d'ontologies. Il est fondé sur la recherche au sein d'un ensemble de reformulations possibles, d'une reformulation de coût minimal lorsqu'évaluée par un système de gestion de bases de données relationnelles. Nous appliquons ce cadre à la logique de description DL-Lite$_{\mathcal{R}}$ sur laquelle se fonde le langage OWL2 QL du W3C. Pour cette logique, nous montrons au travers d'une évaluation expérimentale les gains de performances obtenus lorsque deux systèmes SQL renommés, l'un open-source et l'autre commercial, sont utilisés pour l'évaluation des reformulations.

Mots-clés : Répondre à des requêtes, DL-lite, optimisation de requêtes

1 Introduction

Ontology-based data access (OBDA, in short) [23] aims at exploiting a database, i.e., *facts*, on which hold ontological constraints, i.e., *deductive* constraints modeling the application domain under consideration. For instance, an ontology may specify that any author is a human, has a name, and must have authored some papers. Ontological constraints may greatly increase the *usefulness* of a database: for instance, a query asking for all the humans must return all the authors, just because of a constraint stating they are human; one does not need to store a human tuple in the database for each author. The data interpretations enabled by the presence of constraints has made OBDA a technique of choice when modeling complex real-life applications. For instance, in the medical domain, Snomed Clinical Terms (Snomed)[1] is an biomedical ontology providing a comprehensive clinical terminology; the British Department of Health has a roadmap for standardizing medical records across the country, using this ontology etc.

While query answering under constraints is a classical database topic [2], research on OBDA has bloomed recently through the proposal of many languages for expressing ontological constraints, e.g., Datalog$^{\pm}$ [11], Description Logics [4] and Existential Rules [5], or RDF Schema for RDF graphs[2]. OBDA *query answering* is the task of computing the answer to the given query, by taking into account both the facts and the constraints holding on them. In contrast, query evaluation as performed by database servers leads to computing only the answers derived from the data (facts), while ignoring the constraints.

A large and useful class of ontology languages enjoy *first-order logic (*`FOL`*) reducibility (a.k.a. rewritability)* of query answering, e.g., [12, 11, 26]. Query answering under constraints formulated in these languages reduces to the evaluation of the `FOL` *query reformulation*, obtained by compiling the constraints into the query, against the facts alone. Evaluating this `FOL` query in a relational database management system (RDBMS) by translation into SQL against the facts, suffices to compute the complete query answer.

A longstanding issue in reformulation-based query answering is that `FOL` reformulations tend to be very complex queries, involving very large unions (sometimes with hundreds or thousands of union terms) and/or numerous redundant subexpressions. Such queries are very different from the typical ones RDBMS optimizers are tuned for, thus RDBMSs perform poorly at evaluating them. To mitigate this issue, OBDA optimization research has mostly focused on producing `FOL` reformulations where redundancy is avoided as much as possible, e.g., [31, 30, 14, 36, 21, 37, 35, 19].

We present a *more general*, *performance-oriented* approach: we propose a query optimization framework for *any* logical OBDA setting enjoying `FOL` reducibility of query answering. We extend the language of `FOL` reformulations beyond those considered so far in the literature, and investigate *several (equivalent)* `FOL` *reformulations* of a given query, out of which we pick one likely to lead to the best evaluation performance. This contrasts with existing works from the semantic query answering literature (cf. Section 7), which use reformulation languages allowing *single* `FOL` reformulation

[1] https://www.nlm.nih.gov/research/umls/Snomed/snomed_main.html
[2] http://www.w3.org/2001/sw/Specs.html

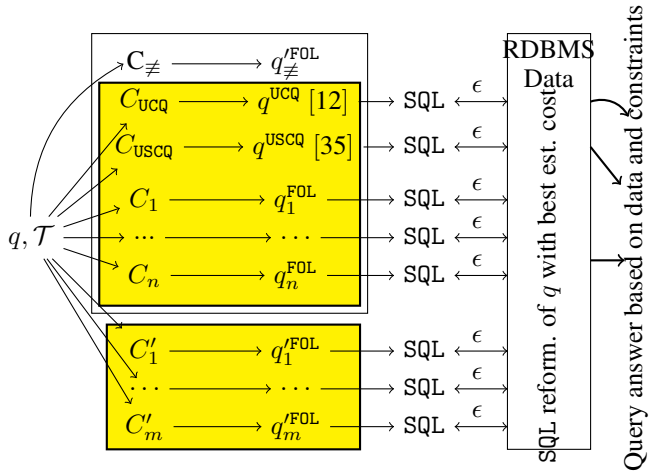


Figure 1: Optimized FOL reformulation approach.

(modulo minimization). Considering a *set of reformulations* and relying on a *cost model* to pick a most efficient one has a very visible impact on the efficiency and feasibility of query answering: indeed, picking the wrong reformulation may cause the RDBMS simply to fail evaluating it (typically due to very lengthy queries), while in other cases it leads to bad performance.

We apply this framework to the DL-Lite$_{\mathcal{R}}$ Description Logic [12] underpinning the popular W3C's OWL2 QL standard for rich Semantic Web applications. Query answering in DL-Lite$_{\mathcal{R}}$ has received significant attention in the literature, notably techniques based on FOL reducibility, e.g., [12, 1, 31, 33, 14, 36].

Contributions. The contributions we bring to the problem of optimizing FOL reducible query answering can be outlined as follows (see Figure 1):

1. For logical formalisms enjoying FOL reducibility of query answering, we provide a general *optimization framework* that reduces query answering to searching among a set of alternative equivalent FOL reformulations, one with minimal evaluation cost in an RDBMS (Section 3). In Figure 1, from the query q and the set of ontological constraints $\mathcal{T}$, we derive first, a space of *query covers*, shown in the top white-background box, and denoted C with some subscripts; from each such cover we show how to derive a FOL query that *may* be a FOL reformulation of q w.r.t. $\mathcal{T}$.

2. We characterize interesting spaces of such alternative equivalent FOL queries for DL-Lite$_{\mathcal{R}}$ (Section 4).

First, we identify a sufficient *safety* condition to pick covers that for sure lead to FOL reformulations of the query. This condition is met by the covers in the top yellow box in Figure 1, and is not met by $C_{\not\equiv}$ above them. Our safe cover space allows considering FOL reformulations encompassing those previously studied in the literature. Second, we introduce a set of *generalized covers* (bottom yellow box in Figure 1) and a generalized cover-based reformulation technique, which always yields FOL query reformulations, oftentimes more efficient than those based on simple covers.

Our approach can be combined with, and helps optimizing, any existing reformulation technique for DL-Lite$_\mathcal{R}$.
3. We then optimize query answering in the setting of DL-Lite$_\mathcal{R}$ by enumerating simple and generalized covers, and *picking a cover-derived* `FOL` *reformulation with lowest estimated evaluation cost* w.r.t. an RDBMS cost model estimation ϵ (denoted by the bidirectional ϵ-labeled arrows in the figure). We provide two algorithms, an exhaustive and a greedy, for this task (Section 5).
4. Evaluating any of our `FOL` reformulations through an RDBMS leads (thick arrows at the right of Figure 1) to the query answer reflecting both the data and the constraints. We demonstrate *experimentally* the effectiveness and the efficiency of our query answering technique for DL-Lite$_\mathcal{R}$, by deploying our query answering technique on top of Postgres and DB2, using several alternative data layouts (Section 6).

From a query processing and optimization perspective, our approach can be seen as belonging to the so-called *strategic optimization* stage introduced in [25] (where application semantics is injected into the query); it is also similar in spirit to the *syntax-level rewrites* performed by optimizers such as Oracle 10g's [3]. We share with [25] the idea of injecting semantics first, and like [3], we use cost estimation to guide our rewrites; a common theme is to rewrite before ordering joins, selecting physical operators etc. From this angle, our contribution can be seen as *a set of alternatives (rewritings) with correctness guarantees and algorithms to guide such rewritings*, for the special class of queries obtained from `FOL` reformulations of `CQ` against ontologies.

In the sequel, Section 2 recalls preliminary notions on knowledge bases and DL-Lite$_\mathcal{R}$. Then, we detail the above contributions. Finally, we discuss related work and conclude in Section 7.

2 Preliminaries

We introduce knowledge bases (Section 2.1), queries and query answering (Section 2.2), and finally position our work from a query optimization perspective, highlighting the hard issues (Section 2.3).

Table 1 summarizes the main notations of this work.

2.1 Knowledge bases

As commonly known, a *knowledge base (KB)* $\mathcal{K}$ consists of a TBox $\mathcal{T}$ (ontology, or axiom set) and an ABox $\mathcal{A}$ (database, or fact set), denoted $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, with $\mathcal{T}$ expressing constraints on $\mathcal{A}$.

Most popular Description Logic dialects [4], and in particular DL-Lite$_\mathcal{R}$ [12], build $\mathcal{T}$ and $\mathcal{A}$ 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* is a set of axioms, whose expressive power is defined by the ontology language. While our work applies to a more general setting (see Section 3), below, we illustrate our discussion on the DL-Lite$_\mathcal{R}$ description logic [12], which is the first order logic foundation of the W3C's OWL2 QL standard for managing semantic-rich Web data. For what concerns

$\mathcal{A}$: Database of facts (Section 2.1)
$\mathcal{T}$: Ontology (semantic rules) (Section 2.1)
$\mathcal{K}$: Knowledge base $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ (Section 2.1)
C_i: Concept (unary relation) (Section 2.1)
R_j: Role (binary relation) (Section 2.1)
`CQ`: Conjunctive query (Section 2.2)
`UCQ`: Union of conjunctive queries (Section 2.2)
`JUCQ`: Join of a set of `UCQ`s (Section 2.2)
$q_{|f_i}$: Fragment query of a `CQ` (Definition 2)
$dep(N)$: Concepts and role names on which N depends (Definition 4)
C_{root}: Root cover (Definition 6)
$\mathcal{L}_q$: Lattice of safe covers (Section 5.1)
$\mathcal{G}_q$: Space of generalized covers (Section 5.1)
$f||g$: Generalized query fragment (Definition 7)
q^g: Generalized cover-based reformulation (Section 5.2)

Table 1: Main notations introduced in this work.

expressive power, DL-Lite$_\mathcal{R}$ is a significant extension of the subset of RDF (comprising RDF Schema) which can be translated into description logics, a.k.a. the DL fragment of RDF; DL-Lite$_\mathcal{R}$ is also a fragment of Datalog$^\pm$ [10].

Given a role R, its *inverse*, denoted R^-, is the set: $\{(b, a) \mid R(a, b) \in \mathcal{A}\}$. We denote $N_R^\pm$ the set of roles made of all role names, together with their inverses: $N_R^\pm = N_R \cup \{r^- \mid r \in N_R\}$. For instance, `supervisedBy` and `supervisedBy`$^-$, whose meaning is *supervises*, are in $N_R^\pm$. A DL-Lite$_\mathcal{R}$ TBox constraint is either:

(i) a **concept inclusion** of the form $C_1 \sqsubseteq C_2$ or $C_1 \sqsubseteq \neg C_2$, where each of C_1, C_2 is either a concept from N_C, or $\exists R$ for some $R \in N_R^\pm$, and $\neg C_2$ is the complement of C_2. Here, $\exists R$ denotes the set of constants occurring in the first position in role R (i.e., the projection on the first attribute of R). For instance, $\exists$`supervisedBy` is the set of those supervised by somebody, while $\exists$`supervisedBy`$^-$ is the set of all supervisors (i.e., the projection on the first attribute of `supervisedBy`$^-$, hence on the second of `supervisedBy`);

(ii) a **role inclusion** of the form $R_1 \sqsubseteq R_2$ or $R_1 \sqsubseteq \neg R_2$, with $R_1, R_2 \in N_R^\pm$.

Observe that the left-hand side of the constraints are negation-free; in DL-Lite$_\mathcal{R}$, negation can only appear in the right-hand side of a constraint. Constraints featuring negation allow expressing a particular form of *integrity constraints*: *disjointness* or *exclusion* constraints. The next example illustrates DL-Lite$_\mathcal{R}$ KBs.

Example 1 (DL-Lite$_\mathcal{R}$ KB)**.** *Consider the DL-Lite$_\mathcal{R}$ TBox $\mathcal{T}$ in Table 2 expressing contraints on the* `Researcher` *and* `PhDStudent` *concepts, and the* `worksWith` *and* `supervisedBy` *roles. It states that PhD students are researchers* $(T1)$*, researchers work with researchers* $(T2)(T3)$*, working with someone is a symmetric relation* $(T4)$*, being supervised by someone implies working with her/him* $(T5)$*, only PhD students are supervised* $(T6)$ *and they cannot supervise someone* $(T7)$*.*

$$
\begin{array}{lrcl}
(T1) & \texttt{PhDStudent} & \sqsubseteq & \texttt{Researcher} \\
(T2) & \exists\texttt{worksWith} & \sqsubseteq & \texttt{Researcher} \\
(T3) & \exists\texttt{worksWith}^- & \sqsubseteq & \texttt{Researcher} \\
(T4) & \texttt{worksWith} & \sqsubseteq & \texttt{worksWith}^- \\
(T5) & \texttt{supervisedBy} & \sqsubseteq & \texttt{worksWith} \\
(T6) & \exists\texttt{supervisedBy} & \sqsubseteq & \texttt{PhDStudent} \\
(T7) & \texttt{PhDStudent} & \sqsubseteq & \neg\exists\texttt{supervisedBy}^-
\end{array}
$$

Table 2: Sample TBox $\mathcal{T}$.

DL constraint	**FOL constraint**	**Relational constraint (under Open World Assumption)**
$A \sqsubseteq A'$	$\forall x[A(x) \Rightarrow A'(x)]$	$A \subseteq A'$
$A \sqsubseteq \exists R$	$\forall x[A(x) \Rightarrow \exists y R(x,y)]$	$A \subseteq \Pi_1(R)$
$A \sqsubseteq \exists R^-$	$\forall x[A(x) \Rightarrow \exists y R(y,x)]$	$A \subseteq \Pi_2(R)$
$\exists R \sqsubseteq A$	$\forall x[\exists y R(x,y) \Rightarrow A(x)]$	$\Pi_1(R) \subseteq A$
$\exists R^- \sqsubseteq A$	$\forall x[\exists y R(y,x) \Rightarrow A(x)]$	$\Pi_2(R) \subseteq A$
$\exists R' \sqsubseteq \exists R$	$\forall x[\exists y R'(x,y) \Rightarrow \exists z R(x,z)]$	$\Pi_1(R') \subseteq \Pi_1(R)$
$\exists R' \sqsubseteq \exists R^-$	$\forall x[\exists y R'(x,y) \Rightarrow \exists z R(z,x)]$	$\Pi_1(R') \subseteq \Pi_2(R)$
$\exists R'^- \sqsubseteq \exists R$	$\forall x[\exists y R'(y,x) \Rightarrow \exists z R(x,z)]$	$\Pi_2(R') \subseteq \Pi_1(R)$
$\exists R'^- \sqsubseteq \exists R^-$	$\forall x[\exists y R'(y,x) \Rightarrow \exists z R(z,x)]$	$\Pi_2(R') \subseteq \Pi_2(R)$
$R \sqsubseteq R'^-$ or $R^- \sqsubseteq R'$	$\forall x,y[R(x,y) \Rightarrow R'(y,x)]$	$R \subseteq \Pi_{2,1}(R')$ or $\Pi_{2,1}(R) \subseteq R'$
$R \sqsubseteq R'$ or $R^- \sqsubseteq R'^-$	$\forall x,y[R(x,y) \Rightarrow R'(x,y)]$	$R \subseteq R'$ or $\Pi_{2,1}(R) \subseteq \Pi_{2,1}(R')$

Table 3: DL-Lite$_{\mathcal{R}}$ inclusion constraints *without* negation in FOL, and in relational notation; A, A' are concept names while R, R' are role names. For the relational notation, we use 1 to designate the first attribute of any atomic role, and 2 for the second.

Now consider the ABox $\mathcal{A}$ below, for the same concepts and roles:

$$
\begin{array}{ll}
(A1) & \texttt{worksWith}(Ioana, Francois) \\
(A2) & \texttt{supervisedBy}(Damian, Ioana) \\
(A3) & \texttt{supervisedBy}(Damian, Francois)
\end{array}
$$

It states that Ioana works with François $(A1)$*, Damian is supervised by both Ioana* $(A2)$ *and François* $(A3)$.

The semantics of inclusion constraints is defined, as customary, in terms of their FOL interpretations. Table 2.1 and 4 provide the FOL and relational notations expressing these constraints equivalently.

A KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ is *consistent* if the corresponding FOL theory, consisting of the $\mathcal{A}$ facts and of the FOL constraints corresponding to $\mathcal{T}$, has a model. In this case, we say also that $\mathcal{A}$ is $\mathcal{T}$*-consistent. In the absence of negation, any KB is consistent*,

DL constraint	**FOL constraint**	**Relational constraint (under Open World Assumption)**
$A \sqsubseteq \neg A'$	$\forall x[A(x) \Rightarrow \neg A'(x)]$	$A \cap A' \subseteq \bot$
$A \sqsubseteq \neg\exists R$	$\forall x[A(x) \Rightarrow \neg\exists y R(x,y)]$	$A \cap \Pi_1(R) \subseteq \bot$
$A \sqsubseteq \neg\exists R^-$	$\forall x[A(x) \Rightarrow \neg\exists y R(y,x)]$	$A \cap \Pi_2(R) \subseteq \bot$
$\exists R \sqsubseteq \neg A$	$\forall x[\exists y R(x,y) \Rightarrow \neg A(x)]$	$A \cap \Pi_1(R) \subseteq \bot$
$\exists R^- \sqsubseteq \neg A$	$\forall x[\exists y R(y,x) \Rightarrow \neg A(x)]$	$A \cap \Pi_2(R) \subseteq \bot$
$\exists R' \sqsubseteq \neg\exists R$	$\forall x[\exists y R'(x,y) \Rightarrow \neg\exists z R(x,z)]$	$\Pi_1(R') \cap \Pi_1(R) \subseteq \bot$
$\exists R' \sqsubseteq \neg\exists R^-$	$\forall x[\exists y R'(x,y) \Rightarrow \neg\exists z R(z,x)]$	$\Pi_1(R') \cap \Pi_2(R) \subseteq \bot$
$\exists R'^- \sqsubseteq \neg\exists R$	$\forall x[\exists y R'(y,x) \Rightarrow \neg\exists z R(x,z)]$	$\Pi_2(R') \cap \Pi_1(R) \subseteq \bot$
$\exists R'^- \sqsubseteq \neg\exists R^-$	$\forall x[R'(y,x) \Rightarrow \neg\exists z R(z,x)]$	$\Pi_2(R') \cap \Pi_2(R) \subseteq \bot$
$R \sqsubseteq \neg R'^-$ or $R^- \sqsubseteq \neg R'$	$\forall x,y[R(x,y) \Rightarrow \neg R'(y,x)]$	$R \cap \Pi_{2,1}(R') \subseteq \bot$ or $\Pi_{2,1}(R) \cap R' \subseteq \bot$
$R \sqsubseteq \neg R'$ or $R^- \sqsubseteq \neg R'^-$	$\forall x,y[R(x,y) \Rightarrow \neg R'(x,y)]$	$R \cap R' \subseteq \bot$ or $\Pi_{2,1}(R) \cap \Pi_{2,1}(R') \subseteq \bot$

Table 4: DL-Lite$_\mathcal{R}$ inclusion constraints *with* negation, i.e., disjointness constraints, in FOL and relational notation. $\bot$ denotes the empty relation.

as negation-free constraints merely lead to infering more facts. If some constraints feature negation, $\mathcal{K}$ is consistent iff none of its (explicit or inferred) facts contradicts a constraint with negation. An inclusion or assertion α is *entailed* by a KB $\mathcal{K}$, written $\mathcal{K} \models \alpha$, if α is satisfied in all the models of the FOL theory corresponding to $\mathcal{K}$.

Example 2 (DL-Lite$_\mathcal{R}$ entailment)**.** *The KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ from Example 1 entails many constraints and assertions. For instance:*

- $\mathcal{K} \models \exists$supervisedBy $\sqsubseteq \neg\exists$supervisedBy$^-$*, i.e., the two attributes of* supervisedBy *are disjoint, due to* $(T6) + (T7)$*;*
- $\mathcal{K} \models$ worksWith$(Francois, Ioana)$*, i.e., François works with Ioana, due to* $(T4) + (A1)$*;*
- $\mathcal{K} \models$ PhDStudent$(Damian)$*, i.e., Damian is a PhD student, due to* $(A2)+(T6)$*;*
- $\mathcal{K} \models$ worksWith$(Francois, Damian)$*, i.e., François works with Damian, due to* $(A3) + (T5) + (T4)$.

Finally remark that $\mathcal{A}$ is $\mathcal{T}$-consistent, i.e., there is no violation of its only constraint using negation $(T7)$*, since the KB $\mathcal{K}$ does not entail that some PhD student supervises another.*

2.2 Queries

A FOL query is of the form $q(\bar{x}) \leftarrow \phi(\bar{x})$ where $\phi(\bar{x})$ is a FOL formula whose free variables are $\bar{x}$; the query *name* is q, its *head* is $q(\bar{x})$, while its *body* is $\phi(\bar{x})$. Without

loss of generality, in the sequel, *we consider only connected queries*, i.e., those which do not feature cartesian products. The *answer set* of a query q against a knowledge base $\mathcal{K}$ is: $ans(q, \mathcal{K}) = \{\bar{t} \in (N_I)^n \mid \mathcal{K} \models \phi(\bar{t})\}$, where $\mathcal{K} \models \phi(\bar{t})$ means that every model of $\mathcal{K}$ is a model of $\phi(\bar{t})$. If q is Boolean, $ans(q, \mathcal{K}) = \{\langle\rangle\}$ encodes true, with $\langle\rangle$ the empty tuple, while $ans(q, \mathcal{K}) = \emptyset$ encodes false. In keeping with the literature on query answering under ontological constraints, *our queries have set semantics*, i.e., a tuple either belongs to the answer or does not, but it cannot appear several times in the answer.

Example 3 (Query answering)**.** *Consider the* FOL *query q asking for the PhD students with whom someone works:*

$$q(x) \leftarrow \exists y\ \mathtt{PhDStudent}(x) \wedge \mathtt{worksWith}(y, x)$$

Given the KB $\mathcal{K}$ of Example 1, the answer set of this query is $\{Damian\}$, since $\mathcal{K} \models \mathtt{PhDStudent}(Damian)$ and $\mathcal{K} \models \mathtt{worksWith}(Francois, Damian)$ hold. Observe that evaluating q against $\mathcal{K}$'s ABox only yields no answer.

To simplify the reading, in what follows, we omit the quantifiers of existential variables, and simply write the above query as $q(x) \leftarrow \mathtt{PhDStudent}(x) \wedge \mathtt{worksWith}(y, x)$.

Query dialects. We will need to refer to several FOL query dialects, whose general forms are schematized below:

CQ	$q(\bar{x}) \leftarrow a_1 \wedge \cdots \wedge a_n$
SCQ	$q(\bar{x}) \leftarrow (a_1^1 \vee \cdots \vee a_1^{k_1}) \wedge \cdots \wedge (a_n^1 \vee \cdots \vee a_n^{k_n})$
UCQ	$q(\bar{x}) \leftarrow \mathtt{CQ}_1(\bar{x}) \vee \cdots \vee \mathtt{CQ}_n(\bar{x})$
USCQ	$q(\bar{x}) \leftarrow \mathtt{SCQ}_1(\bar{x}) \vee \cdots \vee \mathtt{SCQ}_n(\bar{x})$
JUCQ	$q(\bar{x}) \leftarrow \mathtt{UCQ}_1(\bar{x_1}) \wedge \cdots \wedge \mathtt{UCQ}_n(\bar{x_n})$
JUSCQ	$q(\bar{x}) \leftarrow \mathtt{USCQ}_1(\bar{x_1}) \wedge \cdots \wedge \mathtt{USCQ}_n(\bar{x_n})$

Conjunctive Queries (CQs), a.k.a. select-project-join queries, are conjunctions of *atoms*, where an atom is either $A(t)$ or $R(t, t')$, for some t, t' variables or constants. *Semi-Conjunctive Queries* (SCQs) are joins of unions 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. *Unions of* SCQ*s* (USCQs), *Joins of* UCQ*s* (JUCQs), and finally *Joins of* USCQ*s* (JUSCQs) are built on top of simpler languages by adding unions, respectively joins. All the above dialects directly translate into SQL, and thus can be evaluated by an RDBMS.

Notations. Unless otherwise specified, we systematically use q to refer to a CQ query, $a_1, \ldots, a_n$ to designate the atoms in the body of q, $\mathcal{T}$ to designate a DL-Lite$_\mathcal{R}$ TBox, and $\mathcal{A}$ for an ABox.

FOL-reducibility of data management. In a setting where query answering is FOL-reducible, there exists a FOL query $q^{\mathtt{FOL}}$ (computable from q and $\mathcal{T}$) such that:

$$ans(q, \langle \mathcal{T}, \mathcal{A} \rangle) = ans(q^{\mathtt{FOL}}, \langle \emptyset, \mathcal{A} \rangle)$$

for any $\mathcal{T}$-consistent ABox $\mathcal{A}$. Thus, query answering reduces to: a first reasoning step to produce the FOL query from q and $\mathcal{T}$ (this is also known as *reformulating* the query using the constraints), and a second step which evaluates the *reformulated query* $q^{\texttt{FOL}}$ in the standard fashion, *only on the ABox* (i.e., disregarding the TBox constraints). This can be done for instance by translating it into SQL and delegating the evaluation to an RDBMS. From a knowledge base perspective, this allows to take advantage of highly optimized data stores and query evaluation engines to answer queries. From the database perspective, this two-step approach enhances the power of RDBMSs, as it allows to compute answers based only on data stored in the ABox (i.e., the database), but also taking into account the deductive constraints and all their consequences (entailed facts and constraints).

As DL-Lite$_\mathcal{R}$ query answering is FOL reducible [12], the literature provides techniques for computing FOL reformulations of a CQ in settings related to DL-Lite$_\mathcal{R}$. These techniques produce (i) a UCQ w.r.t. a DL-Lite$_\mathcal{R}$ TBox, e.g., [12, 1, 31, 14, 36], or extensions thereof using existential rules [21] or Datalog$^\pm$ [37, 19], (ii) a USCQ [35] 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 [18], whose RDF Schema constraints are the following four, out of the twenty-two, DL-Lite$_\mathcal{R}$ ones: (1) $A \sqsubseteq A'$, (4) $\exists R \sqsubseteq A$, (5) $\exists R^- \sqsubseteq A$ and (11) $R \sqsubseteq R'$.

CQ-to-UCQ reformulation for DL-Lite$_\mathcal{R}$ [12]. We present the pioneering CQ-to-UCQ technique on which we rely 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, *since they produce equivalent* FOL *queries*.

The technique of [12] relies on two operations: *specializing a query atom into another* by applying a negation-free constraint (recall Table 2.1) in the backward direction, and *specializing two atoms into their most general unifier* (mgu, in short). These operations are exhaustively applied to the input CQ; each operation generates a new CQ *contained in* the input CQ w.r.t. the TBox, because the new CQ was obtained by specializing one or two atoms of the previous CQ. The same process is then applied on the new CQs, and so on recursively until the set of generated CQs reaches a fixpoint. The finite union of the input CQ and of the generated ones forms the UCQ reformulation of the input CQ w.r.t. the TBox.

Example 4 (CQ-to-UCQ reformulation)**.** *Consider the query* $q(x) \leftarrow \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(y, x)$ *and KB* $\mathcal{K}$ *of the preceding examples. The* UCQ *reformulation of* q *is:* $q^{\texttt{UCQ}}(x) \leftarrow \bigvee_{i=1}^{10} q^i(x)$ *where* q^1*-*q^{10} *appear in Table 5. In the table,* $q^1(x)$ *has exactly the body of* q*.* $q^2(x)$ *is obtained from* q^1 *by applying the constraint* $(T4)$*:* $\texttt{worksWith} \sqsubseteq \texttt{worksWith}^-$*, which is of the form (10) listed in Table 2.1.* $(T4)$ *is* applied backward*, in the following sense: the query asks for* $\texttt{worksWith}(y, x)$*, and* $(T4)$ *tells us that* one of the possible reasons why this may hold*, is if* $\texttt{worksWith}(x, y)$ *holds. Thus,* q^2 *is contained within* q^1*, in the sense that if* q^2 *holds,* q^1 *is also sure to hold, but the opposite is not true; intuitively, "*q^1 *may hold for other reasons (thanks to other specializations of its atoms)" - and it is exactly the set of such other specializations which the technique explores.*

Similarly, q^3 *is obtained from* q^1 *by applying the constraint* $(T5)$ *backward on the*

$$
\begin{aligned}
q^1(x) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(y,x)\\
q^2(x) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x,y)\\
q^3(x) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(y,x)\\
q^4(x) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(x,y)\\
q^5(x) &\leftarrow \texttt{supervisedBy}(x,z) \wedge \texttt{worksWith}(y,x)\\
q^6(x) &\leftarrow \texttt{supervisedBy}(x,z) \wedge \texttt{worksWith}(x,y)\\
q^7(x) &\leftarrow \texttt{supervisedBy}(x,z) \wedge \texttt{supervisedBy}(y,x)\\
q^8(x) &\leftarrow \texttt{supervisedBy}(x,z) \wedge \texttt{supervisedBy}(x,y)\\
q^9(x) &\leftarrow \texttt{supervisedBy}(x,x)\\
q^{10}(x) &\leftarrow \texttt{supervisedBy}(x,y)
\end{aligned}
$$

Table 5: Union terms in CQ-to-UCQ reformulation (Example 4).

atom $\texttt{worksWith}(y,x)$*, and* q^4 *from* q^2 *by applying* $(T5)$ *on* $\texttt{worksWith}(x,y)$*. To obtain* q^5 *to* q^8*, we apply* $(T6)$ *backward on the atom* $\texttt{PhDStudent}(x)$ *in* q^1 *to* q^4*. Finally,* q^9 *is obtained from* q^7 *through the mgu of its two atoms, namely* $\texttt{supervisedBy}(x,z)$ *and* $\texttt{supervisedBy}(y,x)$*;* q^{10} *is similarly obtained from* q^8*.*

2.3 Evaluating reformulated subqueries can be (very) hard

It is worth noting that the (naïve) exhaustive application of specialization steps leads, in general, to *highly redundant reformulations* w.r.t. the containment of their disjuncts. For instance, minimizing $q^{\texttt{UCQ}}$ in the above example by eliminating disjuncts contained in another leads to: $q^{\texttt{UCQ}}_{min}(x) \leftarrow \bigvee_{i=1}^{3} q^i(x) \vee q^{10}(x)$ where the disjuncts appear in Table 5; they are all contained in q^{10}.

Minimal UCQ reformulations can be obviously processed more efficiently. However, they still repeat some computations, e.g., in the above example, PhDStudent is read three times, worksWith twice etc; in general, subqueries appearing in different union terms are repeatedly evaluated.

Common subexpression elimination (CSE) techniques aim at identifying repeated subexpressions in queries or plans, and reformulating them so that the expression is evaluated only once and its results are shared to increase performance; CSE is often used in a Multi-Query Optimization context (MQO). However, MQO is poorly supported in today's main RDBMS engines[3]. As we will see, our approach, which starts with the TBox and data statistics, and ends by handing over a chosen reformulation to the RDBMS, *never requires work to detect common (repeated) sub-expressions*. We further discuss and stress this aspect in Section 7.

Another source of difficulty is the *sheer size of reformulated queries*; we exhibit some whose size (i.e., length of the SQL formulation) is above 2.000.000 characters. For instance, the minimal UCQ corresponding to query Q_9 in our experiments (Section 6) is a union of 145 CQs, and runs in 5665 ms on DB2 and a database of 100 million facts. In contrast, the SQL translation of the best FOL reformulation identified by our

[3]We checked this on Postgres, DB2, and MySQL plans; according to Paul Larson (among the authors of [39]), no major RDBMS engine as of April 2016 has a comprehensive MQO approach.

approach reduces this to 156 ms (36 times faster), just by giving the engine a different (yet equivalent) SQLized FOL reformulation.

From an optimization viewpoint, the problem we are facing can be seen as follows. We aim at answering queries through RDBMSs in the presence of constraints, for FOL-reducible settings.

The standard UCQ reformulation (and other cost-ignorant ones) perform quite badly. The question is, then: is there an equivalent reformulation which would be evaluated more efficiently?

To answer this, one is faced with a set of FOL (or, alternatively, SQL) reformulations whose size is potentially very high: exponential in the query size for non-redundant queries, larger yet if one considers, for instance, queries featuring semijoins [7]; each query therein may be (very) large, have many unions etc. From these, one would need to find the one(s) best optimized and executed by the RDBMS. The size of the possible space, though, makes this utterly impractical.

The following sections present our alternative approach.

3 Optimization framework

The performance of evaluating (the SQL translation of) a given FOL reformulation of a query through an RDBMS depends on several factors: (i) data properties (size, cardinalities, value distributions etc); (ii) the storage model, i.e., the concrete relations storing the ABox, possible indexes etc; (iii) the optimizer's algorithm. Among these, (i) is completely determined by the dataset (the given ABox). On the storage model (ii), for generality, we make no assumption, other than requiring that FOL query reformulations can be translated into SQL on the underlying store. (We study several such concrete models experimentally, in Section 6). For what concerns optimizers (iii), we note that off-the-shelf they perform very poorly on previously proposed FOL query reformulations, yet we would like to exploit their strengths when possible.

Approach: cover-based query answering. We identify and exploit a novel space of *alternative* FOL *reformulations of the given input* CQ. We estimate the cost of evaluating each such reformulation through the RDBMS using standard database cost formulas, and hand to the RDBMS one with the best estimation.

More specifically, a query *cover* defines a way to split the query 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 of the original query (recall also Figure 1).

Definition 1 (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) $\bigcup_{i=1}^{m} f_i = \{a_1, \ldots, a_n\}$, (ii) no fragment is included into another, and (iii) the atoms of each fragment are connected through joins (common variables).*

Example 5 (CQ cover)**.** *Consider the query*

$$q(x, y) \leftarrow \begin{array}{l} \texttt{teachesTo}(v, x) \wedge \texttt{teachesTo}(v, y), \\ \texttt{supervisedBy}(x, w) \wedge \texttt{supervisedBy}(y, w) \end{array}$$

C, below, is a query cover for q:

$$C = \{\{\texttt{teachesTo}(v,x) \wedge \texttt{supervisedBy}(x,w)\}, \\ \{\texttt{teachesTo}(v,y) \wedge \texttt{supervisedBy}(y,w)\}\}$$

Definition 2 (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.*

Example 6 (Fragment queries of a CQ)**.** *The fragment queries of the query $q(x,y)$ w.r.t. the cover C (Example 5) are:*

$$q_{|f_1}(x,v,w) \leftarrow \texttt{teachesTo}(v,x) \wedge \texttt{supervisedBy}(x,w)$$

$$q_{|f_2}(y,v,w) \leftarrow \texttt{teachesTo}(v,y) \wedge \texttt{supervisedBy}(y,w)$$

As we shall see in the next Section, not every cover of a query leads to a FOL reformulation. Specifically, we define:

Definition 3 (Cover-based 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 reformulation of q w.r.t. $\mathcal{T}$ and C *<u>if</u> it is a* FOL *reformulation of q w.r.t. $\mathcal{T}$.*

To exemplify cover-based FOL reformulations, one needs to chose a specific KB dialect, among all those enjoying FOL reducibility; we present examples in the next Section, when instantiating our framework to the DL-Lite$_\mathcal{R}$ setting.

For now, it helps to see how we derive the SQL query corresponding to the cover-based reformulation. Each reformulated fragment query $q^{\texttt{FOL}}_{|f_i}$ is translated into an SQL query $\texttt{SQL}_\texttt{i}$; then, the overall query is of the form:

$\texttt{WITH SQL}_1 \texttt{ AS } \texttt{q}^{\texttt{FOL}}_{|\texttt{f}_1},\ \texttt{SQL}_2 \texttt{ AS } \texttt{q}^{\texttt{FOL}}_{|\texttt{f}_2}, \ldots, \texttt{SQL}_\texttt{n} \texttt{ AS } \texttt{q}^{\texttt{FOL}}_{|\texttt{f}_\texttt{n}}$
$\texttt{SELECT DISTINCT } \bar{\texttt{x}} \texttt{ FROM SQL}_1, \texttt{SQL}_2, \ldots, \texttt{SQL}_\texttt{n}$
$\texttt{WHERE cond}(1, 2, \ldots, \texttt{n})$

where $\texttt{cond}(1, 2, \ldots, \texttt{n})$ is the conjunction of the join predicates between all the subqueries. This leads to all the WITH-introduced subqueries being evaluated and materialized into intermediary tables[4], while the one with the largest number of results is run in pipeline fashion. The way in which each subquery is evaluated, then their results are joined, is left to the DBMS to determine. The SELECT DISTINCT ensures set semantics for the query answers.

We picked this syntax after experimenting with other variants, which in our experience lead to similar or worse performance. In particular, we tried:

[4]These SQL subqueries are of the form SELECTDISTINCT in order to reduce the size of the intermediate materialized results; this choice lead to the fastest execution in our experiments.

- defining each reformulated fragment query $\mathtt{SQL_i}$ as a (virtual) view, and joining these views in the global reformulation. This gives the query processor more freedom as it does no longer force the materialization of $\mathtt{SQL_i}$ but instead allows its evaluation to be blended with the evaluation of the joins across reformulated fragment queries. We noticed, however, that this did not overall improve performance.

- turning $\mathtt{SQL_i}$ subqueries into nested ones introduced with `EXISTS` as soon as the subquery did not contribute variables to the head of reformulated query. We tried this both on the `WITH` version and on the view-based versions; we did not notice significant improvements.

Problem statement. 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 (data access, joins, unions etc) applied on the ABox to compute the answers of a $q^{\mathtt{FOL}}$ reformulation. The cost estimation ϵ also accounts for the effort needed to join the reformulated fragment query answers, in the most efficient way.

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.*

A cost estimation function is provided by most RDBMSs storing the ABox for instance through the `SQL` `explain` directive. One can also estimate costs outside the engine using well-known textbook formulas, as in e.g., [9]. We use both options in our experiments.

4 Cover-based query answering in DL-Lite$_\mathcal{R}$

We now instantiate our cover-based query answering technique to the popular setting of DL-Lite$_\mathcal{R}$. As already mentioned in Section 2, we use the simple `CQ`-to-`UCQ` reformulation technique of [12] for establishing our results, and in our examples. However, our approach applies to *any* other `FOL` reformulation techniques for DL-Lite$_\mathcal{R}$, e.g., optimized `CQ`-to-`UCQ` or `CQ`-to-`USCQ` reformulation techniques, since these produce *equivalent* (though possibly syntactically different) `FOL` queries.

Example 7 (Running example)**.** *Let* $\mathcal{K}$ *be the KB with TBox* $\mathcal{T} = \{\mathtt{Graduate} \sqsubseteq \exists\mathtt{supervisedBy}, \mathtt{supervisedBy} \sqsubseteq \mathtt{worksWith}\}$ *and ABox*

$$\mathcal{A} = \{\mathtt{PhDStudent}(Damian), \mathtt{Graduate}(Damian)\}$$

Consider the query $q(x) \leftarrow \mathtt{PhDStudent}(x) \wedge \mathtt{worksWith}(x, y) \wedge \mathtt{supervisedBy}(z, y)$*, whose answer against* $\mathcal{K}$ *is* $\{Damian\}$.

The UCQ *reformulation of q is $q^{\texttt{UCQ}}(x) \leftarrow \bigvee_{i=1}^{4} q^i(x)$ with:*

$$\begin{aligned} q^1(x) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x,y) \\ &\quad \wedge \texttt{supervisedBy}(z,y) \\ q^2(x) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(x,y) \\ &\quad \wedge \texttt{supervisedBy}(z,y) \\ q^3(x) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(x,y) \\ q^4(x) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{Graduate}(x) \end{aligned}$$

Above, q^1 has the body of q; q^2 is obtained from q^1 by specializing the atom $\texttt{worksWith}(x,y)$ *through a backward application of* $\texttt{supervisedBy} \sqsubseteq \texttt{worksWith}$*. q^3 (highlighted in blue) results from q^2 by replacing* $\texttt{supervisedBy}(x,y)$ *and* $\texttt{supervisedBy}(z,y)$ *with their most general unifier*[5]*. Finally, q^4 is obtained from q^3, by specializing* $\texttt{supervisedBy}(x,y)$ *through the backward application of* $\texttt{Graduate} \sqsubseteq \exists\texttt{supervisedBy}$*; we also show q^4 in blue to highlight its connection with q^3.*

Now let $C_1 = \{\{\texttt{PhDStudent}(x), \texttt{worksWith}(x,y)\}, \{\texttt{supervisedBy}(z,y)\}\}$ be a cover of q. From Definition 2, the corresponding fragment queries are:

$$\begin{aligned} q_1(x,y) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x,y) \\ q_2(y) &\leftarrow \texttt{supervisedBy}(z,y) \end{aligned}$$

The reformulation of q_1 using $\mathcal{T}$ is $q_1^{\texttt{UCQ}}(x,y) \leftarrow \bigvee_{i=1}^{2} q_1^i(x,y)$, where

$$\begin{aligned} q_1^1(x,y) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x,y) \\ q_1^2(x,y) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(x,y) \end{aligned}$$

q_1^2 is obtained from q_1^1 by the backward application of the constraint $\texttt{supervisedBy} \sqsubseteq \texttt{worksWith}$*.*

The reformulation of q_2 using $\mathcal{T}$ is simply:

$$q_2^{\texttt{UCQ}}(y) \leftarrow \texttt{supervisedBy}(z,y)$$

By Definition 3, the reformulation of q using C_1 is the conjunction $q_{C_1}^{\texttt{JUCQ}}(x) \leftarrow q_1^{\texttt{UCQ}}(x,y) \wedge q_2^{\texttt{UCQ}}(y)$, which is clearly equivalent to the following UCQ *obtained by distributing $\wedge$ over $\vee$:*

$$q_{C_1}^{\texttt{UCQ}}(x) \leftarrow (q_1^1(x,y) \wedge q_2^{\texttt{UCQ}}(y)) \vee (q_1^2(x,y) \wedge q_2^{\texttt{UCQ}}(y))$$

where the first and second disjuncts correspond to the CQ*s:*

$$\begin{aligned} q_{C_1}^1(x) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x,y) \\ &\quad \wedge \texttt{supervisedBy}(z,y) \\ q_{C_1}^2(x) &\leftarrow \texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(x,y) \\ &\quad \wedge \texttt{supervisedBy}(z,y) \end{aligned}$$

[5]In this case, the *mgu* is $\texttt{supervisedBy}(x,y)$ because x is the head variable. Also, q^3 is equivalent to (and a minimal form of) q^2, but in general, q^3 is only guaranteed to be contained in (or equivalent to) q^2.

Above, $q^1_{C_1}(x)$ and $q^2_{C_1}(x)$ are exactly q^1 and q^2 from the UCQ *reformulation of q; however, q^3 and q^4 are missing from $q^{\texttt{JUCQ}}_{C_1}(x)$. Since q^4 derives from q^3, the absence of both can be traced to the absence of q^3. The reason C_1 does not lead to q_3 is that* $\texttt{supervisedBy}(x, y) \wedge \texttt{supervisedBy}(z, y)$ *is not obtained while reformulating $q_1(x, y)$, thus the unification of these two atoms (which could have lead to q^3) is missed. In the* CQ*-to-*UCQ *reformulation of q,* $\texttt{supervisedBy}(x, y) \wedge \texttt{supervisedBy}(z, y)$ *appears in q^2 because* $\texttt{worksWith}(x, y) \wedge \texttt{supervisedBy}(z, y)$ *appears in q^1. However,* C_1 separates the worksWith and supervisedBy atoms in different fragments. *Reformulating them independently misses exactly the opportunity to produce q^3 and q^4.*

Due to these absent subqueries, $q^{\texttt{JUCQ}}_{C_1}$ is not a FOL *reformulation of q w.r.t. $\mathcal{T}$, i.e., it fails to compute q's answer: $ans(q^{\texttt{JUCQ}}_{C_1}, \langle \emptyset, \mathcal{A} \rangle) = \emptyset$ while the answer of q against $\mathcal{K}$ is $\{Damian\}$.*

More generally, given an input CQ and a TBox, each pair of query atoms begetting unifications during the CQ-to-UCQ reformulation of the whole query must not be separated by (must not be in different fragments of) a cover, in order for the corresponding cover-based reformulation to be a FOL reformulation. When this is the case, we say the cover is *safe* for query answering.

Thus, we are interested in a sufficient condition for a cover to be safe; intuitively, we must approximate (by some supersets) those sets of atoms which (directly or after some specializations) are pairwise unified by the CQ-to-UCQ algorithm, and ensure that each such atom set is in the same cover fragment.

Only atoms with the same predicate may unify. Thus, we identify for each *predicate* (i.e., concept or role name) occurring in a query, *the set of all TBox predicates in which this predicate may turn* through some sequence of atom specializations, i.e., backward constraint application and/or unification (the two operations applied by the technique of [12] which we consider here). This is captured by the classical notion of dependencies between predicates within knowledge bases, Datalog programs, etc In DL-Lite$_{\mathcal{R}}$, this notion translates into the following recursive definition.

Definition 4 (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 denoted $dep(N)$ and defined as the fixpoint of:*

$$\begin{aligned} dep^0(N) &= \{N\} \\ dep^n(N) &= dep^{n-1}(N) \\ &\quad \cup \{cr(Y) \mid Y \sqsubseteq X \in \mathcal{T} \text{ and } cr(X) \in dep^{n-1}(N)\} \end{aligned}$$

where $cr(Y)$ returns, for any input Y of the form Z, Z^- or $\exists Z$ (for some concept or role Z), the concept or role name Z itself.

Example 8 (Predicate dependencies)**.** *In the TBox of Example 7:*

$$\begin{aligned} dep(\texttt{PhDStudent}) &= \{\texttt{PhDStudent}\} \\ dep(\texttt{Graduate}) &= \{\texttt{Graduate}\} \\ dep(\texttt{worksWith}) &= \{\texttt{worksWith}, \texttt{supervisedBy}, \texttt{Graduate}\} \\ dep(\texttt{supervisedBy}) &= \{\texttt{supervisedBy}, \texttt{Graduate}\} \end{aligned}$$

Above, worksWith *depends on* supervisedBy *because of the constraint* supervisedBy $\sqsubseteq$ worksWith*; similarly,* supervisedBy *depends on* Graduate *due to the constraint* Graduate $\sqsubseteq \exists$supervisedBy*, thus* worksWith *in turn depends on* Graduate*, too.*

Definition 5 (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 5 requires covers to be partitions, we will relax this restriction in Section 5.2.

Theorem 1 (Cover-based query answering)**.** *Let C be a safe cover for q w.r.t. $\mathcal{T}$. The cover-based reformulation (Definition 3) of q based on C, using any* CQ*-to-*UCQ *(resp.* CQ*-to-*USCQ*) reformulation technique, yields a cover-based reformulation $q^{\mathtt{FOL}}$ of q w.r.t. $\mathcal{T}$.*

Proof. The proof follows from that of correctness of the CQ-to-UCQ reformulation technique in [12] for query answering. It directly *extends* 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 the query produced by the technique described in [12].
Soundness: for any $\mathcal{T}$-consistent Abox $\mathcal{A}$, $ans(q^{\mathtt{FOL}}, \langle\emptyset, \mathcal{A}\rangle) \subseteq ans(q, \langle\mathcal{T}, \mathcal{A}\rangle)$ holds.

Let t be a tuple in $ans(q^{\mathtt{FOL}}, \langle\emptyset, \mathcal{A}\rangle)$. From Definition 3, $q^{\mathtt{FOL}}$ is $q^{\mathtt{FOL}}(\bar{x}) \leftarrow \bigwedge_{i=1}^{m} q^{\mathtt{FOL}}_{|f_i}$, thus t results from $t_i \in ans(q^{\mathtt{FOL}}_{|f_i}, \langle\emptyset, \mathcal{A}\rangle)$, for $1 \leq i \leq m$. Therefore, for $1 \leq i \leq m$, $t_i \in ans(q_{|f_i}, \langle\mathcal{T}, \mathcal{A}\rangle)$ holds, because of the soundness of the CQ-to-UCQ reformulation technique. Hence, from Definition 3, $t \in ans(q, \langle\mathcal{T}, \mathcal{A}\rangle)$ holds.
Completeness: for any $\mathcal{T}$-consistent Abox $\mathcal{A}$, $ans(q, \langle\mathcal{T}, \mathcal{A}\rangle) \subseteq ans(q^{\mathtt{FOL}}, \langle\emptyset, \mathcal{A}\rangle)$ holds.

Let t be a tuple in $ans(q, \langle\mathcal{T}, \mathcal{A}\rangle)$. Let $q^{\mathtt{UCQ}}$ be its reformulation using the CQ-to-UCQ technique. From the completeness of this technique, $t \in ans(q^{\mathtt{UCQ}}, \langle\emptyset, \mathcal{A}\rangle)$ holds. Let $q^{\mathtt{UCQ}}$ be $\bigvee_{l=1}^{\alpha} cq_l$, then necessarily for some l: $t \in ans(cq_l, \langle\emptyset, \mathcal{A}\rangle)$ holds [12].

Let $q^{\mathtt{FOL}}$ be $\bigwedge_{i=1}^{m} q^{\mathtt{FOL}}_{|f_i} = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{\beta_i} cq_{i,j}$. Since Definition 5 makes the reformulation of each fragment independent from another w.r.t. the CQ-to-UCQ technique, for any cq_l in $q^{\mathtt{UCQ}}$: $cq_l = \bigwedge_{i=1}^{m} cq_{i,k\in[1,\beta_i]}$ holds. Hence, $t \in ans(q^{\mathtt{FOL}}, \langle\emptyset, \mathcal{A}\rangle)$ holds. □

If a CQ-to-UCQ reformulation algorithm is used on fragment queries, the cover-based reformulation will be a JUCQ; otherwise, a CQ-to-USCQ reformulation of the fragment queries lead to a JUSCQ reformulation.

Note that the trivial one-fragment cover (comprising all query atoms) is always safe; in this case, our query answering technique reduces to just one reformulation, the CQ-to-UCQ one identified by previous reformulation algorithms from the literature.

Example 9 (JUCQ reformulation with a safe cover)**.** *We now consider the* safe *cover* $C_2 = \{\{\mathtt{PhDStudent}(x)\}, \{\mathtt{worksWith}(x, y), \mathtt{supervisedBy}(z, y)\}\}$. *The cover-based reformulation based on C_2 is the* JUCQ *query* $q^{\mathtt{JUCQ}}(x) \leftarrow q_1^{\mathtt{UCQ}}(x) \wedge q_2^{\mathtt{UCQ}}(x)$*, where:*

$$
\begin{aligned}
q_1^{\texttt{UCQ}}(x) &\leftarrow \texttt{PhDStudent}(x)\\
q_2^{\texttt{UCQ}}(x) &\leftarrow (\texttt{worksWith}(x,y) \wedge \texttt{supervisedBy}(z,y))\\
&\quad \vee (\texttt{supervisedBy}(x,y) \wedge \texttt{supervisedBy}(z,y))\\
&\quad \vee \texttt{supervisedBy}(x,y) \vee \texttt{Graduate}(x)
\end{aligned}
$$

Observe that $ans(q^{\texttt{JUCQ}}, \langle \emptyset, \mathcal{A}\rangle) = \{Damian\} = ans(q, \langle \mathcal{T}, \mathcal{A}\rangle)$.

5 Cover-based query optimization in DL-Lite$_\mathcal{R}$

We study now the query answering optimization problem of Section 3 for DL-Lite$_\mathcal{R}$. We analyze a first optimization space in Section 5.1, before extending our discussion to a larger space in Section 5.2. Finally, we describe our search algorithms in Section 5.3.

5.1 Safe covers optimization space

Below, we study the space of safe covers for a given query and TBox. We start by identifying a particularly interesting one:

Definition 6 (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 predicate on which those of a_1 and a_2 depend w.r.t. $\mathcal{T}$, 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} does not depend on the order in which the fragments are considered (due to the inflationary method building it). Further, C_{root} is safe, given that it keeps in a single fragment any two atoms whose predicates may be unified.

The following important lemma characterizes the structure of C_{root} fragments:

Lemma 1 (C_{root} fragment structure)**.** *A fragment f in the root cover C_{root} is of one of the following two forms:*

1. *a singleton, i.e., $f = \{a_i\}$ for some query atom a_i;*
2. *$f = \{a_{i_1}, \ldots, a_{i_n}\}$, for $n \geq 2$, and for every atom $a_{i_1} \in f$, there exists one atom $a_{i_2} \in f$, and a predicate b_j in the TBox, such that both the predicates of a_{i_1} and of a_{i_2} depend on b_j.*

Proof. The lemma follows directly from the definition of C_{root}. Those atoms that do not share a dependency with any other atom appear in singleton fragments (case 1 above, as the construction of the root cover never groups them together). Atoms which share some dependencies (i.e., atoms whose predicates depend on one another) get unioned in fragments of the form 2 above. □

Example 10 (Root cover)**.** *On the query and TBox from Example 7, the root cover is C_2 from Example 9;* $\texttt{worksWith}(x,y)$ *and* $\texttt{supervisedBy}(z,y)$ *are in the same C_2 fragment because* `worksWith` *depends on* `supervisedBy` *(cf. Example 8).*

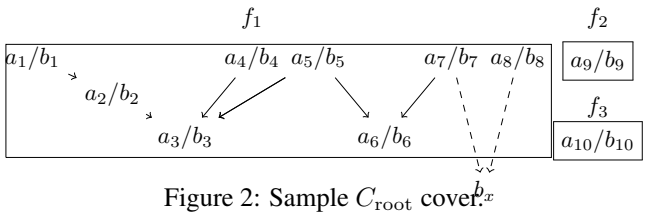


Figure 2: Sample C_{root} cover.

Example 11 (Complex root cover)**.** *Figure 2 depicts a possible C_{root} cover of a 10-atoms query; the cover has 3 fragments, each shown in a rectangle. Every a_i/b_i denotes a query atom a_i whose predicate is b_i; a plain arrow from a node to another denotes that the predicate of the first depends on the predicate of the second. The predicate b_x appears in the TBox but does not appear in the query. In this example, b_1, b_2, b_4 and b_5 depend on b_3; b_5 and b_7 depend on b_6; b_7 and b_8 on b_x etc. Fragment f_1 corresponds to case 2 of Lemma 1, while fragments f_2 and f_3 correspond to its first case.*

Proposition 1 states that C_{root} has the maximal number of fragments (equivalently, it has the smallest fragments) among all the safe covers for q and $\mathcal{T}$; its proof is based on Lemma 1.

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.*

Proof. For ease of explanation, in the proof, we rely on the graphical directed graph representation used in Figure 2 for dependencies between the predicates appearing in the atoms of a cover and/or other predicates from the KB.

Because f holds at least a_i and a_j, it must be a fragment of form 2, as stated in Lemma 1. It follows, thus, that in f there exists what we call an *extended path* e, going from a_i to a_j following the dependency edges either from source to target, or in the opposite direction; in other words, e alternately moves "up (or down) then down (or up)" a certain number of times in the fragment.

If e only contains edges in the same direction (either all are $\rightarrow$ or all are $\leftarrow$), it follows immediately that a_i and a_j are in the same fragment of C.

In the contrary case, there must exist some predicates in the TBox $b_1, \ldots, b_m$, $m \geq 1$, and some f atoms $a_1, \ldots, a_{m-1}$ defining an extended path e from a_i to a_j in f, as follows:

1. $a_i \rightarrow \ldots \rightarrow b_1$ (a $\rightarrow$ path segment), or b_1 is the predicate used in a_i;

2. b_k ($1 \leq k < m-1$), is the predicate used in a_l ($k \leq l < m-1$), or $b_k \leftarrow \ldots \leftarrow a_l$ (a $\leftarrow$ path segment);

3. $b_k \leftarrow \ldots \leftarrow a_l$ (a $\leftarrow$ path segment), with $1 \leq k < m-1$ and $k \leq l < m-1$, and $a_l \rightarrow \ldots \rightarrow b_{k+1}$. :

4. b_{m-1} is the predicate used in a_{m-1} or $b_{m-1} \leftarrow \ldots \leftarrow a_{m-1}$, then $a_{m-1} \rightarrow \ldots \rightarrow b_m$;

5. b_m is the predicate used in a_j or $b_m \leftarrow \ldots \leftarrow a_j$

Observe that items (2) and (3) can repeat (alternately) until b_{m-1} is reached.

Since C is safe, a_i and a_1 must appear in the same fragment in C (and only in that fragment), because they both depend on b_1.

For $1 \leq i \leq m-2$, a_i must appear in the same fragment as a_{i+1} (and only there), given that they both depend on b_i.

Since C is safe, a_j and a_{m-1} must appear in the same fragment of C (and only there).

From the above follows that $\{a_i, a_1, \ldots, a_{m-1}, a_j\}$ are all in the same fragment of C, which contradicts our hypothesis. □

From Proposition 1, we obtain:

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

Proof. Suppose that f is not a union of some fragments from C_{root}, and let us show a contradiction. In this case, f necessarily contains a strict, non-empty subset of a fragment of C_{root}. It follows that there are two atoms whose predicates depend on a common concept or role name w.r.t. $\mathcal{T}$ (as they were together in the fragment of C_{root}) that are not in a same fragment of C. Therefore C is not a safe cover, a contradiction. □

Safe cover lattice. Theorem 2 entails that the safe covers of a query q form a *lattice*, denoted $\mathcal{L}_q$, whose precedence relationship is denoted $\prec$, where $C_1 \prec C_2$ iff each fragment of C_2 is a union of some fragments of C_1. The lattice has as lower bound the single-fragment cover, and as upper bound the *root* cover. For convenience, we also use $\mathcal{L}_q$ to denote the set of all safe covers.

The size of the safe cover lattice is bounded by the number of partitions of the fragments in C_{root}, i.e., by the number of partitions of the query atoms[6], a.k.a. the Bell number B_n for a query of n atoms; the bound occurs when there is no dependency between the atom predicates.

5.2 Generalized covers optimization space

A dependency-rich TBox leads to few, large fragments in C_{root}, thus to a relatively small number of alternative cover-based reformulations. In this section, we explore a notion of *generalized covers*, and propose a method for deriving FOL query reformulations from such covers. This enlarges our space of alternatives and thus potentially leads to a better cost-based choice of reformulation.

We call *generalized fragment* of a query q and denote $f \| g$ a pair of q atom sets such that $g \subseteq f$. A *generalized cover* is a set of generalized fragments $C = \{f_1 \| g_1, \ldots,$

[6]See https://oeis.org/A000110.

$f_m \| g_m\}$ of a query q such that $\cup_{1 \leq i \leq m} f_i$ is the set of atoms of the query, and no f_i is included in f_j for $1 \leq i \neq j \leq m$.

To a generalized fragment $f \| g$ of a generalized cover C, we associate:

Definition 7 (Generalized fragment query of a CQ)**.** *The* generalized 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 other generalized fragment $f' \| g'$ of C.*

In a generalized fragment query, atoms from $f \setminus g$ only *reduce (filter)* the answers, without adding variables to the head. In particular, if $f = g$, $q_{|f\|g}$ coincides with the regular fragment query (Definition 2).

Given a generalized cover, the *generalized cover-based reformulation of a query q* is the FOL query

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

<u>if</u> q^g is a FOL reformulation.

If $f_i = g_i$ for all the fragments $f_i \| g_i$, the generalized cover-based reformulation coincides with the regular cover-based one (Definition 3). As for simple cover-based reformulations, if fragments are reformulated into UCQs, the reformulated query is a JUCQ, whereas if they are reformulated into USCQs, the reformulated query is a JUSCQ.

The introduction of extra atoms in generalized fragments is reminiscent of the classical semijoin reducers [7], whereas one computes $R(x, y) \bowtie_y S(y, z)$ by

$$(R(x, y) \ltimes_y \pi_y(S(y, z))) \bowtie_y S(y, z)$$

where $\ltimes_y$ denotes the left semijoin, returning every tuple from the left-hand side input that joins with the right-hand input. 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 the $\ltimes_y$ operator can be evaluated very efficiently. Further, if only few R tuples survive the $\ltimes_y$, the cost of the $\bowtie_y$ operator likely decreases with the size of its input.

While the benefits of semijoins are well-known, there are many ways to introduce them in a given query, increasing the space of alternative plans to be considered by an optimizer. While some heuristics have been proposed to explore only some carefully chosen semijoin plans [34], we noted that RDBMS optimizers do not explore semijoin options, in particular for the very large queries resulting from the FOL reformulations of CQs. *Generalized fragments mitigate this problem by intelligently using semijoin reducers to fasten the evaluation of the* FOL *reformulation by the RDBMS.*

Generalized search space. We now define the space $\mathcal{G}_q$ of generalized covers for a given query q, based on the safe cover set $\mathcal{L}_q$. A generalized cover $C = \{f_1 \| g_1, \ldots, f_m \| g_m\}$ is part of $\mathcal{G}_q$ iff:

- The cover $C_s = \{g_1, \ldots, g_m\}$ is safe, i.e., $C_s \in \mathcal{L}_q$;

- For each $1 \leq i \leq m$, the atoms in f_i form a connected graph.

Note that an atom $a \in f$, for $f \| g \in C$, *has no impact on the head of the corresponding generalized fragment query*; only the body of this query changes.

The size of $\mathcal{G}_q$ obviously admits that of $\mathcal{L}_q$ as a lower bound. For a query q of n atoms, a upper bound is $B_n * n * 2^{n-1}$, where B_n is the n-th Bell number: for each safe cover C (of which there are at most B_n, see the previous section), each of the n atoms may, in the worst case, be added or not to all the fragments to which it does not belong. In the worst case, there are $n-1$ such fragments.

The core result allowing us to benefit of the performance savings of generalized covers in order to efficiently answer queries is:

Theorem 3 ($\mathcal{G}_q$ cover-based query answering)**.** *The reformulation of a query q based on $\mathcal{T}$ and a generalized cover $C \in \mathcal{G}_q$ is a* FOL *reformulation of q w.r.t. $\mathcal{T}$.*

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

Example 12 (Generalized cover-based reformulation)**.** *Recall the query and KB from Example 7. Let $f_0 = \{\texttt{PhDStudent}(x)\}$ and $f_1 = \{\texttt{worksWith}(x, y), \texttt{supervisedBy}(z, y)\}$ be the two fragments of the root cover C_{root}. Consider also the generalized cover $C_3 = \{f_1 \| f_1, f_2 \| f_0\}$, where $f_2 = \{\texttt{PhDStudent}(x), \texttt{worksWith}(x, y)\}$.*

The generalized fragment query $q_{|f_1 \| f_1}$ of q w.r.t. C_3 is the subquery $q_{|f_1 \| f_1}(x) \leftarrow \texttt{worksWith}(x, y) \wedge \texttt{supervisedBy}(z, y)$. Observe that y is not a free variable of $q_{|f_1 \| f_1}$, as it is neither a free variable of q nor a variable in f_0, whereas $f_2 \| f_0$ is the only other fragment in the cover C_3.

The generalized fragment query $q_{|f_2 \| f_0}$ of q w.r.t. C_3 is the subquery $q_{|f_2 \| f_0}(x) \leftarrow \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x, y)$. Again, note that y is not a (free) variable of f_0, and therefore it is not a free variable of $q_{|f_2 \| f_0}$.

Then, the generalized cover-based reformulation corresponding to C_3 is the FOL *query:*

$$q^g(x) \leftarrow q^{\texttt{FOL}}_{|f_1 \| f_1}(x) \wedge q^{\texttt{FOL}}_{|f_2 \| f_0}(x)$$

where:

$$\begin{aligned} q^{\texttt{FOL}}_{|f_1 \| f_1}(x) \leftarrow \quad & (\texttt{worksWith}(x, y) \wedge \texttt{supervisedBy}(z, y)) \\ & \vee \texttt{supervisedBy}(x, y) \vee \texttt{Graduate}(x) \end{aligned}$$

$$\begin{aligned} q^{\texttt{FOL}}_{|f_2 \| f_0}(x) \leftarrow \quad & (\texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x, y)) \\ & \vee (\texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(x, y)) \\ & \vee (\texttt{PhDStudent}(x) \wedge \texttt{Graduate}(x)) \end{aligned}$$

Applying $\texttt{supervisedBy} \sqsubseteq \texttt{worksWith}$ *to* $q_{|f_1\|f_1}$ *leads to:*

$$\begin{array}{rl} & (\texttt{worksWith}(x,y) \wedge \texttt{supervisedBy}(z,y)) \\ & \vee (\texttt{supervisedBy}(x,y) \wedge \texttt{supervisedBy}(x,y)) \\ \equiv & (\texttt{worksWith}(x,y) \wedge \texttt{supervisedBy}(z,y)) \\ & \vee \texttt{supervisedBy}(x,y) \end{array}$$

Then, applying $\texttt{Graduate} \sqsubseteq \exists\texttt{supervisedBy}$*, we obtain the reformulation of* $q_{|f_1\|f_1}$ *w.r.t. TBox* $\mathcal{T}$*, i.e.,* $q^{\texttt{FOL}}_{|f_1\|f_1}$*. Similarly, applying to* $q_{|f_2\|f_0}$ *the constraint* $\texttt{supervisedBy} \sqsubseteq \texttt{worksWith}$ *and subsequently* $\texttt{Graduate} \sqsubseteq \exists\texttt{supervisedBy}$ *leads to* $q^{\texttt{FOL}}_{|f_2\|f_0}$.

Note that $ans(q^g, \langle \emptyset, \mathcal{A} \rangle) = \{Damian\} = ans(q, \langle \mathcal{T}, \mathcal{A} \rangle)$.

5.3 Cost-based cover search algorithms

Our first algorithm, **EDL (Exhaustive Covers for DL)**, starts from C_{root} and builds all $\mathcal{L}_q$ covers by unioning fragments, and all $\mathcal{G}_q$ covers by adding atoms (Algorithm 1).

Algorithm 1: Exhaustive Cover Search for DL-Lite$_\mathcal{R}$ (**EDL**)

Input : CQ $q(\bar{x}) \leftarrow a_1 \wedge \cdots \wedge a_n$, KB $\mathcal{K}$
Output: Best cover for reformulating for q

1 $\mathcal{L}_q \leftarrow \emptyset; \mathcal{G}_q \leftarrow \emptyset$;
2 $F \leftarrow C_{\text{root}}; C_{best} \leftarrow C_{\text{root}}$;
3 **foreach** $P = \{s_1, \ldots, s_{|P|}\}$ *distinct partition of* F *s.t. the atoms of all the fragments in each set* s_i*, for* $1 \leq i \leq |P|$*, are connected* **do**
4 $\quad C_P \leftarrow \emptyset$;
5 $\quad$ **foreach** *fragment set* $s_i = \{f_i^1, \ldots, f_i^{n_i}\} \in P$ **do**
6 $\quad\quad C_P \leftarrow C_P \cup \{f_i^1 \cup \cdots \cup f_i^{n_i}\}$;
7 $\quad \mathcal{L}_q \leftarrow \mathcal{L}_q \cup \{C_P\}$;
8 $\quad$ **if** C_P *estimated cost* $< C_{best}$ *estimated cost* **then**
9 $\quad\quad C_{best} \leftarrow C_P$;
10 **foreach** *safe cover* $C = \{f_1, f_2, \ldots, f_n\} \in \mathcal{L}_q$ **do**
11 $\quad C' \leftarrow \{f_1||f_1, f_2||f_2, \ldots, f_n||f_n\}$;
12 $\quad$ **foreach** $f||g \in C', a_i \notin f$ *such that* a_i *shares a variable with an* f *atom* **do**
13 $\quad\quad C'' \leftarrow C' \setminus \{f||g\} \cup \{f \cup \{a_i\}||g\}$;
14 $\quad\quad \mathcal{G}_q \leftarrow \mathcal{G}_q \cup \{C''\}$;
15 $\quad\quad$ **if** C'' *estimated cost* $< C_{best}$ *estimated cost* **then**
16 $\quad\quad\quad C_{best} \leftarrow C''$;
17 **return** C_{best};

The second one, **GDL (Greedy Covers for DL)** (Algorithm 2) works in a greedy fashion. It is based on exploring, from a given cover C, the set of possible next moves (lines 2-4 and 5-7); these are all the covers that may be created *from* C by unioning

Query q	$\lvert q^{\texttt{UCQ}}\rvert$
$q_1(u,i,n,e,t) \leftarrow$ ub:Professor(x) $\wedge$ ub:degreeFrom(x,u) $\wedge$ ub:researchInterest$(x,i)\wedge$ ub:name(x,n) $\wedge$ ub:emailAddress(x,e) $\wedge$ ub:telephone(x,t)	145
$q_2(x,e,t) \leftarrow$ ub:Professor$(x)\wedge$ ub:degreeFrom$(x,$"http://www.University870.edu"$)\wedge$ ub:researchInterest$(x,$"Research21"$)$ $\wedge$ ub:name$(x,$"AssociateProfessor2"$)\wedge$ ub:emailAddress$(x,e)\wedge$ ub:telephone(x,t)	145
$q_3(x) \leftarrow$ ub:Professor$(x)\wedge$ ub:degreeFrom$(x,$"http://www.University870.edu"$)\wedge$ ub:researchInterest$(x,$"Research21"$)$ $\wedge$ ub:name$(x,$"AssociateProfessor2"$)\wedge$ ub:emailAddress$(x,$"AssocProf2@Dept1.Univ0.edu"$)\wedge$ ub:telephone$(x,$"xxx-xxx-xxxx"$)\wedge$	145
$q_4(x,y) \leftarrow$ ub:Professor$(x)\wedge$ ub:teacherOf$(x,y)\wedge$ ub:degreeFrom$(x,$"http://www.University870.edu"$)$ $\wedge$ ub:researchInterest$(x,$"Research21"$)\wedge$ ub:name$(x,$"AssociateProfessor2"$)$ $\wedge$ ub:telephone$(x,$"xxx-xxx-xxxx"$)\wedge$ ub:emailAddress$(x,$"AssocProf2@Dept1.Univ0.edu"$)$	145
$q_5(x,y,z) \leftarrow$ ub:Professor$(x)\wedge$ ub:teacherOf$(x,y)\wedge$ ub:worksFor$(x,z)\wedge$ ub:degreeFrom$(x,$"http://www.University870.edu"$)\wedge$ ub:researchInterest$(x,$"Research21"$)$ $\wedge$ ub:name$(x,$"AssociateProfessor2"$)\wedge$ ub:emailAddress$(x,$"AssocProf2@Dept1.Univ0.edu"$)\wedge$ ub:telephone$(x,$"xxx-xxx-xxxx"$)$	290
$q_6(x,n) \leftarrow$ ub:Faculty$(x)\wedge$ ub:publicationAuthor$(y,x)\wedge$ ub:researchInterest$(x,$"Research16"$)\wedge$ ub:name$(y,n)\wedge$ ub:emailAddress$(x,$"AssocProf0@Dept0.Univ0.edu"$)$	35
$q_7(n) \leftarrow$ ub:Professor$(x)\wedge$ ub:teacherOf$(x,c)\wedge$ ub:memberOf$(x,$"http://www.Dep0.Univ0.edu"$)$ $\wedge$ ub:name$(x,n)\wedge$ ub:emailAddress$(x,$"FullProf8@Dept0.Univ0.edu"$)\wedge$ ub:telephone$(x,$"xxx-xxx-xxxx"$)$	116

Figure 3: Queries used in our experiments (part 1). We have shortened for presentation purposes some of the strings, e.g., AssociateProfessor2@Department1.University0.edu becomes AssocProf2@Dept1.Univ0.edu.

two of its fragments or by enlarging one of its fragments, i.e., turning a fragment $f\|g$ into $f \cup \{a\}\|g$ for some query atom a sharing a variable with f. The best one seen

Query q	$\lvert q^{\mathtt{UCQ}}\rvert$
$q_8(x,n) \leftarrow$ ub:Faculty$(x) \wedge$ ub:publicationAuthor$(y,x) \wedge$ ub:researchInterest$(x,$"Research16"$) \wedge$ ub:name(y,n)	35
$q_9(n,e) \leftarrow$ ub:Student$(x) \wedge$ ub:takesCourse$(x,c) \wedge$ ub:advisor$(x,a) \wedge$ ub:memberOf$(x,$"http://www.Dept0.University0.edu"$) \wedge$ ub:telephone$(x,$"xxx-xxx-xxxx"$) \wedge$ ub:teacherOf$(p,c) \wedge$ ub:emailAddress$(p,e) \wedge$ ub:researchInterest$(a,$"Research7"$) \wedge$ ub:memberOf$(a,$"http://www.Dept0.University0.edu"$) \wedge$ ub:name(c,n)	368
$q_{10}(n,e) \leftarrow$ ub:Professor$(x) \wedge$ ub:takesCourse$(x,c) \wedge$ ub:advisor$(x,a) \wedge$ ub:memberOf$(x,$"http://www.Dept0.University0.edu"$) \wedge$ ub:telephone$(x,$"xxx-xxx-xxxx"$) \wedge$ ub:teacherOf$(p,c) \wedge$ ub:emailAddress$(p,e) \wedge$ ub:researchInterest$(a,$"Research7"$) \wedge$ ub:memberOf$(a,$"http://www.Dept0.University0.edu"$) \wedge$ ub:name(c,n)	464
$q_{11}(x) \leftarrow$ ub:Professor$(x) \wedge$ ub:Student(x)	667
$q_{12}(x) \leftarrow$ ub:Professor$(x) \wedge$ ub:Department(x)	609
$q_{13}(x) \leftarrow$ ub:Publication$(x) \wedge$ ub:Department(x)	357

Figure 4: Queries used in our experiments (part 2).

at a given point (w.r.t. the estimated evaluation cost) is kept as the selected next move in the $move$ variable. At the end of this exploration step (line 9), the best move is applied, leading to the new best cover C from which the next exploration step starts. The exploration stops when no possible next move improves the cost of the currently selected best cover C.

When unioning 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 $q^{\mathtt{FOL}}$ 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 5.2). In our context, many such opportunities exist, as our experiments show.

6 Experimental evaluation

We implemented our cover-based query answering approach in Java 8; the source code has about 10.000 lines, including the statistics and cost estimation (see below).

Algorithm 2: Greedy Cover Search for DL-Lite$_{\mathcal{R}}$ (**GDL**)

Input : CQ $q(\bar{x}) \leftarrow a_1 \wedge \cdots \wedge a_n$, KB $\mathcal{K}$
Output: Best cover for reformulating q

1 $C \leftarrow C_{\text{root}}$; $move \leftarrow \emptyset$;
2 **foreach** $f_1, f_2 \in C$ **do**
3 **if** *($move$ is empty and $C.union(f_1, f_2)$ est. cost $\leq C$ est. cost) or ($C.union(f_1, f_2)$ est. cost $< apply(move)$ est. cost)* **then**
4 $move \leftarrow (C, f_1, f_2)$;
5 **foreach** $f \in C, a \in q$ *s.t. a is connected to f* **do**
6 **if** *($move$ is empty and $C.enlarge(f, a)$ est. cost $\leq C$ est. cost) or ($C.enlarge(f, a)$ est. cost $< apply(move)$ est. cost)* **then**
7 $move \leftarrow (C, f, \{a\})$;
8 **while** $move \neq \emptyset$ **do**
9 $C \leftarrow apply(move)$; // the cover obtained from that move
10 $move \leftarrow \emptyset$;
11 // Gather move starting from C as was done at lines 2–7 above
12 **return** C;

6.1 Experimental settings

RDBMSs and data layout. First, we used **PostgreSQL v9.3.2** to store the data and evaluate FOL query reformulations. Our *first data layout* within Postgres stored all the assertions into a single triple table (where each $C(x) \in \mathcal{A}$ leads to a triple x *type* C and each $R(a, b) \in \mathcal{A}$ leads to a triple $a\ R\ b$), and built all six three-attribute indexes on this triple table [29]. Our *second data layout* stored a unary table for each concept and a binary table for each role, and built all one- and two-attribute indexes, respectively, on those tables. Our tests showed that the second layout significantly outperformed the first; this is not surprising, as smaller tables lead to better performance, at the same time it reduces the number of query conditions (as some of them are encoded by accessing a certain table). Thus, *for Postgres, we only report results based on the layout featuring role and concept tables*.

Second, we used the **IBM DB2 Express-C 10.5**. We chose it because (i) we found out in prior work [9] (and confirm below) that it evaluates large FOL reformulations better than Postgres, and (ii) it provides a relatively recent, smart storage layout for RDF graphs [8], intelligently bundling assertions into a small set of tables with potentially many attributes, so that the roles to which an individual participates are stored, to the extent possible, in the same tuple. This reduces the number of joins needed for query evaluation, and has been shown [8] to improve query performance. However, DB2 does not support reasoning, i.e., it only provides query evaluation. For DB2, *we report results based on the concept and role tables (denoted simple layout) and on the RDF layout* of [8].

In the simple layout, as customary in efficient Semantic Web data management

systems, e.g., [29], facts are *dictionary-encoded* into integers, prior to storing them in the RDBMS. The TBox and predicates dependencies are stored in memory.

Hardware. The database servers ran on an 8-core Intel Xeon E5506 2.13 GHz machine with 16GB RAM, using Mandriva Linux r2010.0.

Datasets, queries, and reformulation engine. We used two LUBM$^\exists$ benchmark KBs, comprising a DL-Lite$_\mathcal{R}$ TBox and two ABoxes of 15 million, respectively, 100 million facts, obtained using the EUDG data generator [24]. The TBox consists of 34 roles, 128 concepts and 212 constraints.

We devised a set of 13 CQs against this knowledge base, shown in Figure 3 and 4. The queries have between 2 and 10 atoms, with an average of 5.77 atoms. Their UCQ reformulations are unions of 35 to 667 CQs, 290.2 on average. This parameter characterizing the query can be seen as a (rough) measure of the complexity of its reformulation; it is shown in Table 3 in the column $|q^{\texttt{UCQ}}|$.

We relied on the RAPID [14] CQ-to-UCQ reformulation tool to reformulate (simple or generalized) fragment queries (Definitions 2 and 7); any other CQ-to-UCQ or CQ-to-USCQ reformulation technique could have been used instead.

Cost estimation function. For the cost function estimation ϵ, we first used the RDBMS cost estimation for the SQL translation of each candidate FOL reformulation produced by our algorithms. For Postgres, we obtained this using `explain`[7], while for DB2 we used `db2expln`[8].

Further, for the simple layout, we implemented our own Java-based cost estimation, based on statistics on the stored data (cardinality and number of distinct values in each stored table attribute), and on the uniform distribution and independent distributions assumptions. Better RDF cardinality estimation techniques such as [28] may be used to improve the accuracy of our cost model.

For the sake of completeness, the next Section details how we compute the cost of evaluating a JUCQ (reformulation) sent to an RDBMS; this presentation is borrowed and adapted from [9].

6.2 Our cost estimation function

A JUCQ is a join of UCQs subqueries of the form: $q^{\texttt{JUCQ}}(\bar{x}) \leftarrow q_1^{\texttt{UCQ}} \bowtie \cdots \bowtie q_m^{\texttt{UCQ}}$.

The evaluation cost of $q^{\texttt{JUCQ}}$ is

$$\begin{aligned} c(q^{\texttt{JUCQ}}) = \quad & c_{\texttt{db}} + \sum_{1 \leq i \leq m} \left(c_{eval}(q_i^{\texttt{UCQ}}) \right) \\ & + \sum_{i, 1 \leq i \leq m, i \neq k} \left(c_{mat}(q_i^{\texttt{UCQ}}) \right) + c_{join}(q_{i, 1 \leq i \leq m}^{\texttt{UCQ}}) \\ & + c_{unique}(q^{\texttt{JUCQ}}) \end{aligned} \tag{1}$$

reflecting:

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

[8] See http://www-01.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.admin.cmd.doc/doc/r0005736.html.

(i) the fixed overhead of connecting to the RDBMS $c_{\texttt{db}}$;

(ii) the cost to *evaluate* each of its UCQ sub-queries $q_i^{\texttt{UCQ}}$;

(iv) the *materialization* costs: the SQL query corresponding to a JUCQ may have many sub-queries. At execution time, some of these subqueries will have their results materialized (i.e., stored in memory or on disk) while at most one sub-query will be executed in pipeline mode. We assume without loss of generality, that the largest-result sub-query, denoted $q_k^{\texttt{UCQ}}$, is the one pipelined (this assumption has been validated by our experiments so far); and

(v) the cost to *join* these sub-query results;

(vi) the cost of *eliminating* duplicates, in order to enforce our desired set semantics: from the results of each $q_i^{\texttt{UCQ}}$, and from the final results, by means of DISTINCT clauses. We found that this two-level elimination of duplicates lead to the best performance overall. Note that removing duplicates in the results of $q_i^{\texttt{UCQ}}$ does not break an evaluation pipeline, as those results were materialized anyway.

In the above, duplicates are eliminated because existing reformulation algorithms (and accordingly, our work) operate under *set semantics*.

Notations. For a given query q over a database db, we denote by $|q|_t$ the estimated number of tuples in q's answer set. Also, $q_{|\{a_i\}}$ stands for the restriction of q to its i-th atom. Using the notations above, the number of tuples in the answer set of $q_{|\{a_i\}}$ is denoted $|q_{|\{a_i\}}|_t$.

Duplicate elimination costs. Assuming duplicate elimination is implemented by hashing, we estimate the cost of eliminating duplicate rows from an SQL query $q^{\texttt{JUCQ}}$ and/or $q_i^{\texttt{UCQ}}$) as:

$$c_{unique}(q^{\texttt{JUCQ}}) = c_l \times |q^{\texttt{JUCQ}}|_t$$

where c_l is the CPU and I/O effort involved in sorting the results.

When the results are large enough to require disk merge sort, we estimate the cost of eliminating duplicate rows from $q^{\texttt{JUCQ}}$ (and $q_i^{\texttt{UCQ}}$ as a particular case) result as:

$$c_{unique}(q^{\texttt{JUCQ}}) = c_k \times |q^{\texttt{JUCQ}}|_t \times \log |q^{\texttt{JUCQ}}|_t$$

where c_k is the CPU and I/O effort involved in (disk-based) sorting the results.

UCQ **evaluation costs** are estimated by summing up the estimated costs of the CQs:

$$c_{eval}(q_i^{\texttt{UCQ}}) = \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} c_{eval}(q^{\texttt{CQ}})$$

The cost of evaluating *one* conjunctive query $c_{eval}(q^{\texttt{CQ}})$, where $q^{\texttt{CQ}}(\bar{x}) \leftarrow a_1 \wedge \cdots \wedge a_n$, through the RDBMS is estimated by analyzing the selections (known attribute values) in each atom of $q^{\texttt{CQ}}$, estimating (exactly – see below) how many triples match

these atoms, and *estimating the data access costs and the join costs together*. The data layouts we consider feature indexes on the relations storing class and role instances; as soon as the query selections and joins allow it, the RDBMS heavily relies on the indexes to simultaneously join and access the data i.e., the plan chains index-based accesses and index-based joins. Assuming efficient join algorithms such as hash- or merge-based etc are available [32], the join-only cost of $q^{\mathtt{CQ}}$ is linear in the total size of its inputs:

$$c_{join}(q^{\mathtt{CQ}}) = c_j \times \sum_{a_i \in q^{\mathtt{CQ}}} |q^{\mathtt{CQ}}_{|\{a_i\}}|_t$$

where c_j is a constant factor representing per-tuple join effort. Therefore, we have:

$$c_{eval}(q_i^{\mathtt{UCQ}}) = (c_t + c_j) \times \sum_{q^{\mathtt{CQ}} \in q_i^{\mathtt{UCQ}}} \sum_{a_i \in q^{\mathtt{CQ}}} |q^{\mathtt{CQ}}_{|\{a_i\}}|_t \tag{2}$$

where c_t is a constant representing the per-tuple I/O (access) effort.

`UCQ` **join cost**. As before, we consider the join cost to be linear in the total size of its inputs:

$$c_{join}(q^{\mathtt{UCQ}}_{i,1 \leq i \leq m}) = c_j \times \sum_{i=1}^{m} \sum_{q^{\mathtt{CQ}} \in q_i^{\mathtt{UCQ}}} \sum_{a_i \in q^{\mathtt{CQ}}} |q^{\mathtt{CQ}}_{|\{a_i\}}|_t \tag{3}$$

`UCQ` **materialization cost**. Finally, we consider the materialization cost associated to a query q is $c_m \times |q|_t$ for some constant c_m:

$$c_{mat}(q^{\mathtt{UCQ}}_{i,1 \leq i \leq m, i \neq k}) = c_m \times \sum_{i=1, i \neq k}^{m} \sum_{q^{\mathtt{CQ}} \in q_i^{\mathtt{UCQ}}} \sum_{a_i \in q^{\mathtt{CQ}}} |q^{\mathtt{CQ}}_{|\{a_i\}}|_t \tag{4}$$

where $q_k^{\mathtt{UCQ}}$ is the largest-result sub-query, and the one which is picked for pipelining (and thus not materialized).

Injecting the equations 2, 3 and 4 into the global cost formula 1 leads to the estimated cost of a given `JUCQ`. This formula relies on estimated cardinalities of various subqueries of the `JUCQ`, as well as on the system-dependent constants $c_{\mathtt{db}}$, c_l, c_k, c_j, c_t and c_m, which we determine by running a set of simple *calibration queries* (inspired by the approach of [16]) on the RDBMS being used; calibration details are straightforward and we omit them here.

For what concerns cardinality estimations, as in [29], RDBMS statistics provide, for each query atom, the exact number of triples matching it. Subsequently, textbook formulas are used to estimate the cardinality of more complex subqueries, based on statistics on the minimum and maximum value, and the number of distinct values in each attribute. We make the simple assumptions of uniform distribution of each attribute, and independent distributions among attributes. Any more refined RDF cardinality estimation technique, e.g., [28], could be used to improve the estimation accuracy.

6.3 Search space and EDL running time

We first studied the number of covers in $\mathcal{L}_q$ and $\mathcal{G}_q$ (recall Section 5). Our workload features some queries of 2 atoms, and the immediately larger ones have 6; we quickly realized that the number of generalized covers is prohibitively high for 6 or more atoms. To study this more closely, we derived from Q_1 a set of queries A_i, $3 \leq i \leq 6$, each of which is a star-join of i atoms on a common subject; in particular, A_6 is Q_1. Star queries are frequent over Semantic Web Data, as noted e.g., in [38, 8]. The sizes of the resulting search spaces are reported in Table 6; for A_6 we stopped the search at 20.003 generalized covers (there were more). This demonstrates that exploring the full $\mathcal{G}_q$ space is in general not feasible, as the overhead of examining so many options is prohibitive. Thus, in the sequel, we do not use EDL for our tests, as it is impractical beyond (very) small queries. Table 6 also shows the number of covers explored by the greedy GDL: these grow very moderately with the query size.

Finally, for A_3-A_6, *the running times of the best reformulation found by EDL and GDL (limited at* 20.000 *covers for* A_6*) coincided.* In general this is not guaranteed, but it is still an encouraging indicator of the good options found by GDL.

6.4 Evaluation time of reformulated queries

Figure 5 depicts the evaluation time, using Postgres with the simple layout, of four `FOL` reformulations:

1. the `UCQ` produced by the RAPID [14] reformulation engine;
2. the `JUCQ` reformulation based on C_{root};
3. the `JUCQ` reformulation corresponding to the best-performing (safe or generalized) cover, found by our algorithm GDL, using Postgres' cost estimation (*RDBMS*);
4. the `JUCQ` reformulation corresponding to the best-performing (safe or generalized) cover, found by our algorithm GDL, using our cost estimation (*ext*).

GDL running time is not reported in these graphs (see Section 6.5). We first analyze the top graph corresponding to LUBM$^{\exists}$ 15M. It shows, first, that the `UCQ` reformulation is inefficient (one order of magnitude slower than the best reformulation found, e.g., for Q_5 and Q_9). Second, the cover derived from C_{root} may also be very inefficient, in some cases (Q_6-Q_8, Q_{13}) much worse than the `UCQ`. These are both very large and complex queries; Figure 5 demonstrates that Postgres' optimizer called directly on the fixed-form reformulation may performed quite poorly. The GDL-selected covers, in contrast, *lead to the best-performing reformulations for all queries (often by an order of magnitude)*. Thus, our cost-based approach helps ask *the RDBMS the optimization question it can best answer*, among its equivalent formulations from the search space $\mathcal{G}_q$. Striking exceptions are Q_9, Q_{10} which have *both* many atoms and complex reformulations, and Q_{11} which has 2 atoms but the maximum number (667) of reformulations. Here, the GDL reformulations selected using the RDBMS cost model perform very poorly, whereas the ones based on our own cost estimation are much faster. This may be because Postgres takes drastic shortcuts when estimating the cost

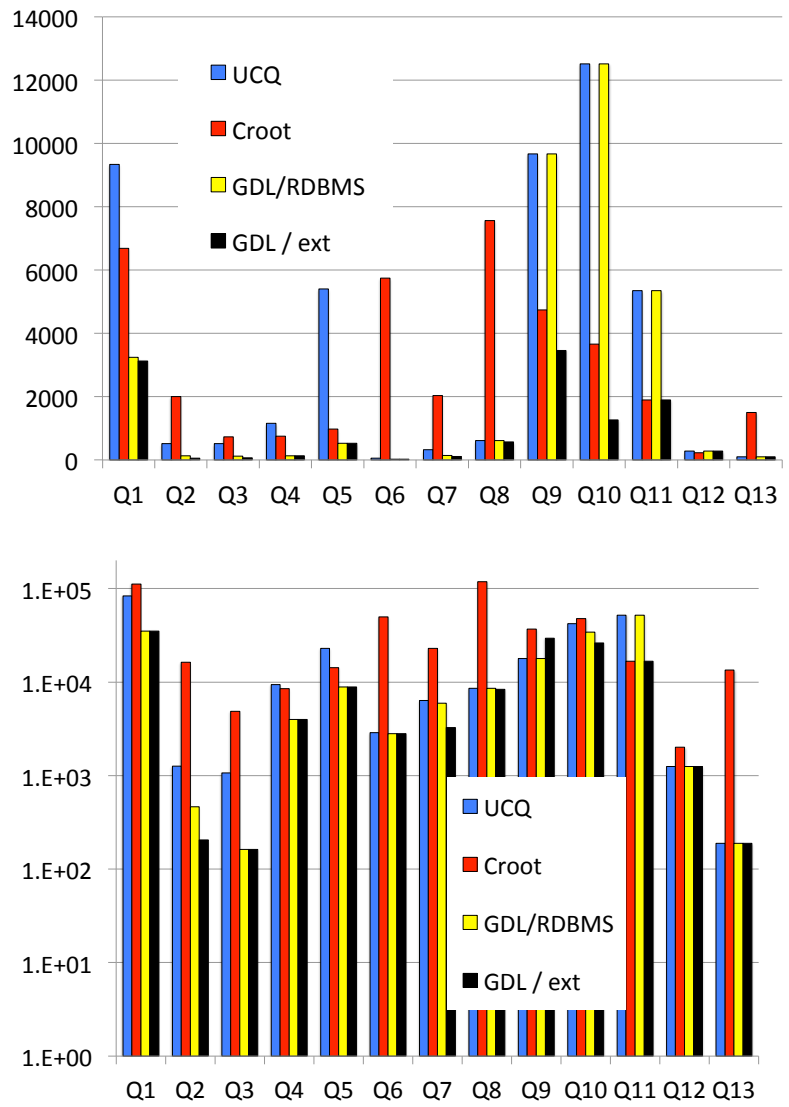


Figure 5: Evaluation time (ms) on Postgres on LUBM$^{\exists}$ 15M (top) and 100M (bottom).

of an extremely large query; in contrast, our cost estimation treats uniformly queries of all sizes. Recall that *Postgres' optimizer always has the last word in chosing how to evaluate the reformulation* we select, using its own cost model. Thus, the difference can only be attributed to the cost estimation.

The bottom graph in Figure 5 corresponds to LUBM$^{\exists}$ 100M; note the logarithmic y axis. Overall, the findings are the same: the `UCQ` and (especially in this case) the C_{root}

Query	A_3	A_4	A_5	A_6
$\|\mathcal{L}_q\|$	2	7	71	93
$\|\mathcal{G}_q\|$	4	67	5674	> 20000
$\mathcal{L}_q$ covers explored by GDL	2	5	11	18
$\mathcal{G}_q$ covers explored by GDL	4	12	27	59

Table 6: Search space sizes for queries A_3 to A_6.

reformulation perform poorly, while those picked by GDL are faster than the standard UCQ by a factor of up to 6.6 (Q_3).

Evaluation on DB2. The graph at the top of Figure 7 shows the evaluation time for DB2, on LUBM$^{\exists}$ 15M, of seven reformulations: the same four which we ran on Postgres, to which we add, on the RDF layout [8]: the UCQ reformulation, the one based on C_{root}, and the ones selected by GDL with the help of the RDBMS cost model. We did not code a cost estimation corresponding to this RDF-specific store, since (i) an accurate model of data access costs under such a complex layout (determined by running a linear programming solver etc.) seemed very hard to attain outside the server and (ii) DB2's cost model performed similarly to (or better than) ours for all the GDL-selected covers, on the simple layout. Thus, replacing it with our own seemed unlikely to improve the performance. Note the logarithmic y axis of the graph.

First, note that five bars are missing (replaced by the vertical lines), one for Q_9 and four for Q_{10}. They all correspond to reformulations against the RDF layout. The server error was "The statement is too long or too complex. Current SQL statement size is 2,247,118" for the UCQ of Q_9, and the same error (with similar query sizes) in the other cases. This shows that *the cummulated impact of, first, the DB2RDF storage layout (which leads to* `IF... THEN... ELSE` *and nesting in the SQL query corresponding to a simple* CQ*), and second, of ontology-based reformulation, yields queries too large for evaluation*. For illustration, the SQL versions of Q_1 before and after UCQ reformulation on DB2's RDF store appear at http://bit.ly/1TqeVMA. In cases where DB2 handled them, the reformulations corresponding to the UCQ, C_{root} and GDL on the RDF layout performed very poorly, up to 1 (UCQ) or even 4 (C_{root}) orders of magnitude worse than the best reformulations identified. Thus, our (somehow unexpected) conclusion is that the RDF-specific layout, while interesting for CQ evaluation, is not the best alternative when evaluating queries issued from reformulation against an ontology.

Focusing only on the simple layout, we see that the cost-unaware UCQ and C_{root}-derived reformulations perform again poorly, while the GDL ones perform best and in many cases coincide. The two cost estimations behaved mostly the same, except that our estimation worked better for Q_8 and worse for Q_9. Overall, our chosen reformulations lead to performance gains of up to a factor of 9 w.r.t. the UCQ and/or C_{root} on the simple layout, for which we found DB2's cost estimation quite reliable.

At the bottom of Figure 7 we show the evaluation times on LUBM$^{\exists}$ 100M for the first eight among the ten reformulations shown in the top graph (we gave up GDL on the RDF layout, given our experience on the smaller dataset). The four execution errors (grey vertical lines) on the UCQ and C_{root} reformulations on the RDF layout are again due to overly large SQL queries. The first four alternatives are

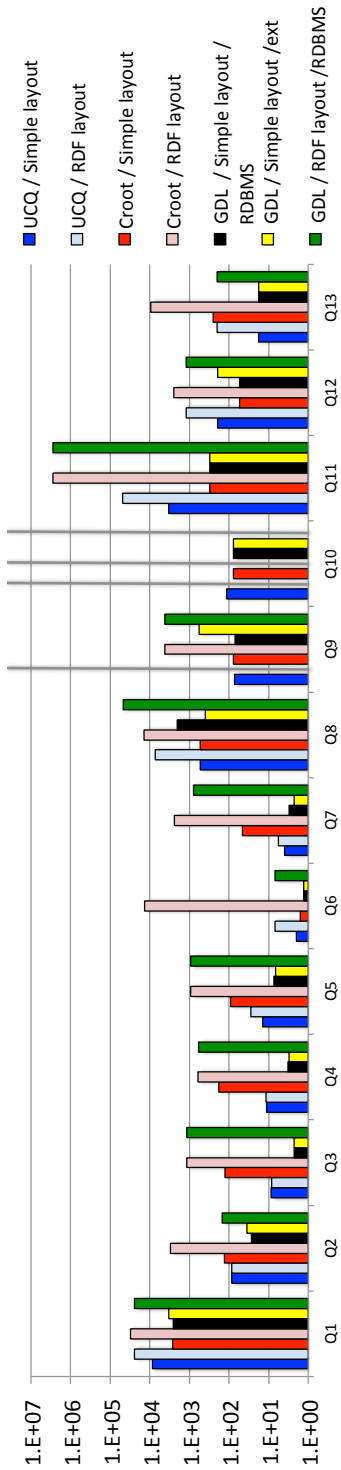


Figure 6: First caption

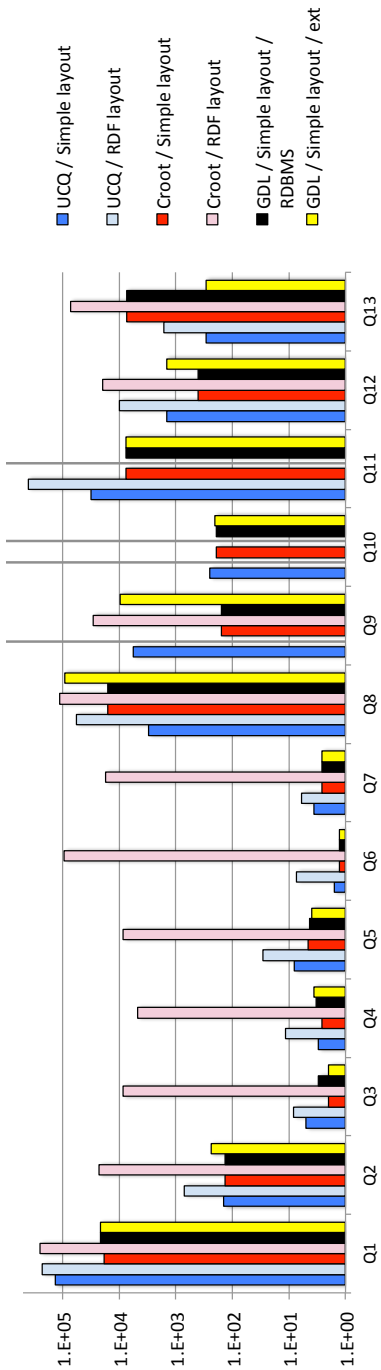


Figure 7: Evaluation time (ms) on DB2 and LUBM$^{\exists}$ 15M (top) and 100M (bottom).

overall the worse, with C_{root} and at a lesser extent UCQ on the RDF layout performing very poorly. When focusing on the simple layout only, we notice that the cost-based reformulations improve over the simple UCQ performance by a factor of up to 36 (4.85 on average). There is an exception for Q_8, where the UCQ was best; in this case, both DB2's and our cost estimations were inaccurate, which we believe cannot be avoided in all cases. DB2's estimation lead to significantly better reformulations than ours for the queries Q_2, Q_8, Q_9 and Q_{12}, while our cost model was clearly better for Q_{13}. Overall, we found DB2's cost estimation more accurate than our own (while the opposite holds for Postgres). By inspecting query plans, we confirmed that DB2 and Postgres do not apply any CSE across union terms. The better performance of DB2 is likely due to efficient runtime support for repeated scans [22].

In all experiments presented in this section, GDL ran between 1 ms (for 2-atom queries) to 207 ms (for the larger Q_1); we discuss the running time of our optimization approach in more detail in Section 6.5.

Finally, always (when using our cost model) and about half of the time (with the RDBMS cost model), GDL picked a generalized cover. This confirms the interest of searching in the $\mathcal{G}_q$ space.

6.5 Time-limited GDL

Figure 8 shows the running time of algorithm GDL on the LUBM datasets of 15M and 100M triples, in four configurations: using no cost estimation (this artificial case where all costs are estimated to 0 was built to measure our algorithm's running time independently of the cost estimation time); using our own cost estimation (described in Section 6.2); using the cost estimation of Postgres; and finally, using the estimation of DB2. Note that the vertical axis is in logarithmic scale. The two graphs are similar, which is to be expected given that we measure optimization time, which is not (strongly) impacted by the data sizes. The times using Postgres' and DB2's cost estimations are not identical: internal heuristics in the Postgres and DB2 systems may have led to different plans shapes being explored for the different database sizes.

We make the following observations.

1. The running time of GDL without any cost estimation is very small, bounded by 23 ms.

2. Using our cost model has a discernible yet still small overhead, bringing the total running time of our optimization technique to about 207 ms.

3. Using Postgres' cost estimation time incurs a significant overhead, going up to 10, 100 and even (in the pathological case of Q_5) 1000 seconds, which is prohibitive for an optimization step.

4. Using DB2's cost estimation is for many queries even more expensive. This is because the db2expln utility requires large queries to be written into an OS file given as parameter to the cost estimation, whose output must then be extracted from the detailed information db2expln returns. This is more expensive than Postgres' provided explainer functionality, which is accessible through the JDBC driver, without the need to make a runtime call from our Java optimizer code etc.

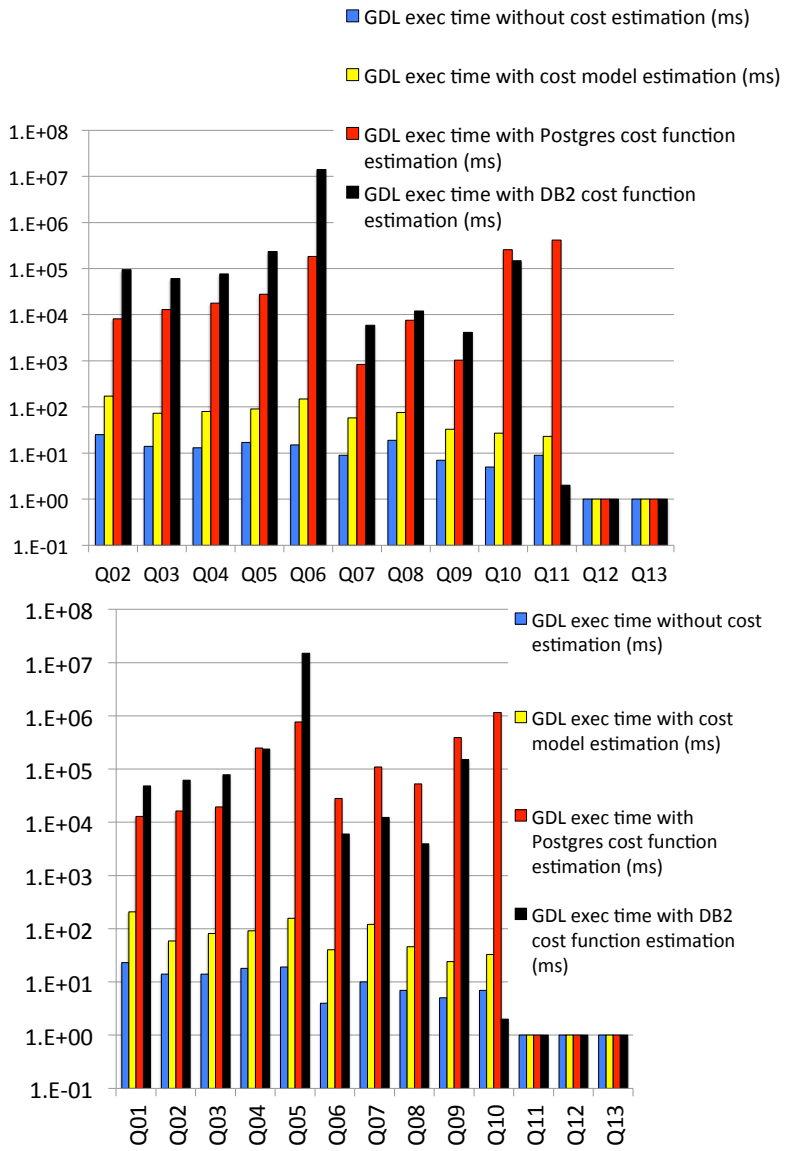
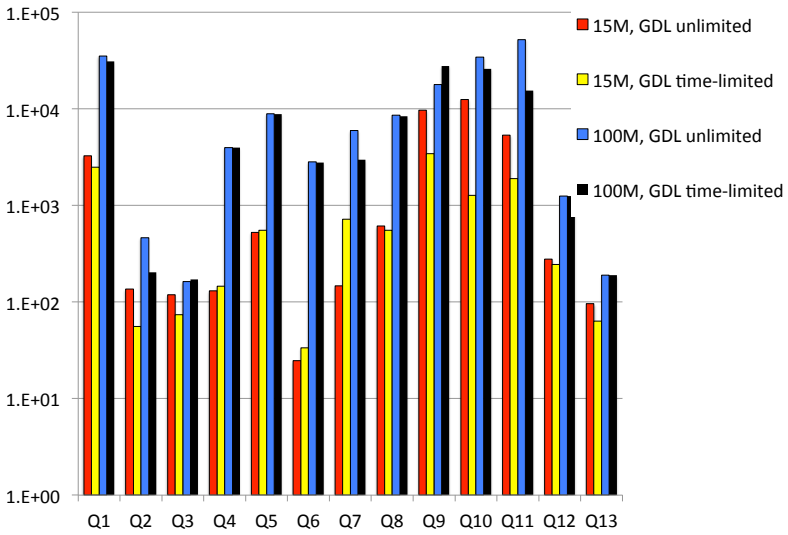


Figure 8: GDL running time (ms) on LUBM$^{\exists}$ 15M (top) and 100M (bottom).

5. GDL including DBMS cost estimation is visibly correlated with the size of the query; note the peaks for Q_5, Q_9 and Q_{10}, which are the most complex (as shown in Table 3).

The graphs show that it is clearly preferrable to run GDL in a context where the cost estimation function is accessible without a high overhead. This was the case when using our own estimation, while the estimations of Postgres and especially DB2 were



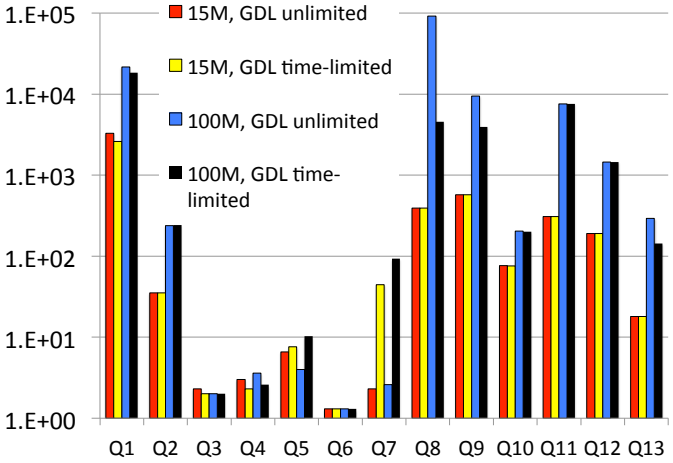


Figure 9: Query evaluation time of GDL-selected covers, without time limits, and limited to 20 ms for Postgres (top) and DB2 (bottom).

harder for us to access.

Time-limited GDL. Therefore, we investigated a *time-limited* version of GDL, which was allowed to explore only during 20 ms. Figure 9 compares the running time of

the cover found by GDL after only 20 ms, with that of the cover found by GDL allowed to run to completion. We see that the running times are very close for Postgres, and also generally close for DB2, demonstrating that interesting covers are quickly found. This is because on our queries, the strongest reduction (mostly through reducing intermediary result sizes) are identified early during the greedy search, thus most of the performance benefits can be reaped early on. More generally, this corresponds to the good behavior of greedy algorithms when there are very advantageous moves to be made. Thus, we find time-limited GDL performs well in practice, for a modest overhead.

Against the expectation, in some cases, the limited GDL performed better than the unlimited one (for instance, on Q_5 and Q_7 on DB2 in Figure 9). This is an accident due to our cost model; it turns out that in these cases, the longer search ended up recommending a state whose cost was slightly worse.

6.6 Experiment conclusions

Our experiments show that plain UCQ reformulation is evaluated poorly by both Postgres and DB2, even more so (or even fails) on DB2's RDF-specific data layout. On the simple layout, the fixed cover-based reformulation corresponding to the root cover C_{root} also performs very poorly. In contrast, GDL-selected reformulation improve over the UCQ in all 13 queries $\times 2$ systems $\times 2$ datasets but one, and they do so by up to a factor of 36. Our cost estimation helped w.r.t. Postgres' `explain`, but when using DB2, we find `db2explain`'s estimation more accurate overall.

The generalized cover space has prohibitive size, thus EDL is impractical. In contrast, our greedy GDL is efficient when used with a low-overhead cost estimation (such as the one we implemented), and effective in optimizing reformulated queries. GDL attains most of its cost reductions early on during the search, making it a robust tool for improving reformulated query answering performance.

7 Related work and conclusion

We proposed a novel framework for any OBDA setting enjoying FOL reducibility of query answering, for which we studied a *space of alternative* FOL *reformulations to evaluate through an RDBMS*. We applied this framework to the DL-Lite$_\mathcal{R}$ description logic, and experimentally demonstrated its performance benefits.

7.1 Relationship with prior work on reformulation-based query answering

Our approach departs from the literature focused on a *single* FOL query reformulation, where optimization mainly reduces to *producing fast a* UCQ *reformulation as minimized as possible*: [14, 15, 36, 21, 37, 30, 19] consider DL-Lite$_\mathcal{R}$, existential rules and Datalog$^\pm$. [35] 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 these works to devise CQ-to-JUCQ and CQ-to-JUSCQ reformulation techniques, and used cost estimations to speed up reformulated query evaluation.

In particular, our generalized covers can be seen as adapting semijoin-based reducers to the query answering setting. [33] proposes a cost-unaware CQ-to-Datalog reformulation technique; it produces a *non-recursive* Datalog program, which amounts to a JUCQ.

One contribution of this work is an optimization framework (Section 3) for any formalism for which query answering is FOL-reducible, e.g., some Description Logics, Datalog$^{\pm}$ and Existential Rules fragments. Our previous work [9] is a particular case of this framework for the RDFS ontology language, which corresponds only to the constraints 1, 4, 5 and 11 from Table 2.1, while the DL-Lite$_{\mathcal{R}}$ language we use comprises 22 such constraints. When reformulating under this rich language, some covers are unsafe (recall Example 7), while in [9] any cover leads to a correct query reformulation for the 4 constraints considered there. DL-Lite$_{\mathcal{R}}$ is important as it is provides the foundations for W3C's standard for *very large* Semantic Web data management OWL2 QL. Thus, the other contributions of our work are: to identify and characterize *safe covers*, guaranteed to lead to reformulations, and a carefully chosen extra space of *generalized covers* which lead to equivalent FOL reformulations and often improve query performance. Our EDL and GDL optimization algoritms (Section 5.3) respectively explore exhaustively and greedily this DL-Lite$_{\mathcal{R}}$-specific space to speed up reformulation-based query answering under DL-Lite$_{\mathcal{R}}$ constraints. Another difference w.r.t. [9] is that this work explores the usage of DB2's RDF store, and find it unsuitable to the complex queries resulting from reformulation.

In the database and Semantic Web communities, there have been intense efforts invested in developing scalable RDF data management platforms, including distributed ones; see e.g., the survey [20]. However, these platforms do not take constraints into account, and thus only support query evaluation, not query answering. Our work is the first to consider optimized algorithms for answering queries under DL-Lite$_{\mathcal{R}}$ constraints through relational databases.

7.2 Relationship with prior work on multi-query optimization (MQO)

Generally speaking, the relationship can be stated as follows. MQO is interesting as soon as the query to be evaluated has redundant (repeated) subexpressions. In our setting, we distinguish:

Reformulation-induced redundancy refers to the repeated subexpressions appearing in a reformulated query due to the reformulation itself. The UCQ has most such redundancies, as illustrated e.g., in Table 3 in the paper.

Reformulation-independent redundancy designates the redundancy that a query may have regardless of the impact of reformulation; for instance, a query may feature repeated subexpressions prior to reformulation, even if the TBox is empty etc.

MQO vs. reformulation-independent redundancy. Any SQL processor capable of MQO can be profitably used to evaluate the cover-based reformulations we chose, in order to *diminish or eliminate reformulation-independent redundancy*. As mentioned in Section 2.3, the was no algebra-level MQO available in the free and commercial

engine used in our study, thus our estimation for a cover-based reformulation cost does not take it into account (although some partial support is provided e.g., in Oracle [6]).

One reason why MQO performed during query optimization is complex is the difficulty of deciding equivalence between two logical expressions; under the set semantics used in our work, this is NP-hard even for CQ. Works such as [25, 6] use *syntactic equality* of plans, as a criterium for deciding *semantic equivalence*; this is sufficient in [25] because of the focus restricted on select-group by queries over a single table, and in [6] because *under bag semantics, equivalent conjunctive queries are isomorphic* [13]. Under set semantics, both approaches would lead to missing many equivalences, and thus many sharing opportunities. In [27], a similar syntactic condition is used for efficiency, knowing that it may miss sharing opportunities.

If MQO-enabled systems became available, the cost estimation should also reflect its presence; this would be immediately the case for the system's own cost estimation, and would require a revision of our cost estimation function.

We view the possible usage of an MQO-capable SQL engine *to handle reformulation-independent redundancy in the query* as orthogonal to our work. To avoid unwanted impact on our study, *none of the queries used in our study featured reformulation-independent redundancy.*

MQO vs. reformulation-induced redundancy. Reformulating a CQ against a TBox may introduce many repeated subexpressions. The UCQ has most such redundancies, as illustrated e.g., in Table 3 in the paper.

However, our technique applies *before the common subexpression factorization stage*; in the terms of [25], our work belongs to the *strategical* optimization stage, which injects application knowledge and constraints into the optimization problem. Unlike the setting envisioned in [25], however, we do not use such knowledge to create *one plan*, but *several reformulations*, each of which can be seen mid-way between a plan and a query. Indeed, while a cover-based reformulation is still a query, it does make some ordering decisions, in particular each reformulated fragment query is evaluated, and all but one are materialized, before the cover-based reformulation evaluation is finalized by a join.

Thus, our approach, which starts with the TBox and data statistics, and ends by handing over a chosen reformulation to the RDBMS, *never requires work to detect common (repeated) sub-expressions*. Instead:

1. In the simpler setting of an RDFS TBox [9], the so-called SCQ reformulation (which pushes all unions immediately above the scans [35]) has the least possible repeated sub-expressions. For instance, on a query of two atoms a_1, a_2, such that a_1 reformulates into a_1^1 and a_2 reformulates into a_2^1 and a_2^2, the SCQ reformulation is:

$$(a_1 \vee a_1^1) \wedge (a_2 \vee a_2^1 \vee a_2^2)$$

while the UCQ (featuring many repeated scans) is:

$$(a_1 \wedge a_2) \vee (a_1 \wedge a_2^1) \vee (a_1 \wedge a_2^2) \vee (a_1^1 \wedge a_2) \vee (a_1^1 \wedge a_2^1) \vee (a_1^1 \wedge a_2^2)$$

(For larger queries, the UCQ also features repeated joins.)

2. *In the context of the present work, the* SCQ *may not be a FOL reformulation*. When redundancy conflicts with correctness, we should clearly favor correctness first. The safe cover C_{root} is the closest approximation of the SCQ in our context, as it applies the least amount of atom merging within fragments.

   Fortunately, C_{root} is very efficient to build from the TBox and the query: the complexity is $O(|q|^2)$, since we need to compare all the pairs of atoms to see whether they depend on a same TBox predicate. This contrasts with the very high complexity of the abovementioned algebraic MQO techniques [39, 27].

Interestingly, an Oracle 10g [3] reference mentions MQO rewritings which factorize common join expressions across union terms. This would apply to the UCQ setting, however, as mentioned above, based on the TBox, we can much more efficiently minimize redundancy by chosing the C_{root} cover. Also, according to [3], MQO exploration would have to be drastically cut for queries with hundreds of union terms. Thus, our approach is more practical as it exploits information about *the source of redundancy* (namely the reformulation process using the ontology). In contrast, an optimizer has to *deal with the consequences* (detect redundant subexpressions).

The abovementioned Oracle work [3] also mentions query transformations (rewrites) applied at the level of the syntax, in a bottom-up fashion; to learn if a certain subplan rewriting is profitable, the optimizer's cost estimation is used. This strategy of preprocessing of the query guided by the DBMS cost estimator is similar in spirit to our approach (Figure 1), and also comparable to the "strategical optimization" stage [25] which applies first in the query optimization process, and injects data and application semantics into the plan.

Our approach could be profitably integrated as a rewrite specific to ontology-based reformulation, within a strong cost-based optimizer such as the one of Oracle [3]. In particular, this would give us access to more sophisticated query cost estimations, while eliminating the overhead we currently pay to get them through a connection to the server.

Currently, we lack access to strong industrial optimizers such as the ones of DB2, SQL Server or Oracle; Postgres' optimizer is easier to extend, but our experiments have shown it is weaker than DB2's.

Other well-known MQO/CSE algorithms have been described e.g., in [17, 25, 27, 39]. Oracle includes several query rewrites to improve nested query performance, among which *subquery coalesce* reduces some redundancy and thus can be seen as related to CSE [6]. A different class of techniques [22, 40] improve the performance of multiple concurrent reads of a table; this can be seen as a physical-level MQO only applying to one-table plans. Such techniques are implemented in DB2, and they indeed help evaluating our reformulations. However, as stated in Section 2.3, our approach does not require detecting repeated subexpressions.

We plan to extend our framework to efficient query answering *using materialized* CQ *views*, which may partially or completely rewrite the CQs appearing in the reformulated fragments.

References

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

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

[3] R. Ahmed, A. Lee, A. Witkowski, D. Das, H. Su, M. Zait, and T. Cruanes. Cost-based query transformation in Oracle. In *VLDB*, 2006.

[4] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The DL Handbook: Theory, Implementation, and Applications*. Cambridge Univ. Press, 2003.

[5] J. Baget, M. Leclère, M. Mugnier, and E. Salvat. On rules with existential variables: Walking the decidability line. *Artificial Intelligence*, 175(9-10), 2011.

[6] S. Bellamkonda, R. Ahmed, A. Witkowski, A. Amor, M. Zait, and C.-C. Lin. Enhanced subquery optimizations in Oracle. *PVLDB*, 2(2), Aug. 2009.

[7] P. A. Bernstein and D. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1), 1981.

[8] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *SIGMOD*, 2013.

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

[10] A. Calì, G. Gottlob, and T. Lukasiewicz. Datalog$^{\pm}$: a unified approach to ontologies and integrity constraints. In *ICDT*, 2009.

[11] A. Calì, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *PODS*, 2009.

[12] 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*, 39(3):385–429, 2007.

[13] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *PODS*. ACM, 1993.

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

[15] 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 *ORE Workshop*, 2012.

[16] G. Gardarin, F. Sha, and Z. Tang. Calibrating the query optimizer cost model of IRO-DB, an object-oriented federated database system. In *VLDB*, 1996.

[17] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared workload optimization. *PVLDB*, 7(6), 2014.

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

[19] G. Gottlob, G. Orsi, and A. Pieris. Query rewriting and optimization for ontological databases. *ACM TODS*, 39(3):25, 2014.

[20] Z. Kaoudi and I. Manolescu. RDF in the Clouds: A Survey. *The International Journal on Very Large Databases*, June 2014.

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

[22] C. A. Lang, B. Bhattacharjee, T. Malkemus, S. Padmanabhan, and K. Wong. Increasing buffer-locality for multiple relational table scans through grouping and throttling. In *ICDE*, April 2007.

[23] M. Lenzerini. Ontology-based data management. In *CIKM*, 2011.

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

[25] S. Manegold, A. Pellenkoft, and M. L. Kersten. A multi-query optimizer for Monet. In *BNCOD*, 2000.

[26] M. Mugnier. Ontology-based query answering with existential rules. In *RuleML*, 2012.

[27] T. Neumann and G. Moerkotte. Generating optimal DAG-structured query evaluation plans. *Computer Science - R&D*, 24(3), 2009.

[28] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, 2011.

[29] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDBJ*, 19(1):91–113, 2010.

[30] G. Orsi and A. Pieris. Optimizing query answering under ontological constraints. *PVLDB*, 4(11), 2011.

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

[32] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., NY, USA, 3 edition, 2003.

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

[34] K. Stocker, D. Kossmann, R. Braumandl, and A. Kemper. Integrating semi-join-reducers into state of the art query processors. In *ICDE*, 2001.

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

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

[37] 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.

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

[39] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, 2007.

[40] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a DBMS. In *VLDB*, 2007.

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

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

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

ISSN 0249-6399