J Data Semant (2014) 3:1–23
DOI 10.1007/s13740-012-0017-6

ORIGINAL ARTICLE

Query Extensions and Incremental Query Rewriting for OWL 2 QL Ontologies

Tassos Venetis · Giorgos Stoilos · Giorgos Stamou

Received: 24 February 2012 / Revised: 3 December 2012 / Accepted: 9 December 2012 / Published online: 8 January 2013

Abstract *Query rewriting* over lightweight ontologies, like DL-Lite ontologies, is a prominent approach for ontology-based data access. It is often the case in realistic scenarios that users ask an initial query which they later refine, e.g., by extending it with new constraints making their initial request more precise. So far, all DL-Lite systems would need to process the new query from scratch. In this paper, we study the problem of computing the rewriting of an extended query by 'extending' a previously computed rewriting of the initial query and avoiding recomputation. Interestingly, our approach also implies a novel algorithm for computing the rewriting of a fixed query. More precisely, the query can be 'decomposed' into its atoms and then each atom can be processed incrementally. We present detailed algorithms, several optimisations for improving the performance of our query rewriting algorithm, and finally, an experimental evaluation.

Keywords Query rewriting · DL-Lite · Incremental query rewriting · Query extension

This is a revised and considerably extended version of the workshop paper [33].

T. Venetis (✉) · G. Stoilos · G. Stamou
School of Electrical and Computer Engineering,
National Technical University of Athens,
Zographou Campus, 15780 Athens, Greece
e-mail: avenet@image.ece.ntua.gr

G. Stoilos
e-mail: gstoil@image.ece.ntua.gr

G. Stamou
e-mail: gstam@cs.ece.ntua.gr

1 Introduction

Efficiently managing and retrieving large amounts of data is a key problem for many modern applications. Ontologies expressed in the W3C's Web Ontology Language OWL [15] and its revision OWL 2 [10] are often used for providing a formal and unified schema that describes the data that are stored in distributed and/or heterogeneous data sources. A key application for such systems is ontology-based data access (OBDA) [29], where answers to user queries reflect both the data as well as the ontology that describes them. However, reasoning and conjunctive query (CQ) answering in the ontology languages OWL 2 DL, OWL DL, and OWL Lite is of very high computational complexity in the worst case [20,25]. Actually, no complete algorithm for querying OWL 2 DL ontologies is currently known.

The need for efficient query answering has motivated the development of many *lightweight* ontology languages which provide reasoning services of at most polynomial data complexity. One such language is DL-Lite [2,8]. DL-Lite forms the logical underpinnings of the ontology language OWL 2 QL, a well-known profile of OWL 2 [23]. Query answering in DL-Lite is usually performed via a technique called *query rewriting* [2,8,28]. According to this technique a query q and a DL-Lite ontology are transformed into another query q', called a *rewriting*, such that the answers of q' over the input data and discarding the ontology are precisely the answers of q over the data and the ontology. Common target languages for representing rewritings in DL-Lite are non-recursive Datalog [13,32] or union of conjunctive queries (UCQs) [8,9,27].

The nice properties of DL-Lite have motivated the development of many algorithms and systems for computing a rewriting, such as QuOnto [1], Requiem [27], Presto [32], Nyaya [11], Quest [30] and Rapid [9]. For a given query most of these systems will compute a rewriting by applying (usually in a brute-force manner) a certain set of equivalence-preserving transformations to the query discarding any information computed for previously rewritten queries. However, it is quite often the case that user queries have very small differences with previously executed ones. For example, in Web search scenarios, it has been shown that users usually first ask some 'general' query and then, according to the returned results, they refine it by adding further constraints making their request more precise [16,17,26]. Consequently, a query can be refined several times until the user (possibly) finds the intended information.

For example, a user might initially ask to retrieve from a student database all those students who are athletes using the following conjunctive query:

$$\{x \mid \mathsf{Student}(x), \mathsf{Athlete}(x)\}$$

where x is a variable that needs to be replaced with actual students from the database while Student and Athlete are concept atoms (unary predicates). Then, the user can refine the search by requesting only those athletes who are female by extending the previous query with the new concept atom $\mathsf{Female}(x)$ giving raise to the following query:

$$\{x \mid \mathsf{Student}(x), \mathsf{Athlete}(x), \mathsf{Female}(x)\}.$$

Finally, the new query can be further extended requesting only those female athlete students who take a specific course. This can be done by adding the role atom (binary predicate) $\mathsf{takesCourse}(x, y)$. In all these cases all aforementioned DL-Lite systems would compute a rewriting for the extended queries by running their algorithm each time from scratch discarding any information computed during previous runs. However, due to the overlap between the queries much of the previously computed information would need to be re-computed by each system.

In the current paper, we study the problem of computing a rewriting for queries that have been extended with new atoms. More precisely, given a DL-Lite ontology, a query, a rewriting computed (possibly previously) for this query and a new atom that would extend the original one, we study how to compute a rewriting for the new query by "extending" the input rewriting and avoid computing one from scratch. We study the problem theoretically and design a detailed practical algorithm. Roughly speaking, the algorithm computes a rewriting for a query that is relevant *only* to the newly added atom and then combines this with the input rewriting using proper operations. This way the algorithm performs *only* the additional work that is required to compute a rewriting for the extended query.

Interestingly, our techniques for rewriting extended queries imply a novel approach for computing a rewriting for *fixed* queries. More precisely, given a (fixed) query one can pick one of its atoms compute a rewriting for it and then iteratively add the rest of the atoms by extending the previously computed rewriting. When all the atoms of the input query have been processed a rewriting for the given query would have been computed. Based on this idea, we present a detailed query rewriting algorithm for conjunctive queries over DL-Lite ontologies. Subsequently, in order to improve its efficiency we also present several novel optimisations.

Finally, we have implemented all the proposed algorithms and we have conducted an extensive experimental evaluation comparing our algorithms against several available state-of-the-art rewriting systems. The evaluation shows that computing a rewriting incrementally is in the vast majority of cases faster than the currently fastest DL-Lite system. Especially, the comparison with the original DL-Lite algorithm [8] with which our algorithm shares many common features showed that the new algorithm is several orders of magnitude faster. This can be justified by the more guided rewriting strategy that processes each atom at a time in contrast to the brute-force approach followed by most existing systems. In summary, our paper makes the following major contributions:

- It studies the problem of computing a rewriting for queries that have been extended with new atoms given a rewriting for them.
- It shows how the reduction step of the original algorithm for DL-Lite [8] can be optimised in order to avoid an exponential blow-up of redundant queries.
- It presents a novel query rewriting algorithm that incrementally processes the atoms of the query.
- It presents several optimisations by which the performance of the incremental query rewriting algorithm can be significantly improved.
- It provides an extensive experimental evaluation of the proposed algorithms, also contrasting them with existing state-of-the-art query rewriting systems.

Besides the immediate practical benefits, our study has several theoretical consequences and gives many interesting opportunities for future research. First, it shows that rewriting in DL-Lite can largely be performed in parallel. That is, one can complete *all* "rewriting" work for each atom independently and then combine the results without referring to the rewriting module again. To the best of our knowledge, this is the first complete and practical algorithm for query rewriting that is based on this approach. Second, the experimental evaluation shows that with relatively few optimisations incremental rewriting behaves very well in practice. Hence, it would be interesting to apply such a strategy over other ontology languages such as Linear-Datalog$^{\pm}$ [11] or OWL 2 EL [23].

2 Preliminaries

In this section, we first briefly recall some notions from graph theory. Then, we present the Description Logic DL-Lite$_R$ [8], a lightweight knowledge representation formalism which consists of the logical underpinnings of the ontology language OWL 2 QL. Subsequently, we define the syntax and semantics of conjunctive queries and unions of conjunctive queries and finally, we provide a brief overview of the so-called PerfectRef query rewriting algorithm for DL-Lite$_R$ ontologies [8].

2.1 Directed Graphs

A *(directed) graph* is a tuple $G = \langle V, E\rangle$, where V is a set of vertices and $E \subseteq V \times V$ is a binary relation over V. We often abuse notation and use G to refer to the set of edges of the graph; in that case it is understood that V is exactly the set of endpoints of elements of E. Hence, adding a pair $\langle a, b\rangle$ to G creates a new graph $G = \langle V \cup \{a, b\}, E \cup \{\langle a, b\rangle\}\rangle$.

For $a, b \in V$, we say that b is *reachable* from a in G, written $a \rightsquigarrow_G b$, if $c_0, \dots, c_n$ with $n \geq 0$ exist where $c_0 = a$, $c_n = b$ and $\langle c_i, c_{i+1}\rangle \in E$ for each $0 \leq i < n$. Note that, according to this definition, each $c \in V$ is reachable from itself. Finally, an element $c \in V$ is called *top* in G if for each $c' \in V$ we have $c \rightsquigarrow_G c'$.

2.2 The DL-Lite$_R$ language

A DL-Lite$_R$ signature is the disjoint union of a countably infinite set $\mathbf{C}$ of *atomic concepts* (unary predicates), $\mathbf{R}$ of *atomic roles* (binary predicates) and $\mathbf{I}$ of *individuals* (constants). A DL-Lite$_R$*-role* is either an atomic role $P \in \mathbf{R}$ or its *inverse* P^-. Let $A \in \mathbf{C}$ be an atomic concept and R a DL-Lite$_R$-role; then, a DL-Lite$_R$*-concept* is either an atomic concept A or a concept of the form $\exists R$.

Let B_i be DL-Lite$_R$-concepts and R_i be DL-Lite$_R$-roles. A DL-Lite$_R$-TBox, denoted by $\mathcal{T}$, is a finite set of axioms of one the following forms:

$$B_1 \sqsubseteq B_2 \qquad R_1 \sqsubseteq R_2$$

An *ABox* is a finite set of *assertions* of the form $A(c)$ or $P(c, d)$ for $A \in \mathbf{C}$, $P \in \mathbf{R}$ and $c, d \in \mathbf{I}$. A DL-Lite$_R$-ontology $\mathcal{O} = \mathcal{T} \cup \mathcal{A}$ is the union of a TBox and an ABox.[1] In the following, since we are concerned with a particular DL language, we simply speak of roles, concepts, TBoxes, and ABoxes referring only to those expressible in DL-Lite$_R$.

[1] Note that, in DL-Lite$_R$ one can also allow for concept (role) disjointness axioms of the form $B_1 \sqsubseteq \neg B_2$ ($R_1 \sqsubseteq \neg R_2$). However, these axioms do not have any effects in query *answering* when $\mathcal{T} \cup \mathcal{A}$ is consistent [8,28]; hence we will discard them here.

Table 1 The translation of DL-Lite$_R$-axioms and ontologies into FOL sentences. Note that, B is a concept, R is a role, and a, b are individuals.

$\pi(B_1 \sqsubseteq B_2)$	$=$	$\forall x.(\pi_x(B_1) \rightarrow \pi_x(B_2))$
$\pi(R_1 \sqsubseteq R_2)$	$=$	$\forall x, y.(\pi_{x,y}(R_1) \rightarrow \pi_{x,y}(R_2))$
$\pi(A(c))$	$=$	$A(c)$
$\pi(P(c, d))$	$=$	$P(c, d)$
$\pi(\mathcal{O})$	$=$	$\bigwedge_{\alpha \in \mathcal{O}} \pi(\alpha)$

Table 2 The translation into FOL formulas

$\pi_x(B)$	$=$	$B(x)$
$\pi_{x,y}(R)$	$=$	$R(x, y)$
$\pi_{x,y}(R^-)$	$=$	$R(y, x)$
$\pi_x(\exists R)$	$=$	$\exists y.\pi_{x,y}(R)$

DL-Lite$_R$ can be seen as a fragment of First-Order Logic (FOL); thus, it can be given formal semantics via a translation to FOL, where concepts are translated into formulae with one free variable, roles into formulae with two free variables, axioms into FO-sentences, and TBoxes, ABoxes and ontologies into FO-sentences. The transformation $\pi(\cdot)$ for axioms and ontologies is given in Table 1, while the translation of concepts is defined as given in Table 2.

An ontology $\mathcal{O}$ is *consistent* if $\pi(\mathcal{O})$ has a model, while entailment ($\models$) is defined as usual in FOL.

2.3 Conjunctive Queries

We use standard notions of (function-free) term, variable and substitution from First-Order Logic. For α an atom and σ a substitution, the result of applying σ to α is denoted as α_σ. Moreover, we use the notation $\mathsf{dom}(\sigma)$ to denote the domain of a substitution σ. Furthermore, every substitution σ induces a directed graph $G = \langle V, E\rangle$, where $t \in V$ iff t is a term in σ and $\langle x, t\rangle \in E$ iff $x \mapsto t \in \sigma$.

A *concept atom* is of the form $A(t)$ with A an atomic concept and t a term. A *role atom* is of the form $R(t, t')$ for R an atomic role, and t, t' terms. A *conjunctive query* (CQ) q is an expression of the form:

$$\{\mathbf{x} \mid \alpha_1, \dots, \alpha_m\}$$

where $\mathbf{x} = (x_1, \dots, x_n)$ is a tuple of variables called *distinguished* (or *answer*) and each α_i is a concept or role atom called *body atom*. Each distinguished variable appears in at least some atom α_i; all other variables of the query are called *undistinguished*. A variable that is either distinguished or appears in at least two different atoms α_i, α_j with $i \neq j$ in q is called *bound*, otherwise it is called *unbound*. For an atom α, we use $\mathsf{var}(\alpha)$ to denote the set of its variables; var can be extended to queries in the obvious way. Moreover, by $\mathsf{avar}(q)$ we denote all the distinguished variables of q.

Table 3 Function gr defined for an atom α and an axiom I

if $\alpha = A(x)$ and

1.$I = B \sqsubseteq A$, then $\mathsf{gr}(\alpha, I) = B(x)$;

2.$I = \exists P \sqsubseteq A$, then $\mathsf{gr}(\alpha, I) = P(x, y)$ for y a new variable in q;

3.$I = \exists P^- \sqsubseteq A$, then $\mathsf{gr}(\alpha, I) = P(y, x)$ for y a new variable in q

if $\alpha = P(x, z)$, z is unbound in q and

1. $I = A \sqsubseteq \exists P$, then $\mathsf{gr}(\alpha, I) = A(x)$;

2. $I = \exists S \sqsubseteq \exists P$, then $\mathsf{gr}(\alpha, I) = S(x, y)$ for y a new variable in q;

3. $I = \exists S^- \sqsubseteq \exists P$, then $\mathsf{gr}(\alpha, I) = S(y, x)$, for y a new variable in q.

if $\alpha = P(z, x)$, with z unbound, then

1.$I = A \sqsubseteq \exists P^-$, then $\mathsf{gr}(\alpha, I) = A(x)$;

2. $I = \exists S \sqsubseteq \exists P^-$, then $\mathsf{gr}(\alpha, I) = S(x, y)$, where y is new in q;

3. $I = \exists S^- \sqsubseteq \exists P^-$, then $\mathsf{gr}(\alpha, I) = S(y, x)$, where y is new in q.

if $\alpha = P(x, y)$ and

1.$I = S \sqsubseteq P$ or $I = S^- \sqsubseteq P^-$, then $\mathsf{gr}(\alpha, I) = S(x, y)$;

2.$I = S \sqsubseteq P^-$ or $I = S^- \sqsubseteq P$, then $\mathsf{gr}(\alpha, I) = S(y, x)$.

Finally, a union of conjunctive queries (UCQ) is a set of CQs.

For a query q, we will often abuse notation and use q to refer to the set $\{\alpha_1, \ldots, \alpha_m\}$ of its body atoms. Hence, for β an atom by $q \cup \{\beta\}$ we denote the new CQ of the form $\{\mathsf{avar}(q) \mid \alpha_1, \ldots, \alpha_m, \beta\}$.

Given CQs q_1, q_2 with distinguished variables $\mathbf{x}$ and $\mathbf{y}$, respectively, we say that q_2 *subsumes* q_1 (or that q_2 is a *subsumer* of q_1), if there exists a substitution σ from $\mathsf{var}(q_2)$ to $\mathsf{var}(q_1)$ such that $[\{Q(\mathbf{y})\} \cup q_2]_\sigma$ is a subset of $\{Q(\mathbf{x})\} \cup q_1$, where Q is a predicate of the same arity as $\mathbf{x}$ ($\mathbf{y}$) that does not appear in q_1 and q_2. For a UCQ u and CQ q, we say that q is *redundant* in u if another query q' in u exists that subsumes q; otherwise it is called *non-redundant* in u.

A *certain answer* to a CQ q with respect to an ontology $\mathcal{O}$ is a tuple $\mathbf{c} = (c_1, \ldots, c_n)$ of individuals such that $\mathcal{O}$ *entails* the FOL formula obtained by building the conjunction of all atoms α_i in q, replacing each distinguished variable x_j with c_j and existentially quantifying over undistinguished variables. We denote with $\mathsf{cert}(q, \mathcal{O})$ the set of all certain answers to q w.r.t. $\mathcal{O}$.

W.l.o.g. we assume that CQs are connected. More precisely, let q be a CQ. We say that q is connected if, for all terms t, t', there exists a sequence $t_1, \ldots, t_n$ such that $t_1 = t$, $t_n = t'$ and, for all $1 \leq i < n$, there exists a role R such that $R(t_i, t_{i+1}) \in q$.

2.4 Query Answering and Rewriting for DL-Lite$_R$

Query answering over DL-Lite-ontologies is performed with a technique known as *query rewriting* [8,28]. Given a TBox $\mathcal{T}$ and query q, the technique computes another query q', called a *rewriting* for $q, \mathcal{T}$, with the following property: for each ABox $\mathcal{A}$ such that $\mathcal{O} = \mathcal{T} \cup \mathcal{A}$ is consistent we have:

$$\mathsf{cert}(q, \mathcal{O}) = \mathsf{cert}(q', \mathcal{A}) \tag{1}$$

Query rewriting has been extensively used for query answering over “lightweight” ontologies [8,9,32]. A rewriting can be computed by applying certain transformations over the input TBox $\mathcal{T}$ and query q. Several techniques and algorithms have been proposed so far in the literature [8,9,11, 19,28,32] and many of them differ quite substantially from each other. For example, several techniques have proposed the use of non-recursive Datalog for representing the rewriting q' [32,13], while others return q' in its equivalent disjunctive normal form—that is, as a union of conjunctive queries u which we next call a *UCQ rewriting* for $q, \mathcal{T}$. Non-recursive Datalog provides a more compact (polynomial size) structure for computing rewritings in contrast to the worst case exponential size of UCQs [18]. However, it has been advocated [31] that UCQs provide a more suitable form when it comes to actually evaluating the computed rewriting over the stored data. In addition, recent optimisation techniques show how the structure of the data (structure of the ABox) can be used to significantly reduce the size of the UCQ [30,31].

Next, we briefly present the $\mathsf{PerfectRef}$ algorithm proposed by Calvanese et al. [8] as our algorithm uses several of its techniques. Given a CQ q and a TBox $\mathcal{T}$, $\mathsf{PerfectRef}$ computes a UCQ rewriting for $q, \mathcal{T}$, by applying exhaustively a *reformulation* and *reduction* step. Each one of them takes as input a CQ and possibly some axiom from $\mathcal{T}$ and generates a new CQ. This process terminates when no new query is generated.

In the reformulation step the algorithm picks a CQ q, an atom α in the CQ and an axiom I in $\mathcal{T}$ and checks whether I can be used to replace α in q with a new atom, hence creating a new CQ. For example, for the CQ $q_1 = \{x \mid S(x, y), S(y, z)\}$ and the axiom $I_1 = B \sqsubseteq \exists S$ reformulation would replace $S(y, z)$ producing a new query $q_2 = \{x \mid S(x, y), B(y)\}$. Next, if an axiom of the form $I_2 = \exists S^- \sqsubseteq B$ exists in $\mathcal{T}$ then this can also be used to replace $B(y)$ in q_2 producing the query $q_3 = \{x \mid S(x, y), S(w, y)\}$, where w is a new variable not appearing in q_2. More formally, for an atom α of a query and an axiom $I \in \mathcal{T}$, the function $\mathsf{gr}(\alpha, I)$ returns a new atom as defined in Table 3. If for some axiom I and some atom α one of the conditions in Table 3 holds, then we say that I is *applicable* to α, and applying I to some α in some CQ q creates a new CQ of the form $q[\alpha/\mathsf{gr}(\alpha, I)]$—that is, a new query that contains the atom $\mathsf{gr}(\alpha, I)$ instead of the atom α.

In the reduction step a new CQ is generated by applying to some CQ q the most general unifier (mgu) of two of its atoms. For example, applying reduction on query q_3 from above generates the new query $q_4 = \{x \mid S(x, y)\}$, since $S(w, y)$ unifies with $S(x, y)$.

3 Rewriting Under Atom Extensions

In this section, we study the problem of computing a UCQ rewriting for queries that have been extended with new atoms, by extending a previously computed UCQ rewriting for them. First, we provide an overview of the algorithm emphasising several of its technical points, after which we present the algorithm in detail.

3.1 An Overview

Consider the following TBox about an academic domain and the CQ which retrieves all individuals that teach someone:

$$\mathcal{T} = \{\mathsf{Professor} \sqsubseteq \exists\mathsf{teaches},\ \exists\mathsf{teaches}^- \sqsubseteq \mathsf{Student}\}$$
$$q = \{x \mid \mathsf{teaches}(x, y)\}$$

The following set is a UCQ rewriting for $q, \mathcal{T}$ computed using the $\mathsf{PerfectRef}$ algorithm:

$$u = \{q, q_1\}, \text{ where } q_1 = \{x \mid \mathsf{Professor}(x)\}.$$

More precisely, q_1 is produced by applying the first TBox axiom on atom $\mathsf{teaches}(x, y)$ of q, replacing it with $\mathsf{Professor}(x)$.

Suppose now, that the initial query is extended in order to retrieve only individuals that teach students—that is, q is extended with atom $\alpha = \mathsf{Student}(y)$ and the new query is the following:

$$q' = \{x \mid \mathsf{teaches}(x, y), \mathsf{Student}(y)\}.$$

Again, using the $\mathsf{PerfectRef}$ algorithm, we can compute the following UCQ rewriting for $q', \mathcal{T}$:

$$u' = \{q', q'_1, q, q_1\}$$

where $q', q,$ and q_1 are as defined previously, and $q'_1 = \{x \mid \mathsf{teaches}(x, y), \mathsf{teaches}(\mathsf{z}, \mathsf{y})\}$. More precisely, q'_1 is obtained from q' by applying the second TBox axiom on atom $\mathsf{Student}(y)$, replacing it with $\mathsf{teaches}(z, y)$ for z a fresh variable, while q is obtained from q'_1 by applying reduction on its two atoms. Note that variable y in query q'_1 is bound since it appears in both atoms $\mathsf{teaches}(x, y)$ and $\mathsf{teaches}(z, y)$, while after reduction it becomes unbound. Hence, finally, the algorithm can produce query q_1 from q as shown earlier.

From the above example, we can observe that when run for $q', \mathcal{T}$, the $\mathsf{PerfectRef}$ algorithm has to recompute queries q and q_1, although these have been computed previously for $q, \mathcal{T}$. To avoid repeating this work it would be beneficial to perform any rewriting work only for the newly added atom and then appropriately 'combine' the result with the previously computed rewriting (which in the following we informally call *reference* rewriting).

Consider the query $q_\alpha = \{y \mid \mathsf{Student}(y)\}$. The set $u_\alpha = \{q_\alpha, q'_\alpha\}$, where $q'_\alpha = \{y \mid \mathsf{teaches}(z, y)\}$ for z a fresh variable, is a UCQ rewriting for $q_\alpha, \mathcal{T}$ computed using the $\mathsf{PerfectRef}$ algorithm. It can be easily seen that query q' of u' can be constructed by adding the atoms of q_α to q (which has been computed previously in u), while query q'_1 can also be constructed by adding the atoms of q'_α to q. However, note that adding the atoms of q_α to $q_1 \in u$ creates the CQ $\{x \mid \mathsf{Professor}(x), \mathsf{Student}(y)\}$ which is not part of u'.

The above considerations suggest that given a rewriting u for $q, \mathcal{T}$ and an atom α, a rewriting for $q \cup \{\alpha\}, \mathcal{T}$ can be computed by computing a rewriting u_α for a 'special' query q_α and then properly extending the CQs in u with the body atoms of queries from u_α. More precisely, for α, u, q and $\mathcal{T}$ the algorithm will compute a UCQ rewriting u_α for the query $q_\alpha = \{\mathsf{var}(\alpha) \cap \mathsf{var}(q)\}\alpha$ and then add the atoms of a query $q'_\alpha \in u_\alpha$ to the atoms of a query $q' \in u$ if $\mathsf{avar}(q'_\alpha) \subseteq \mathsf{var}(q')$.

Intuitively, the above approach is possible because the $\mathsf{PerfectRef}$ algorithm is to a large extent 'local' with respect to the atoms of a query. For example, the application of the reformulation step on some query atom is independent from the rest of the query atoms. Unfortunately, the second operation of the $\mathsf{PerfectRef}$ algorithm, that of reduction, involves more than one atoms in the query and hence refutes our independence argument. A straightforward approach would be to also use reduction between queries from the two rewritings. In our running example, atom $\mathsf{teaches}(x, y)$ in query q unifies with atom $\mathsf{teaches}(z, y)$ in query q'_α. The result of unifying these two queries indeed produces the CQ q of u'. However, this approach has two issues. First, it is well-known that, in terms of performance, reduction is an inefficient step that can create a large number of (possibly redundant) queries [12,28,32]. Second, even with this operation we still cannot produce query $q_1 \in u'$.

The reduction step was initially introduced because an axiom I might only be applicable to a reduction of some CQ. In our running example, q_1 is produced from q because variable y that was bound in q'_1 became unbound in q after reduction. An advantage in our case is that we already know from the reference rewriting that q_1 can be produced from q. Hence, our algorithm only needs to check whether some CQ in u_α can be 'unified' into q in such a way that all queries that are produced due to q in the reference rewriting can still be produced without producing all possible unifications. If such a query exists, then q and all queries 'produced by' q (in our case q_1) should be part of the result. The algorithm presented in the next section identifies such cases using the function $\mathsf{mergeCQs}$ defined next.

Definition 1 Let q, q' be two queries. Then, function $\mathsf{mergeCQs}(q', q)$ returns the smallest set Σ of substitutions such that:

- If there exists atom $\alpha \in q' \cap q$, then Σ contains $\{x \mapsto x \mid x \in \mathsf{var}(\alpha)\}$;
- If there exist $R(z, y) \in q'$, $R(x, y) \in q$ or $R(y, z) \in q'$, $R(y, x) \in q$ and x, y, z are all distinct, then Σ contains $\{z \mapsto x\}$.

Note that, our restricted reduction approach is similar to the *factorisation step* proposed by Gottlob et al. [11]. However, since DL-Lite only allows for predicates with at most two variables (cf. FO translation in Table 2) the checks we need to perform (i.e., the ones of Definition 1) are much more tailor made and lightweight. Whereas, to implement the factorisation step one has to check applicability of certain TBox axioms over an atom, that is, try to actually apply a rewriting step.

As the next example shows, the substitutions returned by function mergeCQs indicate how a query can be merged into another one and this is key to our algorithm.

Example 1 Consider the following TBox and CQ:

$$\mathcal{T} = \{A \sqsubseteq \exists R\} \quad q = \{x \mid R(x, y)\}$$

with the UCQ rewriting:

$$u = \{q, q_1\}, \quad \text{where} \quad q_1 = \{x \mid A(x)\}.$$

Consider now that we extend q with atom $\alpha_1 = R(z, y)$ and that we want to compute a UCQ rewriting for $q' = q \cup \{R(z, y)\}$ and $\mathcal{T}$ using the approach described earlier. First, we construct $q_{\alpha_1} = \{y \mid R(z, y)\}$ since y is the only variable in $\mathsf{var}(a) \cap \mathsf{var}(q)$ and then the UCQ rewriting $u_{\alpha_1} = \{q_{\alpha_1}\}$ for $q_{\alpha_1}, \mathcal{T}$. Finally, a UCQ u' is computed as follows: The atoms of q_{α_1} are added to q creating the CQ $q' = q \cup \{R(z, y)\}$, while query q_{α_1} merges into q since $\mathsf{mergeCQs}(q_{\alpha_1}, q)$ contains $\{z \mapsto x\}$; hence, q and q_1 are added to u'. It can be verified that $u' = \{q', q, q_1\}$ is a UCQ rewriting for $q', \mathcal{T}$.

Suppose now that we want to further extend q' with the atom $\alpha_2 = B(z)$ and that we want to compute a UCQ rewriting for $q'' = q' \cup \{B(z)\}$ and $\mathcal{T}$. One such rewriting computed using PerfectRef is the following:

$$u'' = \{q'', q'_1, q'_2\}, \text{ where } q'_1 = \{x \mid R(x, y), B(x)\} \text{ and } q'_2 = \{x \mid A(x), B(x)\}.$$

Suppose now that we want to compute u'' using our approach. Hence, we create the CQ $q_{\alpha_2} = \{z \mid B(z)\}$, compute the UCQ rewriting $u_{\alpha_2} = \{q_{\alpha_2}\}$ for $q_{\alpha_2}, \mathcal{T}$ and then combine queries from u' and u_{α_2}. Adding the atoms of q_{α_2} to q' creates query q'', however, for all other queries $q_i \in u'$ we have $\mathsf{avar}(q_{\alpha_2}) \not\subseteq \mathsf{var}(q_i)$ and hence no other query of u'' can be constructed.

The problem in the previous example is that after we merged q_{α_1} into q our reference to variable z was lost as this was mapped to x. This information is critical when we later further extend the input query and we want to decide whether it is possible to extend queries from u' with queries from u_{α_2}. To correctly handle these cases, instead of the subset condition between variables, our algorithm uses the more involved method canBeJoined defined next.

Definition 2 Let q be a query, let σ be a substitution, and let vars be a set of variables. Then, function $\mathsf{canBeJoined}(q, \sigma, \mathsf{vars})$ returns true if for each $z \in \mathsf{vars}$ there exists $x \in \mathsf{var}(q)$ such that $z \rightsquigarrow_G x$ for G the graph induced by σ.

Example 2 Consider query q'', UCQs u_{α_2} and u', and substitution σ from Example 1. For queries $q_{\alpha_2} \in u_{\alpha_2}$ and $q \in u'$ we have that $\mathsf{canBeJoined}(q, \sigma, \mathsf{avar}(q_{\alpha_2})) = \mathsf{true}$. Hence, the atoms of q_{α_2} (i.e., $B(z)$) can be added to those of q. Note, however, that due to σ the new query that should be created is the query $q \cup \{B(z)_\sigma\}$, which is precisely q'_1 from Example 1. Similarly, for the CQ $q_1 \in u'$ we have $\mathsf{canBeJoined}(q_1, \sigma, \mathsf{avar}(q_{\alpha_2})) = \mathsf{true}$, hence the query $q_1 \cup \{B(z)_\sigma\}$ (i.e., q'_2) is also created. Consequently, u'' of Example 1 can be constructed.

Summarising the above, when there exist queries $q_i \in u_\alpha$ and $q_j \in u$ such that $\mathsf{mergeCQs}(q_i, q_j) \neq \emptyset$ (i.e., q_i can be 'merged' into q_j) first, we need to know which queries have been generated in u due to q_j, in order to 'copy' them to the result, and second, which variable mappings are used in the merge. To capture this information, in contrast to previous approaches, our algorithm operates over graphs $\mathcal{G}$ and $\mathcal{G}_\alpha$ of queries which store the dependencies between queries and the used mappings rather than on UCQs u and u_α.

Definition 3 Let q be a CQ and let $\mathcal{T}$ be a TBox. A *rewriting graph* for $q, \mathcal{T}$ is a directed labelled graph $\mathcal{G} = \langle u, \mathcal{H}, m \rangle$, where u is a UCQ rewriting for $q, \mathcal{T}$, $\mathcal{H}$ is a binary relation over u, and m maps each $q_i \in u$ to a set of variable mappings. Moreover, $\mathcal{G}$ satisfies the following properties:

- If $\langle q_1, q_2 \rangle \in \mathcal{H}$, then q_2 is produced from q_1 by the application of a reformulation or reduction step.
- For each $\langle q_1, q_2 \rangle \in \mathcal{H}$ if q_2 is produced by a reformulation step, then $m(q_2) = m(q_1)$, while if it is produced by a reduction step with σ the mgu, then $m(q_2) = m(q_1) \cup \sigma$.

Concluding our algorithm overview, an important question is whether we can compute a UCQ rewriting for an extended query given any reference UCQ rewriting. As the following example shows this is not always possible.

Example 3 Consider the following TBox and CQ:

$$\mathcal{T} = \{A \sqsubseteq \exists R\} \quad q = \{x \mid A(x), R(x, y)\}.$$

Algorithm 1 ExtendRewritingForNewAtom$(\mathcal{T}, \mathcal{G}, \alpha)$

```
input: A TBox $\mathcal{T}$, a rewriting graph $\mathcal{G} = \langle u, \mathcal{H}, m\rangle$ for some $q, \mathcal{T}$
and a new atom $\alpha$.
1: $\mathcal{G}_\alpha :=$ ex-PerfectRef$(\{\mathsf{var}(\alpha) \cap \mathsf{var}(q) \mid \alpha\}, \mathcal{T})$
2: $\mathcal{G}' :=$ joinGraphs$(\mathcal{G}, \mathcal{G}_\alpha, \mathsf{var}(\alpha) \cap \mathsf{var}(q), \mathsf{avar}(q))$
3: return $\mathcal{G}'$
```

Then, $\mathcal{G}_1 = \langle u_1, \mathcal{H}, \emptyset\rangle$ where $u_1 = \{q, q_1\}$, $q_1 = \{x \mid A(x)\}$ and $\mathcal{H} = \{\langle q, q_1\rangle\}$ is a rewriting graph for $q, \mathcal{T}$. However, q is redundant in u_1. Thus, $\mathcal{G}_2 = \langle\{q_1\}, \emptyset, \emptyset\rangle$ is also a rewriting graph for $q, \mathcal{T}$.

Now suppose that we want to extend q with the atom $\alpha = B(y)$ creating the query $q' = q \cup \{B(y)\}$. A UCQ rewriting for $q', \mathcal{T}$ computed using PerfectRef consists of the UCQ $u' = \{q'\}$.

Consider now the query $q_\alpha = \{y \mid B(y)\}$. Using PerfectRef, we compute the UCQ rewriting $u_\alpha = \{q_\alpha\}$. Clearly, it is not possible to compute u' from $\mathcal{G}_2$. However, u' can be constructed from $\mathcal{G}_1$. More precisely, adding the atoms of q_α to q creates q'.

The problem in the previous example is that although q is redundant in u_1 due to q_1, neither q_1 nor any other query related or produced by q_1 by the extension is part of the final UCQ rewriting. Hence, q and all queries that are produced by q are 'relevant' for computing a UCQ for $q \cup \{\alpha\}$ and are not redundant. Next, we formalise a property that is sufficient for computing a rewriting graph for an extended query.

Definition 4 Let q be a query, let $\mathcal{T}$ be a TBox, let $\mathcal{G} = \langle u, \mathcal{H}, m\rangle$ be a rewriting graph for $q, \mathcal{T}$. We say that $\mathcal{G}$ is *reformulation-closed* for $q, \mathcal{T}$ if the following properties are satisfied:

1. q is a top element in $\mathcal{G}$.
2. For each top element q_i in $\mathcal{G}$ we have $m(q_i) = \emptyset$.
3. If a query q_2 can be produced using a single reformulation step on some query $q_1 \in u$, then $\langle q_1, q_2\rangle \in \mathcal{H}$.
4. If $q_1 \in u$ and atoms $R(z, y)$ and $R(x, y)$ or atoms $R(y, z)$ and $R(y, x)$ appear in q_1, then $\langle q_1, q_2\rangle \in \mathcal{H}$, where $q_2 = q_{1\{z\mapsto x\}}$.

It is easy to see that the rewriting graph $\mathcal{G}_2$ from Example 3 is not reformulation-closed for $q, \mathcal{T}$ since q does not appear as a top element, while $\mathcal{G}_1$ is.

3.2 The Rewriting Extension Algorithm

Our algorithm for computing a rewriting graph for a query q extended with an atom α is shown in Algorithm 1. The algorithm accepts a TBox $\mathcal{T}$, a rewriting graph $\mathcal{G}$ for $q, \mathcal{T}$, and an atom α and it returns a rewriting graph for $q \cup \{\alpha\}, \mathcal{T}$. First, it computes a rewriting graph $\mathcal{G}_\alpha$ for the query $\{\mathsf{var}(\alpha) \cap \mathsf{var}(q) \mid \alpha\}$, which explicates from $\mathcal{T}$ the knowledge that regards atom α. This is accomplished using the sub-routine ex-PerfectRef, which consists of a straightforward extension of the standard PerfectRef algorithm [8]. More precisely, apart from applying the standard reformulation and reduction steps this algorithm also stores the dependencies between the queries using a relation as well as the mappings that are created due to the reduction steps. Subsequently, Algorithm 1 'combines' $\mathcal{G}$ and $\mathcal{G}_\alpha$ using algorithm joinGraphs and computes a rewriting graph for the extended query.

Algorithm 2 joinGraphs$(\mathcal{G}, \mathcal{G}_\alpha, jv, av)$

```
input: A rewriting graph $\mathcal{G} = \langle u, \mathcal{H}, m\rangle$ for some $q, \mathcal{T}$ and a rewrit-
ing graph $\mathcal{G}_\alpha$, the set $jv$ of the join-points, and a set of variables $av$.
1: Initialise a UCQ $u' := \emptyset$, a binary relation $\mathcal{H}' := \emptyset$,
     and a mapping $m' = \emptyset$
2: $\mathcal{G}' := \langle u', \mathcal{H}', m'\rangle$
3: Initialise a queue $Q$ with the CQ $q$
4: while $Q \neq \emptyset$ do
5:     Remove the head $q_h$ of $Q$ and let $\kappa := m(q_h)$
6:     if canBeJoined$(q_h, \kappa, jv)$ then
7:         Initialise a queue $Q_\alpha$ with a top element of $\mathcal{G}_\alpha$
8:         while $Q_\alpha \neq \emptyset$ do
9:             Remove the head $q_\alpha$ of $Q_\alpha$
10:            $nv := \mathsf{avar}(q_h) \cup (\mathsf{var}((q_\alpha)_\kappa) \cap av)$
11:            $q_c := \{nv \mid q_h \cup (q_\alpha)_\kappa\}$
12:            $m'(q_c) := \kappa$
13:            Add $q_c$ to $u'$
14:            for all $\sigma \in$ mergeCQs$((q_\alpha)_\kappa, q_h)$ do
15:                Add $\langle q_c, (q_h)_\sigma\rangle$ to $\mathcal{G}'$
16:                for all $q'$ s.t. $q_h \rightsquigarrow_{\mathcal{G}} q'$ do
17:                    if $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(q') \cup \mathsf{dom}(m(q'))$ then
18:                        $\mu' :=$ buildSubst$(\sigma, m(q'))$
19:                        $m'(\{nv \mid q'\}_{\mu'}) := \mu'$
20:                        for all $\langle q', q''\rangle \in \mathcal{G}$ do
21:                            $\mu'' :=$ buildSubst$(\sigma, m(q''))$
22:                            Add $\langle\{nv \mid q'\}_{\mu'}, \{nv \mid q''\}_{\mu''}\rangle$ to $\mathcal{G}'$
23:                        end for
24:                    end if
25:                end for
26:            end for
27:            for all $\langle q_h, q'\rangle \in \mathcal{G}$ do
28:                buildChildren$(q_c, q', q_\alpha, \mathcal{G}, \mathcal{G}', av, jv)$
29:                Add $q'$ to $Q$
30:            end for
31:            for all $\langle q_\alpha, q'\rangle \in \mathcal{G}_\alpha$ do
32:                buildChildren$(q_c, q_h, q', \mathcal{G}, \mathcal{G}', av, jv)$
33:                Add $q'$ to $Q_\alpha$
34:            end for
35:        end while
36:    end if
37: end while
38: return $\mathcal{G}'$
```

Algorithm joinGraphs is shown in Algorithm 2. Intuitively, it computes the Cartesian product of the two input rewriting graphs. The intuition is that if $\langle q, q'\rangle \in \mathcal{G}$ (i.e., q' is produced by q) and q_α is a vertex in $\mathcal{G}_\alpha$, then the same step would also be applicable to query $q \cup q_\alpha$—that is, $q \cup q_\alpha$

will produce the CQ $q' \cup q_\alpha$. Similarly, for q a vertex in $\mathcal{G}$ and $\langle q_\alpha, q'_\alpha\rangle \in \mathcal{G}_\alpha$.

More precisely, the algorithm uses queues Q and Q_α in order to traverse $\mathcal{G}$ and $\mathcal{G}_\alpha$, respectively. In line 5 it picks an element q_h from $\mathcal{G}$ and checks if the query can be extended with atoms of queries in $\mathcal{G}_\alpha$ (line 6). If it does, then a CQ q_α is also picked from $\mathcal{G}_\alpha$ (line 9) and a new query q_c from q_h and q_α is created (line 11) that has as distinguished variables the variables in nv. Moreover, $m(q_h)$ is set as the mapping for q_c (line 12). Subsequently, the algorithm checks if q_α can be merged into q_h. If it does, then it adds $\langle q_c, (q_h)_\sigma\rangle$ ($(q_h)_\sigma$ being the query produced after applying σ to q_h) to the new graph (line 15) and then also 'copies' to the result the part of $\mathcal{G}$ that has been produced due to q_h (lines 16–25) applying also a proper substitution μ. This substitution is constructed using function buildSubst.

Definition 5 Let σ and κ be two sets of mappings. Then, function buildSubst(σ, κ) returns a new set of mappings μ constructed in the following steps:

1. Set $\mu := \kappa \cup \sigma$
2. For each $z \mapsto y \in \sigma$ if $z \mapsto y'$ in κ exists then replace $z \mapsto y'$ in μ with $y \mapsto y'$.

Note that, this function is important when for q' a 'copied' query we have $m(q') \neq \emptyset$.

After checking if a query can be merged, the algorithm iterates through the queries that are produced by q_h (lines 27–30) as well as over those that are produced by q_α (lines 31–34) and constructs proper successors of q_c. This is done using the sub-routine buildChildren, depicted in Algorithm 3, which follows a similar approach as before. That is, it computes the join between q_h (a child of q_h) and some child of q_α (and q_α) and then also checks if the two queries can be merged, in which case it copies the relevant sub-graph to the result. Finally, the children of q_h (q_α) are added to Q (Q_α).

Note that $\mathcal{G}$ and $\mathcal{G}_\alpha$ can be cyclic. Hence, the algorithm needs to keep track which nodes it has visited (copied) and avoid revisiting (recopying) them. This can be done using standard graph traversal techniques.

Example 4 Consider the following rewriting graphs:

$$\mathcal{G} = \langle\{q_1, q_2, q_3\}, \mathcal{H}, m\rangle \quad \text{and} \quad \mathcal{G}_\alpha = \langle\{q^1_\alpha, q^2_\alpha\}, \mathcal{H}_\alpha, m_\alpha\rangle$$

where $\mathcal{H}$ and $\mathcal{H}_\alpha$ are as shown in Fig. 1a and b, respectively. Assume also that $m(q_i) = \emptyset$ for each $1 \leq i \leq 3$ and that all queries q_i can be extended. Figure 1c depicts the rewriting graph that is computed by Algorithm 1 for $\mathcal{G}$ and $\mathcal{G}_\alpha$, assuming that for each $q^j_\alpha \in u_\alpha$ and for each $q_i \in u$, we have mergeCQs$(q^j_\alpha, q_i) = \emptyset$—that is, no merges occur. More precisely, we have the following steps:

– At the first iteration we have $Q = \{q_1\}$ and $Q_\alpha = \{q^1_\alpha\}$, hence $q_h = q_1$ and $q_\alpha = q^1_\alpha$. In line 11, CQ $q_1 \cup q^1_\alpha$ is created (recall that $m(q_1) = \emptyset$). Then, in the for-loop in lines 27–30, queries $q_2 \cup q^1_\alpha$ and $q_3 \cup q^1_\alpha$ are created, they are set as children of $q_1 \cup q^1_\alpha$ and q_2, q_3 are added to Q. Subsequently, in the for-loop in lines 31–34 CQ $q_1 \cup q^2_\alpha$ is created, it is set as child of $q_1 \cup q^1_\alpha$ and q^2_α is added to Q_α.

Algorithm 3 buildChildren$(q_c, q, q_\alpha, \mathcal{G}, \mathcal{G}', av, jv)$

input: q_c, q, q_α are CQs, $\mathcal{G} = \langle u, \mathcal{H}, m\rangle$ and $\mathcal{G}' = \langle u', \mathcal{H}', m'\rangle$ are graphs and av, jv are sets of variables.

```
1: $\kappa := m(q)$
2: if canBeJoined$(q, \kappa, jv)$ then
3:   $nv := \mathsf{avar}(q) \cup (\mathsf{var}((q_\alpha)_\kappa) \cap av)$
4:   $q_n := \{nv \mid q \cup (q_\alpha)_\kappa\}$
5:   $m'(q_n) := \kappa$
6:   Add $\langle q_c, q_n\rangle$ to $\mathcal{G}'$
7:   for $\sigma \in$ mergeCQs$((q_\alpha)_\kappa, q)$ do
8:     Add $\langle q_n, q_\sigma\rangle$ to $\mathcal{G}'$
9:     for all $q'$ s.t. $q \rightsquigarrow_{\mathcal{G}} q'$ do
10:      if $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(q') \cup \mathsf{dom}(m(q'))$ then
11:        $\mu' :=$ buildSubst$(\sigma, m(q'))$
12:        $m'(\{nv \mid q'\}_{\mu'}) := \mu'$
13:        for all $\langle q', q''\rangle \in \mathcal{G}$ do
14:          $\mu'' :=$ buildSubst$(\sigma, m(q''))$
15:          Add $\langle\{nv \mid q'\}_{\mu'}, \{nv \mid q''\}_{\mu''}\rangle$ to $\mathcal{G}'$
16:        end for
17:      end if
18:    end for
19:  end for
20: end if
```

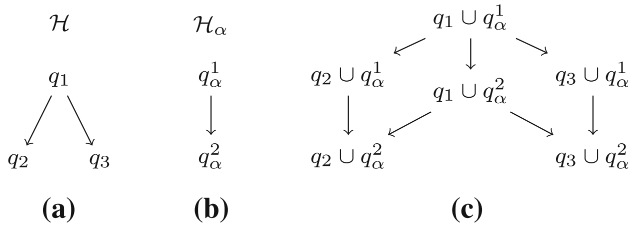

Fig. 1 Rewriting graphs of Example

– In the next iteration $Q_\alpha = \{q^2_\alpha\}$, hence q^2_α is picked and now $q_h = q_1$ and $q_\alpha = q^2_\alpha$. Then, in line 11, CQ $q_1 \cup q^2_\alpha$ is created, and then, in a similar way as described before, CQs $q_2 \cup q^2_\alpha$ and $q_3 \cup q^2_\alpha$ are created and pairs $\langle q_1 \cup q^2_\alpha, q_2 \cup q^2_\alpha\rangle$ and $\langle q_1 \cup q^2_\alpha, q_3 \cup q^2_\alpha\rangle$ are added to the result.
– In the next iteration, the algorithm picks q_2 from Q and Q_α is again initialised to $\{q^1_\alpha\}$. Then, again query $q_2 \cup q^1_\alpha$ will be constructed and 'connected' with query $q_2 \cup q^2_\alpha$ that was constructed previously.
– Finally, the algorithm picks q_3 from Q and again constructs $q_3 \cup q^1_\alpha$ and 'connects' it with $q_3 \cup q^2_\alpha$. Then, the algorithm terminates.

Concluding this section we show the correctness of Algorithm 1.

By the definition of the function mergeCQs it is clear that our algorithm does not apply the standard reduction of the PerfectRef algorithm. For example, for the query

$\{x \mid R(x, y), R(y, z)\}$ the standard reduction will produce the query $\{x \mid R(x, x)\}$; however, for $q_1 = \{x \mid R(x, y)\}$ and $q_2 = \{x \mid R(y, z)\}$, we have $\mathsf{mergeCQs}(q_2, q_1) = \emptyset$ and hence q_2 is not merged into q_1. Correctness of our merge approach follows by the following lemma which proof is given in the Appendix.

Lemma 1 *Let q_1, q_2 be two CQs such that q_2 subsumes q_1 and let q_1' be the result of applying an axiom I to q_1. If q_2 does not subsume q_1', then either I is applicable to q_2 and the result subsumes q_1' or for $1 \leq i \leq n$, atoms of the form $P(u, v)$, $P(z_i, v)$ or $P(v, u)$, $P(v, z_i)$ exist in q_2 such that for $1 \leq i, j \leq n$, $i \neq j$ we have $z_i \neq z_j$ and for $\lambda = \{z_i \mapsto u \mid 1 \leq i \leq n\}$, I is applicable to $(q_2)_\lambda$ and the result subsumes q_1'.*

As explained in Sect. 3.1, the reduction step was introduced because an axiom I might only be applicable to a reduction q' of some CQ q. However, it is well-known that if q' is the reduction of q, then q subsumes q'. Hence, if some I is applicable to q' but not to q, Lemma 1 dictates that only a restricted form of reductions over atoms in q are necessary to obtain a CQ over which I is applicable. Such reductions are precisely those used to decide whether a query from u_α can be merged into a query in the reference rewriting.

Consider a CQ q, a TBox $\mathcal{T}$, an atom α, a rewriting graph $\mathcal{G}$ for $q, \mathcal{T}$ and a rewriting graph $\mathcal{G}_\alpha$ computed by Algorithm 1 in line 1. Assume also that u is the UCQ rewriting computed using the standard $\mathsf{PerfectRef}$ algorithm over $q \cup \{\alpha\}, \mathcal{T}$. To show correctness, we will use induction over the number of steps that the $\mathsf{PerfectRef}$ algorithm has been applied over $q \cup \{\alpha\}, \mathcal{T}$. For example, assume that $u_i \subseteq u$ is the UCQ computed at step i and that for every $q_i \in u_i$ some q_h in $\mathcal{G}$ and some q_α in $\mathcal{G}_\alpha$ exist such that $\{nv \mid q_h \cup (q_\alpha)_\kappa\}$, where $\kappa = m(q_h)$ and $nv = \mathsf{avar}(q_h) \cup (\mathsf{var}((q_\alpha)_\kappa) \cap \mathsf{avar}(q))$ subsumes q_i. Then, assume that at step $i + 1$ some axiom I is applied to q_i producing q_{i+1}. By Lemma 1 either I is applicable to $\{nv \mid q_h \cup (q_\alpha)_\kappa\}$ and the result subsumes q_{i+1} or several restricted reductions are applicable. In the former case I is either applicable to some atom in q_h producing some query q_h' for which we will have $\langle q_h, q_h' \rangle \in \mathcal{G}$ or I is applicable to $(q_\alpha)_\kappa$. Hence, in the former case we will have some q_h' in $\mathcal{G}$ such that $\{nv \mid q_h' \cup (q_\alpha)_\kappa\}$ subsumes q_{i+1}. In the latter case the existence of some q_α' in $\mathcal{G}_\alpha$ such that $\langle q_\alpha, q_\alpha' \rangle \in \mathcal{G}_\alpha$ and $\{nv \mid q_h \cup (q_\alpha')_\kappa\}$ subsumes q_{i+1} follows by the following lemma which we show in the Appendix.

Lemma 2 *Let q be a CQ that contains exactly one body atom and let κ be a substitution such that some axiom I is applicable to q_κ producing q'. Then I is also applicable to q and for q'' the result we have $q''_\kappa = q'$ (modulo renaming of fresh variables).*

Finally, the following theorem establishes the correctness of the algorithm. The proof is given in the Appendix and consists of a systematic analysis of all of the aforementioned cases.

Theorem 1 *Let q be a CQ, let $\mathcal{T}$ be a TBox, let α be an atom such that $\mathsf{var}(\alpha) \cap \mathsf{var}(q) \neq \emptyset$ and let $\mathcal{G}$ be a reformulation-closed rewriting graph for $q, \mathcal{T}$. Let $\mathcal{G}'$ be the graph returned by Algorithm 1 when applied to $\mathcal{G}, \alpha$ and $\mathcal{T}$; then $\mathcal{G}'$ is a reformulation-closed rewriting graph for $q \cup \{\alpha\}, \mathcal{T}$.*

4 An Incremental Query Rewriting Algorithm

Algorithm 2 can form the basis for developing a UCQ rewriting algorithm for *fixed* queries over TBoxes. More precisely, for a fixed CQ q one can pick some atom $\alpha \in q$, compute a rewriting graph for the query $q_1 = \{\mathsf{var}(\alpha) \cap \mathsf{avar}(q) \mid \alpha\}$ over $\mathcal{T}$ using $\mathsf{ex\text{-}PerfectRef}$ and then extend this rewriting graph by iteratively adding the rest of the body atoms of q using Algorithm 2.

The above idea is illustrated in Algorithm 4. The algorithm first selects some atom α such that some of its variables appear as distinguished variables in q (line 2) and computes a rewriting graph $\mathcal{G}$ for the query $\{\mathsf{var}(\alpha) \cap \mathsf{avar}(q) \mid \alpha\}$ (line 3). At this point a rewriting graph for a query that contains only atom α of q, variables $cv = \mathsf{var}(\alpha)$ of q and distinguished variables $\mathsf{var}(\alpha) \cap \mathsf{avar}(q)$ of q have been computed. Then, the algorithm selects one-by-one the remaing atoms and extends the previously computed rewriting graph (lines 5–11). More precisely, at the i-th iteration the algorithm has computed a rewriting graph $\mathcal{G}$ for a query q_i that contains $i + 1$ atoms of q, has as distinguished variables the variables in $\mathsf{var}(q_i) \cap \mathsf{avar}(q)$, while cv contains the variables of q that appear in q_i. Hence, the algorithm picks an atom α' such that some of its variables also appear in cv (line 6), it computes a rewriting graph $\mathcal{G}_{\alpha'}$ for the query $\{\mathsf{var}(\alpha') \cap cv \mid \alpha'\}$ (line 8) and then, it joins $\mathcal{G}$ with $\mathcal{G}_{\alpha'}$ using Algorithm 2 (line 9). Finally, the algorithm uses the well-known redundancy elimination algorithm proposed in [27] to remove the redundant (subsumed) queries (line 12) and return a UCQ. Note that the latter is possible since we assume that the query is fixed.

Theorem 2 *Let q be a CQ and let $\mathcal{T}$ be a TBox. When applied to q and $\mathcal{T}$ Algorithm 4 terminates. Let u be the UCQ produced by the algorithm; then, u is a UCQ rewriting for $q, \mathcal{T}$.*

The theorem follows by Theorem 1 and induction over the atoms of q that have been added by Algorithm 4.

We now comment on the maximum number of CQs generated by our rewriting algorithm. Since our algorithm computes a UCQ rewriting it is thus of the same worst case complexity as other systems, like QuOnto, Nyaya, and Requiem, i.e., exponential w.r.t. the size of the input CQ [18].

Algorithm 4 IncrementalRew($q, \mathcal{T}$)

```
input: A CQ $q$ and a TBox $\mathcal{T}$.
1: Let $S$ be the set of body atoms in $q$
2: Remove an atom $\alpha$ from $S$ s.t. $\mathsf{var}(\alpha) \cap \mathsf{avar}(q) \neq \emptyset$
3: $\mathcal{G} := \mathsf{ex\text{-}PerfectRef}(\{\mathsf{var}(\alpha) \cap \mathsf{avar}(q) \mid \alpha\}, \mathcal{T})$
4: $cv := \mathsf{var}(\alpha)$
5: while $S \neq \emptyset$ do
6:     Remove an atom $\alpha'$ from $S$ s.t. $\mathsf{var}(\alpha') \cap cv \neq \emptyset$
7:     $jv := cv \cap \mathsf{var}(\alpha')$
8:     $\mathcal{G}_{\alpha'} := \mathsf{ex\text{-}PerfectRef}(\{jv \mid \alpha'\}, \mathcal{T})$
9:     $\mathcal{G} := \mathsf{joinGraphs}(\mathcal{G}, \mathcal{G}_{\alpha'}, jv, \mathsf{avar}(q))$
10:    $cv := cv \cup \mathsf{var}(\alpha')$
11: end while
12: return $\mathsf{removeRedundant}(\mathcal{G})$
```

Lemma 3 *Let $\mathcal{T}$ be a DL-Lite TBox, let q be a CQ, and let $\mathcal{G} = \langle u, \mathcal{H}, m\rangle$ be the output of Algorithm 4. Then, the maximal size of $|u|$ is $\mathcal{O}((|\mathcal{T}| \cdot |q|)^{|q|})$.*

Proof Let m be the number of concepts and roles appearing in $\mathcal{T}$ and n the number of atoms in q. Let also u_i be the UCQ computed at the i-1-th iteration of the while-loop of Algorithm 4. In the next iteration, u_{i+1} is computed by first computing a rewriting graph $\mathcal{G}_{\alpha_i} = \langle u_{\alpha_i}, \mathcal{H}_{\alpha_i}, m_{\alpha_i}\rangle$ for an atom α_i of q and then using the queries in u_{α_i} and u_i to build new queries by either extending or merging. Since the query for which u_{α_i} is computed contains exactly one atom, then there can be at most $m \cdot (2+1)^2$ different queries in u_α (the factor '2 + 1' is because the atom can be a role so there are two variables plus 1 for possibly freshly introduced variables). Next, the algorithm extends the queries in u_i with atoms from the queries in u_{α_i}. Hence, there can be $|u_i| \cdot m \cdot 3^2$ CQs generated by extension. Moreover, the algorithm checks if queries from u_α can be merged to CQs by computing all possible merges. For each merge, all CQs in u_i 'below' the merged query are copied. Since the queries in u_i have at most i atoms (i atoms have been processed thus far), there can be at most $i \cdot |u_i| \cdot m \cdot 3^2$ CQs computed due to merging. Summarising, $|u_{i+1}|$ contains at most $(i+1) \cdot |u_i| \cdot m \cdot 3^2$ CQs. By recursively analysing $|u_i|$ and since $|u_0| = u_{\alpha_0}$ we can conclude that $|u_n|$ contains at most $n \cdot (n-1) \cdot \ldots \cdot 2 \cdot m^n \cdot 3^{2 \cdot n}$ CQs. Since m is bounded by $|\mathcal{T}|$ and n by $|q|$ we have that $|u| = |u_n| = \mathcal{O}(|\mathcal{T}|^{|q|} \cdot |q|! \cdot 3^{2 \cdot |q|}) = \mathcal{O}(|\mathcal{T}|^{|q|} \cdot |q|!) = \mathcal{O}((|\mathcal{T}| \cdot |q|)^{|q|})$. □

Clearly, in absence of any optimisations or refinements Algorithm 4 is not likely to behave well in practice. In the following sections we present several optimisations which aim at improving its performance. Note that these optimisations intend to improve the computation time of the algorithm rather than reduce the size of the UCQ. Recall, however, that recent techniques show how we can also significantly reduce the size of the UCQ using the input data when we finally want to evaluate the rewriting [30,31].

4.1 Optimising the Last Iteration

As explained in Section 3, Algorithm 2 computes the Cartesian product between the input rewriting graphs. The structure of the computed graph is important for subsequent additions of atoms, however, it is not important after processing the *last* atom of a fixed query using Algorithm 4. Consequently, when the last atom α' is selected in line 6, Algorithm 4 can proceed as follows: First, in line 8 it can compute a UCQ rewriting $u_{\alpha'}$ for the query $\{jv \mid \alpha'\}$ using the standard PerfectRef algorithm, instead of a rewriting graph $\mathcal{G}_{\alpha'}$. Then, when joining $\mathcal{G}$ and $u_{\alpha'}$ it can call a simplified version of Algorithm 2 that constructs a UCQ rather than a rewriting graph. Algorithm 5 depicts the simplified algorithm. Roughly speaking, it is obtained from Algorithm 2 by removing the for-loops in lines 27–34 and adding the computed queries to a UCQ u' rather than a graph.

Another way to further improve the performance of Algorithm 5 is to identify cases where queries from $\mathcal{G}$ do not need to be further processed and extended with atoms of queries from u_α. The next proposition demonstrates a case where 'copying' a query from the reference rewriting to u' due to a merge is sufficient to discard certain queries from $\mathcal{G}$.

Proposition 1 *Let $\mathcal{G} = \langle u, \mathcal{H}, m\rangle$ be a rewriting graph, let u_α be a UCQ, let nv be a set of variables and let $q_h \in u$. If there exists a query q_α in u_α and substitution λ such that for all $\{z \mapsto x\} \in \mathsf{mergeCQs}((q_\alpha)_\lambda, q_h)$ we have $z \notin \mathsf{var}(q_h)$, then for each q' such that $q_h \rightsquigarrow_{\mathcal{G}} q'$, each q'_α in u_α and each substitution κ, the CQ $\{nv \mid q'\}_{\mu'}$ where $\mu' = \mathsf{buildSubst}(\{z \mapsto x\}, m(q'))$ subsumes $\{nv \mid q' \cup (q'_\alpha)_\kappa\}$.*

Consider Algorithm 5 and assume that for some query q_h in line 4, some query q_α in line 6 and some $\{z \mapsto x\} \in \mathsf{mergeCQs}((q_\alpha)_\lambda, q_h)$ in line 9 the algorithm adds to u' all CQs $\{nv \mid q'\}_{\mu'}$ such that $q_h \rightsquigarrow_{\mathcal{G}} q'$ (line 14). If $z \notin \mathsf{var}(q_h)$, then by the properties of reformulation and reduction for all such q' we have $z \notin \mathsf{var}(q')$ and hence also no mapping of the form $z \mapsto y$ appears in $m(q')$. Moreover, we clearly have that $q'_{m(q')} = q'$. Hence, for any such query q' and for $\mu' = \mathsf{buildSubst}(\{z \mapsto x\}, m(q'))$ we have $q'_{\mu'} = q'$ and thus $\{nv \mid q'\}_{\mu'}$ trivially subsumes $\{nv \mid q' \cup (q'_\alpha)_\kappa\}$ for any κ and q'_α. Consequently, Algorithm 5 adds the successors of a selected query q_h to the queue in line 19 only if $\Sigma = \emptyset$ or there exists $\{z \mapsto x\} \in \Sigma$ with $z \in \mathsf{var}(q_h)$ since otherwise the extensions of such queries is known to lead to redundant queries.

4.2 Optimising Redundancy Elimination

As described in the previous section, in line 12 Algorithm 4 applies the well-known redundancy elimination algorithm [27]. It has been shown by several experimental evaluations [9,27] that this method usually does not perform

Algorithm 5 OptimisedExtensionStep($\mathcal{G}, u_\alpha, jv, av$)

```
Input: A rewriting graph $\mathcal{G} = \langle u, \mathcal{H}, m \rangle$, a UCQ $u_\alpha$, and two sets of variables $jv$ and $av$.
1: Initialise a queue $Q$ with a top element in $\mathcal{G}$
2: Initialise a UCQ $u' := \emptyset$
3: while $Q \neq \emptyset$ do
4:     Remove the head $q_h$ of $Q$ and let $\kappa := m(q_h)$
5:     if canBeJoined($q_h, \kappa, jv$) then
6:         for all $q_\alpha \in u_\alpha$ do
7:             $nv := \mathsf{avar}(q_h) \cup (\mathsf{var}((q_\alpha)_\kappa) \cap av)$
8:             Add $\{nv \mid q_h \cup (q_\alpha)_\kappa\}$ to $u'$
9:             $\Sigma := \mathsf{mergeCQs}((q_\alpha)_\kappa, q_h)$
10:            for all $\sigma \in \Sigma$ do
11:                for all $q'$ s.t. $q_h \rightsquigarrow_{\mathcal{G}} q'$ do
12:                    if $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(q') \cup \mathsf{dom}(m(q'))$ then
13:                        $\mu' := \mathsf{buildSubst}(\sigma, m(q'))$
14:                        Add $\{nv \mid q'\}_{\mu'}$ to $u'$
15:                    end if
16:                end for
17:            end for
18:            if ($\Sigma = \emptyset$ or $\{z \mapsto x\} \in \Sigma \wedge z \in \mathsf{var}(q)$) then
19:                Add each $q'$ such that $\langle q_h, q' \rangle \in \mathcal{G}$ to $Q$
20:            end if
21:        end for
22:    end if
23: end while
24: return $u'$
```

well in practice because it consists of two nested for-loops over the (potentially large) computed UCQ u'. More precisely, the algorithm needs to check whether each CQ $q_1 \in u'$ is subsumed by some other CQ $q_2 \in u'$. In order to improve the performance of this method our algorithm uses three approaches which attempt to reduce the size of the sets over which the algorithm would execute these for-loops.

First, it tries to identify on the fly, during the execution of Algorithm 5, queries that if produced and added to u' are going to be redundant. The more redundant queries are identified the smaller the size of u' over which algorithm removeRedundant would be executed. However, u' can still be very large. Hence, second, it tries to identify queries that are going to be non-redundant in the computed set u'. Such queries can then be excluded from the final check reducing the size of the first for-loop of method removeRedundant. More precisely, if u'_{nr} is the set of non-redundant queries then only the CQs in $u' \setminus u'_{nr}$ need to be checked against the CQs in u'. Third, it tries to identify queries that are non-subsumers. This can be used to reduce the size of the second for-loop of the method. More precisely, if u'_{ns} contains all such queries then each CQs in $u' \setminus u'_{nr}$ needs to be checked only against the CQs in $u' \setminus u'_{ns}$.

4.2.1 Pruning Redundant Queries

The following proposition presents two properties by which we can identify that a query possibly generated by Algorithm 5 will be redundant in the final UCQ.

Proposition 2 *Let $\mathcal{G} = \langle u, \mathcal{H}, m \rangle$ be a rewriting graph, let u_α be a UCQ, let jv, av be sets of variables, and let u' be the output of Algorithm 5 when applied to $\mathcal{G}$, u_α, jv, and av. Let a CQ $q \in u$ with* canBeJoined$(q, m(q), jv) =$ true, *let nv be the set of variables constructed for q in line 7, and assume that q is subsumed by some other CQ $q' \in u$; then the following properties hold:*

(R1) If $\{nv \mid q'\} \in u'$, then $\{nv \mid q\}$ is redundant in u'.
(R2) If canBeJoined$(q', m(q'), jv) =$ true*, $q' \subseteq q$, and $m(q') = m(q)$, then for each $q_\alpha \in u_\alpha$ the CQ $\{nv \mid q \cup (q_\alpha)_{m(q)}\}$ is redundant in u'.*

Proof Property $(R1)$ holds straightforwardly, hence we only show Property $(R2)$.

First, we show that for all $q_\alpha \in u_\alpha$ the CQ $\{nv \mid q \cup (q_\alpha)_{m(q)}\}$ is subsumed by the CQ $\{nv' \mid q' \cup (q_\alpha)_{m(q')}\}$ where nv' is the set of variables constructed for q' in line 7. More precisely, since $q' \subseteq q$ and $m(q') = m(q)$ the for all $q_\alpha \in u_\alpha$ we have $q' \cup (q_\alpha)_{m(q')} \subseteq q \cup (q_\alpha)_{m(q)}$. Moreover, for the same reasons nv is equal to nv'; hence $\{nv' \mid q' \cup (q_\alpha)_{m(q')}\}$ subsumes $\{nv \mid q \cup (q_\alpha)_{m(q)}\}$ for each $q_\alpha \in u_\alpha$.

Now, since canBeJoined$(q', m(q'), jv) =$ true, upon termination of Algorithm 5 we have that for each $q_\alpha \in u_\alpha$ either the CQ $\{nv' \mid q' \cup (q_\alpha)_{m(q')}\}$ has been added to u' (line 8) or another query that subsumes it. In either case $\{nv \mid q \cup (q_\alpha)_{m(q)}\}$ is redundant in u'. □

Proposition 5 can be used to avoid adding redundant queries to the result u' when Algorithm 4 calls Algorithm 5 in the last iteration. However, to check for Properties $(R1)$ and $(R2)$ we need to know the subsumption relations between the CQs in $\mathcal{G}$. To identify these relations, before calling Algorithm 5, Algorithm 4 executes the standard subsumption checking algorithm over $\mathcal{G}$ and stores all such relations. Note that, no queries are removed at this point as $\mathcal{G}$ needs to be reformulation-closed for Algorithm 5 to produce a UCQ rewriting. Furthermore, note that the size of $\mathcal{G}$ at this point of iteration is expected to be significantly smaller than that of the final UCQ, hence the subsumption algorithm is expected to behave well in practice. Then, Algorithm 5 uses Properties $(R1)$ and $(R2)$ as follows:

- In line 14, it *does not* add the query $\{nv \mid q'\}_{\mu'}$ to u' if q' is subsumed by some q'' in $\mathcal{G}$ and $\{nv \mid q''\}$ has already been added to u'.
- Let q_h be a query selected in line 4. If q_h is subsumed by some q' in $\mathcal{G}$ and either $\{nv' \mid q'\}$ has already been added to u', or $q' \subseteq q_h$, $m(q') = m(q_h)$, and canBeJoined$(q', m(q'), jv) =$ true, then the algorithm 'skips' q_h—that is, it adds each q'' such that $\langle q_h, q'' \rangle \in \mathcal{G}$ to Q and it continues with the next iteration.

Note that, although the conditions of Property $(R2)$ seem rather strict, it is actually very often the case in practice that a query q' subsumes a query q under the identity substitution and that $m(q') = m(q) = \emptyset$. Moreover, regarding Property $(R1)$, note that in the vast majority of cases the set of variables nv constructed for each CQ in line 7 is the same for all queries, while in line 10 for $\sigma = \{z \mapsto x\}$ we usually have $z \notin \mathsf{var}(q')$; hence, in line 14 we will have that $\{nv \mid q'\}_{\mu'} = \{nv \mid q'\}$. Consequently, both conditions of Proposition 2 can be very effective in practice as also supported by our experimental evaluation.

Example 5 Assume that for some CQ and TBox, Algorithm 4 has at some point computed the graph $\mathcal{G} = \langle u, \mathcal{H}, m\rangle$, where u contains $q_1 = \{x \mid A(x), R(x, y)\}$ and $q_2 = \{x \mid A(x)\}$ and also $m(q_1) = m(q_2) = \emptyset$. Clearly, q_1 is subsumed by q_2, however, the algorithm cannot remove this query from $\mathcal{G}$ at this point. In the final step, assume that Algorithm 5 is called for $\mathcal{G}$ and $u_\alpha = \{q_\alpha\}$, where $q_\alpha = \{x \mid B(x)\}$. When the algorithm processes q_1 and q_α it is easy to see that the conditions of Property $(R2)$ are satisfied. More precisely, q_1 is subsumed by q_2, $q_2 \subseteq q_1$ and $m(q_1) = m(q_2)$. Hence, Algorithm 5 can avoid creating query $q = \{x \mid A(x), R(x, y), B(x)\}$ from q_1 and q_α. Indeed the algorithm will at some point pick q_2 and q_α and it will produce the query $q' = \{x \mid A(x), B(x)\}$ which subsumes q; hence, q is indeed going to be redundant if added to the result u'.

4.2.2 Tracking Non-Redundant Queries

It is quite often the case that extending a query that is non-redundant in step i produces a query that is also non-redundant in step i+1. Since usually the largest number of CQs in the final UCQ are non-redundant, it would be beneficial for the algorithm if these could be identified during the course of Algorithm 5. Such queries can then be excluded from the final reduction elimination check (line 12 of Algorithm 4), hence reducing significantly the size of the for-loops that this method needs to execute.

The following proposition provides means to identify non-redundant queries during the execution of Algorithm 5 assuming we have identified the subsumption relations in the input rewriting graph.

Proposition 3 *Let $\mathcal{G} = \langle u, \mathcal{H}, m\rangle$ be a rewriting graph, let u_α be a UCQ, let jv, av be sets of variables, and let u' be the output of Algorithm 5 when applied to $\mathcal{G}$, u_α, jv, and av. Let a CQ $q \in u$ with* $\mathsf{canBeJoined}(q, m(q), jv) = \mathsf{true}$*, let nv be the set of variables constructed for q in line 7, and assume that q is non-redundant in u; then the following properties hold:*

(N1) $\{nv \mid q\}$ is also non-redundant in u'.

(N2) Let $\kappa = m(q)$ and consider a CQ $q_\alpha \in u_\alpha$. If none of the predicates of the body atoms of q_α appear in any CQ $q' \in u$ different from q and for each $q'_\alpha \in u_\alpha$ we have $\mathsf{mergeCQs}((q'_\alpha)_\kappa, q) = \emptyset$*, then the query $\{nv \mid q \cup (q_\alpha)_\kappa\}$ is also non-redundant in u'.*

Proof (Property $(N1)$) Let q' be an arbitrary CQ in u different from q. Upon termination, either a CQ of the form $\{nv' \mid q'\}_{\mu'}$ for some μ' or a CQ of the form $\{nv' \mid q' \cup (q_\alpha)_\kappa\}$ for some $q_\alpha \in u_\alpha$ is added to u' by Algorithm 5. Assume that $\{nv' \mid q'\}_{\mu'}$ subsumes $\{nv \mid q\}$. Then a θ exists such that $q'_{\mu' \circ \theta} \subseteq q$ and so, $q'_{\theta'} \subseteq q$ for $\theta' = \mu' \circ \theta$, contrary to our assumption; similarly if we assume that $\{nv' \mid q' \cup (q_\alpha)_\kappa\}$ subsumes q. Hence, we can conclude that none of the aforementioned queries can subsume $\{nv \mid q\}$. Since q' was an arbitrary CQ it follows that no query in u' subsumes $\{nv \mid q\}$ and hence this is non-redundant in u'.

(Property $(N2)$) Since q is non-redundant in u then for all $q' \in u$ different from q and for all substitutions θ, we have $[\{Q(\mathsf{avar}(q'))\} \cup q']_\theta \not\subseteq [\{Q(\mathsf{avar}(q))\} \cup q]$. Let q' be one such arbitrary CQ and let θ be an arbitrary substitution. If $Q(\mathsf{avar}(q'))_\theta \neq Q(\mathsf{avar}(q))$, then the property trivially holds as q_α is irrelevant. Hence, assume that $Q(\mathsf{avar}(q'))_\theta = Q(\mathsf{avar}(q))$. This implies that there must be some atom At in the body of q' such that $At_\theta \in q'_\theta$ and $At_\theta \notin q$. Consider now an arbitrary query $q_\alpha \in u_\alpha$ such that none of the predicates of its body atoms appear in any body atom of q'. This implies that $At_\theta \notin q_\alpha$ and $At_\theta \notin (q_\alpha)_\kappa$ for any substitution κ and hence also $At_\theta \notin q \cup (q_\alpha)_{m(q)}$. Consequently, neither $\{nv' \mid q'\}_{\mu'}$ for any substitution μ' nor $\{nv' \mid q' \cup (q'_\alpha)_{m(q')}\}$ for any $q'_\alpha \in u_\alpha$ subsume $\{nv \mid q \cup (q_\alpha)_{m(q)}\}$. Furthermore, for each query $q'_\alpha \in u_\alpha$ we have $\mathsf{mergeCQs}(q'_\alpha, q) = \emptyset$ and again because no predicate name of q'_α appears in any other CQ, no query of the form $\{nv \mid q\}_{\mu'}$ for μ' a substitution is ever added to u' by Algorithm 5. Summarising, $\{nv \mid q \cup (q_\alpha)_{m(q)}\}$ is non-redundant in u'. □

Let $\mathcal{G} = \langle u, \mathcal{H}, m\rangle$ be the graph and u_α the UCQ with which Algorithm 5 is called. Then, in order to use Properties $(N1)$ and $(N2)$ the algorithm is modified as follows:

- At the beginning it initialises an empty set *NR* of non-redundant queries.
- In line 14, if $\{nv' \mid q'\}_{\mu'} = \{nv \mid q'\}$ and q' is non-redundant in u, then it adds $\{nv \mid q'\}_{\mu'}$ to *NR*.
- In line 8, it adds $\{nv \mid q_h \cup (q_\alpha)_{m(q)}\}$ to *NR* if q_h is non-redundant in u, none of the predicates in q_α appear in any query in u and if for each $q'_\alpha \in u_\alpha$ we have $\mathsf{mergeCQs}(q'_\alpha, q) = \emptyset$.
- Finally, it returns both the UCQ u' and the set *NR*.

Subsequently, the returned set *NR* is used by method $\mathsf{removeRedundant}$ (line 12 of Algorithm 4) to exclude all these queries from redundancy checking.

Again, note that checking whether for each q_α no predicate appears in a body atom of a query in u requires iterating for each $q_\alpha \in u_\alpha$ through all queries in u. However, this can be performed only once at the beginning of Algorithm 5 and moreover, as mentioned before, at this point of the algorithm the set u is expected to be moderate in size. Furthermore, as we will show in the evaluation section, in several cases this technique helps us to significantly decrease the cost of checking redundancy of the set u' and in many cases even avoid it completely if $u' = NR$. This significantly outperforms its implementation overhead.

Example 6 Consider again $\mathcal{G}, u$, and u_α from Example 5. If u contains no other queries, then it is easy to see that for $q_2 \in u$ and $q_\alpha \in u_\alpha$ the conditions of Property $(N2)$ are satisfied. More precisely, q_2 is non-redundant in u, none of the predicates of q_α appear in any other CQ in u and $\mathsf{mergeCQs}(q_\alpha, q_2) = \emptyset$. Hence, the CQ produced by q_2 and q_α—that is, q' from Example 5, is non-redundant in the result. However, assume that besides q_1 and q_2 the UCQ u also contained the CQ $q_3 = \{x \mid S(x, z)\}$ and u_α also contained the CQ $q'_\alpha = \{x \mid S(x, z)\}$. Now, Property $(N2)$ is not satisfied for q_2 and q'_α, since the predicate of q'_α appears in q_3. Hence, we cannot guarantee that q_2 and q'_α will produce a non-redundant query. Actually, the query that they produce (i.e., $\{x \mid A(x), S(x, z)\}$) is going to be subsumed by the query produced from q_3 and q'_α (i.e., $\{x \mid S(x, z)\}$). However, the query q' produced by q_2 and q_α is still non-redundant.

4.2.3 Tracking Non-Subsumers

Finally, the following proposition presents a property by which we can identify that a query generated during the execution of Algorithm 5 will not subsume any other CQ generated by the same algorithm.

Proposition 4 *Let $\mathcal{G} = \langle u, \mathcal{H}, m\rangle$ be a rewriting graph, let u_α be a UCQ, let jv, av be sets of variables, and let u' be the output of Algorithm 5 when applied to $\mathcal{G}, u_\alpha, jv$, and av. Let a CQ $q \in u$ with $\mathsf{canBeJoined}(q, m(q), jv) = \mathsf{true}$, let nv be the set of variables constructed for q in line 7, and assume that q does not subsume any CQ in u; then the following property holds:*

(S) Let a CQ $q_\alpha \in u_\alpha$. If none of the predicates of the body atoms of q_α appear in any CQ in u (including q), then the query $\{nv \mid q \cup (q_\alpha)_{m(q)}\}$ does not subsume any CQ in u'.

Proof Since q does not subsume any CQ in u, we have $q_\theta \not\subseteq q'$ for all $q' \in u$ different from q and substitutions θ. Hence, for each θ there exists an atom At in the body of q such that $At_\theta \in q_\theta$ and $At_\theta \notin q'$. To show that $\{nv \mid q \cup (q_\alpha)_{m(q)}\}$, called q_n in the following, is not a subsumer of any CQ in u' consider an arbitrary CQ $q' \in u$. Upon termination, Algorithm 5 can add to u' either the CQ $\{nv' \mid q'\}_{\mu'}$ for some μ' or the CQ $\{nv' \mid q' \cup (q'_\alpha)_\kappa\}$ for some (possibly different) $q'_\alpha \in u_\alpha$. We will show that q_n does not subsume any such CQ.

By assumption none of the predicates of the atoms in q_α appear in q'. Hence, there clearly exists some $At' \in q_n$ (actually some that appears in q_α) such that for any θ we have $At'_\theta \notin \{nv' \mid q'_{\mu'}\}$; hence, q_n does not subsume $\{nv' \mid q'\}_{\mu'}$. Moreover, by these arguments it also follows that q_n does not subsume $\{nv' \mid q' \cup (q'_\alpha)_\kappa\}$ for any κ and any $q'_\alpha \in u_\alpha$ different from q_α.

Finally, consider the CQ $\{nv' \mid q' \cup (q_\alpha)_\kappa\}$. Recall that for every θ there exists $At_\theta \in q_\theta$ such that $At_\theta \notin q'$. Hence, again by assumption since none of the atoms in q_α appear in q we have $At_\theta \notin q_\alpha$ and hence also $At_\theta \notin \{nv' \mid q' \cup (q_\alpha)_\kappa\}$.

Since q' was an arbitrary query, it follows that q_n cannot subsume any CQ produced by Algorithm 5. □

Similarly as before, Algorithm 5 uses additional sets to store such queries which are then excluded from the for-loops of algorithm $\mathsf{removeRedundant}$.

Example 7 Consider again Examples 5 and 6, and the UCQ $u = \{q_1, q_2, q_3\}$. Then, it is easy to see that for $q_1 \in u$ and $q_\alpha \in u_\alpha$ the conditions of Property (S) are satisfied. More precisely, q_1 does not subsume any CQ in u and none of the predicates of the body of atoms of q_α appear in any CQ in u. Hence, the CQ produced by q_1 and q_α—that is $q' = \{x \mid A(x), R(x, y), B(x))\}$ is not going to subsume any CQ in the result. However, assume that u_α also contained the CQ $q'_\alpha = \{x \mid R(x, z)\}$. Now, Property (S) is not satisfied for q_1 and q'_α, since the predicate of q'_α appears in q_1. Hence, we cannot guarantee that q_1 and q'_α will produce a query that is not going to subsume any other CQ in the result. Actually the query that they produce (i.e., $\{x \mid A(x), R(x, y), R(x, z))\}$) subsumes the query produced by q_2 and q'_α (i.e., $\{x \mid A(x), R(x, z)\}$).

5 Implementation and Evaluation

We have implemented Algorithms 1–5 in a prototype tool called IQAROS.[2] We have also implemented an extended $\mathsf{PerfectRef}$ algorithm, called $\mathsf{PerfectRef}^+$, that uses a restricted reduction step based on Lemma 1.

We have conducted three experimental evaluations. The first one compares $\mathsf{PerfectRef}$ with $\mathsf{PerfectRef}^+$ to assess how much the restricted reduction step improves the performance of the original $\mathsf{PerfectRef}$ algorithm. In the second one we compared several versions of the IQAROS system using each time a different set of the optimisations presented in Section 4. In addition, we compared against $\mathsf{PerfectRef}^+$ to assess how much the incremental step-by-step algorithm

[2] http://code.google.com/p/iqaros/

improves the original strategy. Finally, in the third experiment we compared IQAROS against several state-of-the-art query rewriting systems. More precisely, we were able to compare with Rapid [9], Nyaya [11] and Presto [32], while we did not compare against Requiem [27] because Rapid significantly outperforms it [9].

For the evaluation we used the framework proposed in [27]. It consists of nine ontologies, namely V that captures information about European history, P1 and P5 that are two hand-crafted artificial ontologies, S that models information about European Union financial institutions, U that is a DL-Lite$_R$ version of the well-known LUBM[3] ontology and A that is an ontology capturing information about abilities and disabilities. Moreover, we also used the ontologies P5X, UX and AX that consist of normalised versions[4] of the ontologies P5, U and A. For each ontology, a set of five hand-crafted queries is proposed. All experiments were conducted on a MacBook Pro with a 2.66GHz processor and 4GB of RAM, with a time-out of 600 s.

5.1 Comparing PerfectRef and PerfectRef$^+$

Table 4 presents the results of running PerfectRef (denoted as PR in the table) and PerfectRef$^+$ (denoted as PR$^+$ in the table) over the test queries and ontologies. The table presents the size of the computed UCQs before applying the final redundancy elimination algorithm, then the corresponding computation times (in milliseconds), and finally the time to execute the redundancy elimination algorithm in the computed UCQ. Furthermore, the column marked as $\sharp NR$ presents the size of the non-redundant UCQ that systems should have computed. Note that this is the size of the UCQ computed by both system after the final redundancy elimination. Also, note that we do not present results for ontology P1 as it is trivial for the tested systems.

From the table we can observe that in many cases, most notably in ontologies P5, P5X, and S the size of the UCQ computed by PerfectRef$^+$ is much smaller than the one computed by PerfectRef. Especially, in P5 and P5X this is because the test queries consist of a role chain of the form $\{x \mid R(x_1, x_2), \ldots, R(x_{i-1}, x_i)\}$, hence the (standard) reduction of PerfectRef produces many redundant queries like those presented before Lemma 1. However, in all other cases the differences between the systems are marginal.

Regarding performance, we note that PerfectRef$^+$ demonstrates better performance than PerfectRef in all tests where the former managed to compute fewer redundant queries than the latter. However, in all other cases both systems behave the same and sometimes PerfectRef$^+$ is actually slower than PerfectRef. This is due to the overhead of implementing the checks for the restricted reduction step. No significant differences were observed in the execution of the final redundancy elimination algorithm, not even in the ontologies where the UCQ computed by PerfectRef$^+$ is notably smaller that that computed by PerfectRef.

5.2 Comparing IQAROS and PerfectRef$^+$

Table 4 also shows the results of running three different versions of IQAROS; the first one (called Inc$_1$) implements Algorithms 4 and 2 without any optimisations, the second one (called Inc$_2$) uses Algorithm 5 instead of Algorithm 2 when it adds the last atom of the query, while the third one (called Inc$_3$) refines Inc$_2$ by also implementing the various optimisations detailed in the previous section for improving the efficiency of the redundancy elimination algorithm.

First, we can observe that the sizes of the UCQs computed by Inc$_1$ and PerfectRef$^+$ are almost identical (with some minor differences in some queries and ontologies). This is because both systems are based on the restricted reduction step and because in its core Inc$_1$ is based on the same reformulation algorithm for explicating knowledge from $\mathcal{T}$. However, despite their similarities in the computed UCQs we can observe that Inc$_1$ is significantly more efficient than PerfectRef$^+$. For example, in ontologies P5, P5X, S, U, and UX, Inc$_1$ is several times faster than PerfectRef$^+$, while in query 5 in ontologies A and AX, it manages to be up to two orders of a magnitude faster than PerfectRef$^+$. Since in their core both systems are based on the same approach for materialising knowledge from $\mathcal{T}$ and since the restricted reduction step implemented in PerfectRef$^+$ did not improve much the performance of the original PerfectRef system, we concluded that this improvement is mainly due to the incremental rewriting strategy. More precisely, the incremental approach provides a much more guided and localised strategy (it processes a single atom at a time), compared to the blind brute-force application of the inference rules of PerfectRef($^+$). Finally, since both systems compute UCQs of comparable size the time for eliminating the redundant queries using algorithm removeRedundant are quite similar. Note, however, that this algorithm heavily depends on the form of input (e.g., the order that it processes the queries) and hence small variations may occur.

Comparing the different versions of IQAROS we can observe that the size of the computed UCQs decreases as we move from Inc$_1$ to Inc$_2$ and finally to Inc$_3$. For example, in several cases Inc$_2$ computes even up to 30 times smaller UCQs than Inc$_1$ (see ontologies P5X, S, U, and UX). This decrease can be justified by the fact that Inc$_2$ uses (the simpler) Algorithm 5 instead of Algorithm 2, when

[3] http://swat.cse.lehigh.edu/projects/lubm/

[4] In the normalised ontologies each axiom of the form $A \sqsubseteq \exists R.B$ in the original ontology is rewritten into the axioms $A \sqsubseteq \exists R_n$, $R_n \sqsubseteq R$, and $\exists R_n^- \sqsubseteq B$ for R_n a new role.

Table 4 Comparison between PerfectRef, PerfectRef$^+$ and various versions of IQAROS

$\mathcal{O}$	Q	Size of Computed UCQ					$\sharp NR$	UCQ Computation Time					Redundancy Elimination Time				
		PR	PR+	Inc$_1$	Inc$_2$	Inc$_3$		PR	PR+	Inc$_1$	Inc$_2$	Inc$_3$	PR	PR+	Inc$_1$	Inc$_2$	Inc$_3$
V	1	15	15	15	15	15	15	5	4	7	6	6	1	1	1	1	1
	2	11	10	10	10	10	10	8	5	8	6	8	3	3	2	2	1
	3	72	72	72	72	72	72	56	46	24	12	14	85	93	41	37	0
	4	185	185	185	185	185	185	101	81	41	22	31	53	38	45	34	0
	5	150	150	30	30	30	30	111	126	11	10	17	125	115	3		13
P5	1	6	6	6	6	6	6	1	1	1	1	1	1	0	0	1	0
	2	11	10	10	10	10	10	15	11	7	3	5	2	2	1	1	1
	3	22	13	13	13	13	13	256	152	76	19	19	3	3	2	2	1
	4	45	15	15	15	15	15	1,828	875	288	173	123	1	2	3	5	2
	5	90	16	16	16	16	16	32,255	5,706	838	306	362	1	0	3	4	2
P5X	1	14	14	14	14	14	14	0	0	0	0	0	0	0	0	1	1
	2	86	81	81	25	25	25	2	3	2	3	1	2	2	4	1	1
	3	530	413	413	133	103	58	36	24	24	6	17	19	16	71	13	14
	4	3,476	2,070	2,070	670	369	179	656	325	187	46	123	73	76	126	237	382
	5	23,744	10,352	10,352	3,352	1,885	718	41,454	6,762	828	214	418	1,327	1,340	1,989	864	879
S	1	6	6	6	6	6	6	0	0	0	0	1	0	0	0	0	0
	2	202	202	204	12	2	2	12	15	12	4	5	1	1	0	0	0
	3	1,005	995	864	96	4	4	190	180	60	8	16	4	5	5	1	0
	4	1,548	1,548	1,428	84	4	4	254	247	104	9	14	5	5	12	1	0
	5	8,693	7,855	6,048	672	8	8	8,216	5,888	1,018	227	160	90	85	128	9	0
U	1	2	2	2	2	2	2	0	1	1	1	2	0	0	0	0	0
	2	189	189	190	5	1	1	24	17	12	3	2	1	1	0	0	0
	3	296	296	300	20	4	4	112	112	77	5	8	2	1	2	0	0
	4	1,763	1,746	1,688	45	2	2	826	808	253	8	22	7	5	5	0	0
	5	3,418	3,410	3,375	90	10	10	2,680	2,624	527	17	41	17	16	55	1	0
UX	1	5	5	5	5	5	5	0	1	1	1	6	0	0	0	1	0
	2	286	286	287	7	1	1	14	7	10	4	7	0	1	1	0	0
	3	1,248	1,248	1,260	84	12	12	118	121	80	10	34	10	10	24	11	0
	4	5,385	5,325	5,137	129	5	5	829	818	201	11	24	20	19	42	4	0
	5	9,220	9,200	8,955	225	25	25	2,625	2,731	427	31	61	95	97	166	36	0
A	1	402	402	357	77	77	27	24	28	17	5	16	1	3	1	6	1
	2	103	103	103	54	54	50	124	124	39	12	10	0	1	0	31	0
	3	104	104	104	104	104	104	656	677	173	103	65	5	6	4	225	0
	4	492	492	471	320	320	224	1,237	1,264	170	58	57	17	16	27	72	1
	5	624	624	624	624	624	624	355,571	324,006	3,412	258	812	166	145	255	233	5
AX	1	783	783	794	431	431	41	30	27	18	4	7	4	5	6	4	3
	2	1,812	1,812	1,812	1,653	1,545	1,431	141	141	57	26	34	620	546	695	746	6
	3	4,763	4,763	4,763	4,466	4,466	4,466	707	701	186	48	144	7,985	7,141	9,832	7,958	47
	4	7,251	7,251	7,229	6,639	4,479	3,159	1,282	1,384	192	37	88	3,527	3,322	4,699	3,542	49
	5	78,885	78,885	78,885	74,025	32,944	32,921	319,681	337,649	4,361	665	1,559	–	–	–	–	840

it adds the last atom of the query. Moreover, we can see that the use of a lightweight algorithm for the last iteration of Algorithm 4 is also reflected in the computation times of Inc$_2$ compared to Inc$_1$. More precisely, in nearly all ontologies Inc$_2$ is several times faster than Inc$_1$. The effects of computing much smaller UCQs for ontologies P5X and AX are also reflected in the final redundancy elimination algorithm. However, note that neither system can compute a non-redundant UCQ for query 5 in ontology AX.

Finally, Inc_3 produces the smallest UCQ rewriting compared to all previous systems. This is due to the additional implemented techniques for identifying redundant queries that have been described in Proposition 2. We can also observe that due to these optimisations, in most cases the UCQ computed by Inc_3 is actually also non-redundant or it contains very few redundant queries. Regarding computation time, Inc_3 is generally as fast as Inc_2, however, we can observe that in several cases it is slightly slower. This is due to the overhead of implementing the various optimisations. However, when considering the time for the redundancy elimination algorithm the benefits of tracking (non-)redundant queries become most apparent. Inc_3 is faster than all other systems and much faster in all queries of ontology AX. Especially, Inc_3 is the only system that can compute a non-redundant UCQ for query 5 in ontologies A and AX. More precisely, from the 32,960 queries that the algorithm has computed after processing the last atom of query 5 it has identified 31,593 non-redundant queries. These queries are then skipped from the redundancy elimination step and hence the algorithm finishes in less than 2.4 s.

5.3 Comparing IQAROS, Nyaya, Presto and Rapid

Table 5 presents a comparison between Inc_3 and the systems Rapid, Nyaya and Presto (again we do not present the results for P1). Moreover, in the columns marked as 'UCQ Computation Time' we also present two additional times. The first one, marked as Inc_1^n, is the time required by Inc_1 to process the last atom α of the input query q using Algorithm 4, while Inc_3^n is the same time but for the configuration Inc_3 using Algorithm 5. Hence, these times reflect only the time that is required to extend the rewriting graph $\mathcal{G}^-$ computed so far for $q^-, \mathcal{T}$, where $q^- = q \setminus \{\alpha\}$, into a rewriting for $q, \mathcal{T}$—that is, if we were given $\mathcal{G}^-$ for $q^-, \mathcal{T}$ these times would reflect only the time to extend the input rewriting into a new UCQ rewriting for q^- extended with α. Note here that the output of Inc_1^n is a rewriting graph which can then be further extended, while the output of Inc_3^n is a UCQ.

As before, after the final redundancy elimination all systems return UCQ rewritings of the same size (those reported in Table 4 column $\sharp NR$), except for **Presto** in queries 2–5 in ontology P5 and queries 2 and 4 in ontology AX. After manually inspecting the ontologies and computed UCQs and contrasted with the ones computed by all other systems we concluded that Presto is incomplete in these cases. More precisely, it fails to compute queries which are *not* subsumed by other queries that it computes. Hence, there exist an ABox for which equation (1) fails; similarly, in ontology AX.

Compared to Nyaya, Inc_3 (as well as all other versions of IQAROS from Table 4) is in general much faster, in some cases even for several orders of magnitude. Furthermore, Inc_3 also computes much smaller UCQs. Since **Nyaya** is also mainly based on the same reformulation algorithm as PerfectRef for materialising knowledge from $\mathcal{T}$ the reasons for this difference are again similar to the ones mentioned before.

Compared to Presto, Inc_3 computes smaller UCQs with most distinct cases queries 2–5 in P5X, queries 1 and 2 in A and finally all the queries in AX. The effects of computing much smaller UCQs are also reflected in the UCQ computation time where Inc_3 performs in general much faster, even for several orders of magnitude in some cases (query 1 in V, queries 4 and 5 in P5X, query 1 in A, and all queries in AX) with most notable one query 5 in AX where Presto fails to terminate in the provided timeout. However there exist cases that Presto is 'notably faster'[5] such as query 5 in UX and query 5 in A. In addition we can see that the redundancy elimination algorithm of Inc_3 is much more efficient than that of Presto due to the several optimisations techniques that are used to identify (non-)redundant and non-subsumer queries.

Compared to Rapid, Inc_3 computes similarly small UCQs with some small exceptions (either against or in favour) in queries 3–5 in ontology P5X, in query 1 in ontologies A and AX and in queries 2, 4 and 5 in ontology AX. Moreover, Rapid is notably faster only in queries 4 and 5 in P5 and 5 in S and A. However, even in these cases the difference between the systems is rather marginal as it never exceeds 253 ms. In all the other scenarios Inc_3 is faster with most notable cases queries 4 and 5 in P5X and 2–5 in AX. Moreover, we can also see that the redundancy elimination algorithm of Inc_3 is much more efficient than that of Rapid with again notable case query 5 in ontology AX. Once more, this is justified by the optimisation techniques that are used in Inc_3 in order to identify (non-)redundant and non-subsumer queries.

However, we can see that the most efficient approach is Inc_3^n. Hence, indeed computing a UCQ rewriting by extending a previously computed rewriting graph is the fastest way to compute a UCQ rewriting for an extended query. However, as noted before, the output of this algorithm is not a rewriting graph and hence cannot be used for further extensions of the query. However, by observing the computation time of Inc_1^n we can see that a rewriting graph can also be computed relatively efficiently. Hence, we argue that when a query q is extended with a new atom α Algorithm 5 can be used to efficiently compute a new UCQ rewriting for $q' = q \cup \{\alpha\}$, while at the same time, as a background process, Algorithm 4 (discarding redundancy elimination) can be used to compute a new rewriting graph for q', which can then be used in a similar way to compute a UCQ rewriting and a rewriting graph for extensions of q'.

[5] We consider a system X to be 'notably faster' than a system Y if $t_Y - t_X > 20$ ms.

Table 5 Comparison between Rapid, Nyaya, Presto and IQAROS

$\mathcal{O}$	Q	Size of Computed UCQ				UCQ Computation Time						Redundancy Elimination Time			
		Rapid	Nyaya	Presto	Inc_3	Rapid	Nyaya	Presto	Inc_3	Inc_1^n	Inc_3^n	Rapid	Nyaya	Presto	Inc_3
V	1	15	15	15	15	13	84	793	6	6	6	0	0	1	1
	2	10	10	10	10	13	121	4	8	3	4	0	0	1	1
	3	72	72	72	72	78	360	42	14	15	9	0	0	21	0
	4	185	185	185	185	102	442	52	31	30	25	0	0	36	0
	5	30	52	30	30	98	476	12	17	5	8	0	24	2	13
P5	1	6	6	6	6	7	14	5	1	1	1	0	0	0	0
	2	10	10	6	10	14	128	4	5	6	3	1	0	0	1
	3	13	13	6	13	22	726	16	19	62	6	1	0	1	1
	4	15	15	6	15	33	1,889	2	123	212	19	3	0	7	2
	5	16	16	6	16	75	16,062	3	362	660	30	2	0	4	2
P5X	1	14	14	14	14	10	12	20	0	0	0	0	0	1	1
	2	25	66	81	25	23	130	7	1	1	1	3	40	13	1
	3	127	374	413	103	92	540	76	17	15	13	43	875	1,095	14
	4	636	2,475	2,070	369	343	1,672	1,371	123	120	106	838	2,170	1,891	382
	5	3,180	17,584	10,352	1,885	2,061	15,095	31,797	418	581	276	3,191	127,485	73,762	879
S	1	6	6	6	6	6	15	57	1	0	1	0	0	0	0
	2	2	3	2	2	9	11	9	5	9	2	0	1	0	0
	3	4	7	4	4	14	46	23	16	45	4	0	2	0	0
	4	4	5	4	4	14	34	19	14	86	6	0	0	0	0
	5	8	13	8	8	36	159	23	160	808	51	1	4	0	0
U	1	2	2	2	2	9	25	63	2	0	1	0	0	0	0
	2	1	1	1	1	19	7	15	2	9	1	0	0	0	0
	3	4	4	4	4	13	172	14	8	68	4	1	0	0	0
	4	2	2	2	2	17	15	4	22	238	11	0	0	0	0
	5	10	11	10	10	18	107	5	41	484	18	1	1	0	0
UX	1	5	5	5	5	11	24	74	6	0	3	0	0	0	0
	2	1	1	1	1	13	6	13	7	6	3	0	0	0	0
	3	12	12	12	12	20	166	7	34	64	24	1	0	0	0
	4	5	5	5	5	17	15	23	24	160	13	0	0	0	0
	5	25	26	25	25	26	115	6	61	358	36	4	5	0	0
A	1	27	248	402	77	18	1,231	778	16	15	14	0	73	41	1
	2	54	93	103	54	43	4,928	12	10	37	6	2	39	2	0
	3	104	105	104	104	97	35,451	15	65	145	31	0	40	5	0
	4	333	455	492	320	170	17,121	47	57	153	49	38	390	65	1
	5	624	–	624	624	383	–	160	812	3,243	624	1	–	197	5
AX	1	41	556	782	431	26	1,282	2,245	7	14	6	0	367	128	3
	2	1,546	1,738	1,781	1,545	649	4,493	615	34	41	32	542	1,095	1,034	6
	3	4,466	4,742	4,752	4,466	1,694	34,032	8,000	144	129	81	531	17,320	17,260	47
	4	4,497	6,565	7,100	4,479	1,247	16,569	8,236	88	152	82	1,538	19,891	22,543	49
	5	32,956	–	–	32,944	3,810	–	–	1,559	3,628	1,430	56,196	–	–	840

Summarising, Fig. 2 presents (using a logarithmic scale) the average computation time (both rewriting and final redundancy elimination) for each ontology, query and system presented in Table 5. The results depicted in the figure verify our previous analysis—that is, in the three non-trivial ontologies (i.e., V, P5X, and AX5) we can observe that Inc_3^n is the most efficient system followed by Inc_3, Rapid and then Presto. However, in most of the other ontologies all systems behave quite close to each other and no significant differences can be noted.

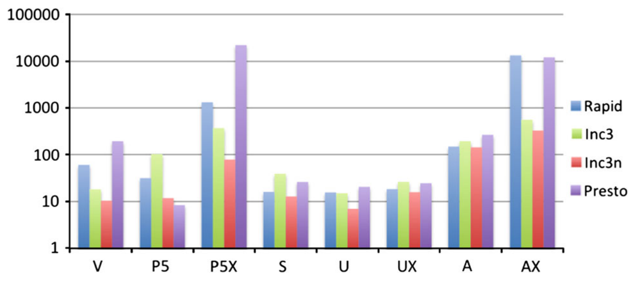

Fig. 2 Average rewriting time for all queries for each ontology

Finally, we have conducted a brief analysis of the memory required by each system in order to compute the final UCQ. The maximum amount of memory required by Rapid is 140MB, followed by Nyaya with 150MB, then Presto with 180MB, and finally Inc_3 with 200MB. These maximum numbers were observed in ontology AX query 5.

6 Related Work

To the best of our knowledge there is no previous work in computing a rewriting of an extended query q based on a previously computed rewriting for q in the presence of logical constraints in either the ontology or database literature. The only relevant problem studied in the database literature is *view adaptation* [14,22], where the problem is to compute the materialisation of a *re-defined* materialised view. However, in both works the focus is on updating the data (the materialisation of the view) and additionally there are no database constraints (ontological axioms) involved.

However, much work has been spent the last couple of years towards studying the problem of query rewriting over lightweight ontology languages both from a complexity point of view [7,13,18] as well as from the point of view of developing practical query rewriting systems [8,9,11,24,27,32]. Next, we briefly overview the works that are most relevant to ours; a more extensive and detailed literature survey on query rewriting can also be found at [12].

The first algorithm for query rewriting over DL-Lite ontologies was introduced by Calvanese et al. [8] and was later implemented in the QuOnto system [1]. The algorithm rewrites an input query and TBox into a union of conjunctive queries using the reformulation and reduction steps. Then, Pérez-Urbina et al. [28] presented a resolution-based query rewriting algorithm for DL-Lite. The algorithm was implemented in the system Requiem [27] and the experimental evaluation showed that it outperforms QuOnto. Requiem was the first system to use query subsumption in order to reduce the number of computed redundant queries. Subsequently, Rosati and Almatelli [32] presented Presto that computes a rewriting in the form of a non-recursive Datalog program instead of a UCQ. Hence, the computed rewriting is much smaller compared to the output of Requiem and QuOnto. Also Presto was the first system to provide a technique that avoids the exponential blow-up of the reduction step of $\mathsf{PerfectRef}$. Recently, Chortaras et al. [9] presented a highly optimised resolution-based algorithm for DL-Lite that was implemented in the system Rapid. It was shown that Rapid outperforms both QuOnto and Requiem. Finally, a new system called Quest [30] uses similar rewriting techniques to $\mathsf{PerfectRef}$ but with many additional optimisations that use the structure of the input data to reduce the size of the computed rewriting and speed up the process.

Moreover, query rewriting has also attracted the attention in the field of query answering over database constraints. Calì et al. [3,5] have studied and presented several lightweight classes of *tuple-generating dependencies* as well as of the *Entity-Relationship* model [4,6]. Moreover, in subsequent works Gottlob et al. [11] also presented a practical query rewriting algorithm for *Linear-Datalog*$^{\pm}$ which is the one implemented in the Nyaya system. This algorithm is also based on the original DL-Lite algorithm but improves it with many optimisations, like *atom factorization* which is intended to reduce the number of redundant queries produced in the reduction step. Subsequently, a new algorithm that computes a non-recursive Datalog program for *Linear-Datalog*$^{\pm}$ was also presented [24].

Finally, a slightly different approach than the previous ones, called *combined rewriting*, has been proposed by Lutz et al. [21] for the DL language $\mathcal{EL}$ and by Kontchakov et al. [19] for the DL language DL-Lite$^{\mathcal{N}}_{horn}$. This approach computes small rewritings, however, it also requires preprocessing of the database.

7 Conclusions

In the current paper, we studied the problem of computing a UCQ rewriting for queries that have been extended with new atoms, by extending a previously computed UCQ rewriting for them and avoiding the computation of a UCQ rewriting from scratch. We studied the problem theoretically and presented detailed algorithms. Our study also gave rise to a novel query rewriting algorithm that is based on the incremental processing of query atoms. More precisely, given a fixed input query one can process one atom at a time and extend a previously computed rewriting until a UCQ for the input query has been computed. To improve its efficiency we have proposed several novel optimisations which greatly improve the computation time and reduce the number of computed redundant queries. Finally, we have implemented all algorithms and have conducted a detailed experimental evaluation. Our results show that the algorithm is highly efficient and generally outperforms all state-of-the-art systems that are currently available.

Apart from providing a novel efficient query rewriting system, our results have several important theoretical consequences and give many opportunities for future work. First, they show that rewriting over DL-Lite ontologies can largely be performed in parallel giving a complete efficient algorithm which, to the best of our knowledge, was previously unknown. This direction has not been explored in this paper and can be part of future work. Other directions of future work can be the investigation and design of such incremental algorithms for other lightweight languages, like Linear-Datalog$^{\pm}$, or for more expressive languages, like $\mathcal{ELHI}$. Furthermore, in the current paper, we have not studied other types of refinements for input queries, like the removal of atoms and the addition/removal of distinguished variables. Last but not least, another interesting problem that largely remains open in the area of query rewriting is how to efficiently evaluate the computed UCQ rewriting over a database. We feel that the incremental algorithm can provide some new opportunities in addressing this problem.

Acknowledgments Work by Giorgos Stoilos is supported by a Marie Curie Career Reintegration Grant within European Union's Seventh Framework Programme (FP7/2007-2013) under REA grant agreement 303914.

A Omitted Proofs

Lemma 1 *Let q_1, q_2 be two CQs such that q_2 subsumes q_1 and let q_1' be the result of applying an axiom I to q_1. If q_2 does not subsume q_1', then either I is applicable to q_2 and the result subsumes q_1' or for $1 \leq i \leq n$, atoms of the form $P(u, v)$, $P(z_i, v)$ or $P(v, u)$, $P(v, z_i)$ exist in q_2 such that for $1 \leq i, j \leq n$, $i \neq j$ we have $z_i \neq z_j$ and for $\lambda = \{z_i \mapsto u \mid 1 \leq i \leq n\}$, I is applicable to $q_{2\lambda}$ and the result subsumes q_1'.*

Proof Since q_2 subsumes q_1, there exists a substitution θ s.t. $[\{Q(\mathsf{avar}(q_2))\} \cup q_2]_\theta \subseteq \{Q(\mathsf{avar}(q_1))\} \cup q_1$; however, Q does not appear in either q_1 or q_2, hence $q_{2\theta} \subseteq q_1$. Moreover, since q_2 does not subsume q_1' we have $q_{2\theta} \nsubseteq q_1'$. Hence, since q_1' is the result of applying I to q_1 we have that I is applied to an atom $\alpha \in q_1$ that is also in $q_{2\theta}$ replacing it with a new atom β that is not in $q_{2\theta}$. Next, $\alpha \in q_{2\theta}$ implies that q_2 contains an atom α_0 such that $\alpha = \alpha_{0\theta}$, i.e., an atom with the same predicate name as α, but possibly containing different variables. By case analysis on the different types of applications between axioms and query atoms we show that either I is also applicable to α_0 and for q_2' the result of applying I to q_2, θ can be extended to a new substitution σ s.t. $q_{2\sigma}' \subseteq q_1'$, or for $i \geq 1$ atoms of the form $P(u, v)$, $P(z_i, v)$ or $P(v, u)$, $P(v, z_i)$ exist in q_2 such that for $\lambda = \{z_i \mapsto u \mid i \geq 1\}$, I is applicable to $q_{2\lambda}$ and the result subsumes q_1'.

Recall first that $\alpha \in q_1$, $\alpha \in q_{2\theta}$, $\alpha_0 \in q_2$, and $\beta \in q_1'$. We have the following cases:

1. If $\alpha = A(x)$, then I is of the form $C \sqsubseteq A$, with C a concept, β is of the form $C(x)$, while α_0 is of the form $A(u)$, with $u \mapsto x \in \theta$. Then, clearly, I is also applicable to q_2 creating a new query q_2' that contains $C(u)$. Then, we have the following cases:
   - If I is of the form $B \sqsubseteq A$, then $\beta = B(x)$. In this case, $B(u) \in q_2'$, and for $\sigma = \theta$ we have $B(x) \in q_{2\sigma}'$ which implies that $q_{2\sigma}' \subseteq q_1'$.
   - If I is of the form $\exists P \sqsubseteq A$, then $\beta = P(x, y)$ for y a new variable in q_1. In this case, $P(u, z) \in q_2'$ for z a new variable in q_2. Since z is new in q_2, θ can be extended to $\sigma = \theta \cup \{z \mapsto y\}$. Then, $P(x, y) \in q_{2\sigma}'$ which implies that $q_{2\sigma}' \subseteq q_1'$.
   - The case where I is of the form $\exists P^- \sqsubseteq A$ is symmetric with the previous one.
2. If $\alpha = P(x, y)$ and I is of the form $S \sqsubseteq P$ (or equivalently $S^- \sqsubseteq P^-$), then $\beta = S(x, y)$, while α_0 is of the form $P(u, v)$ with $\{u \mapsto x, v \mapsto y\} \subseteq \theta$. Then, clearly I is also applicable to q_2 and its application creates q_2' with $S(u, v) \in q_2'$, while for $\sigma = \theta$ we have $S(x, y) \in q_{2\sigma}'$ which implies that $q_{2\sigma}' \subseteq q_1'$. The cases where I is of the form $S \sqsubseteq P^-$ or $S^- \sqsubseteq P$ can be shown similarly.
3. If $\alpha = P(x, y)$, y is unbound in q_1, and I is of the form $C \sqsubseteq \exists P$ with C a concept, then $\beta = C(x)$ and $\alpha_0 = P(u, v)$ with $\{u \mapsto x, v \mapsto y\} \subseteq \theta$. If v is also unbound in q_2, then I is applicable to q_2 and for q_2' the result we will have $A(u) \in q_2'$; hence, for $\sigma = \theta$ we will have $A(x) \in q_{2\sigma}'$ and also $q_{2\sigma}' \subseteq q_1'$. If v is not unbound in q_2 this implies that there is another atom α_2 in q_2 that mentions v. α_2 cannot be a concept atom of the form $A(v)$ because due to $q_{2\theta} \subseteq q_1$ we would have $A(v)_\theta = A(y) \in q_1$ and hence y would not be unbound in q_1 as well (it would appear in $A(y)$ and $P(x, y)$) leading to a contradiction; hence, α_2 must be a role atom. However, α_2 cannot be of the form $P(v, z)$, for z an arbitrary variable since then, again $q_{2\theta} \subseteq q_1$ implies that $P(v, z)_\theta = P(y, z_\theta) \in q_1$ and thus y would not be unbound in q_1. The only possible case is that α_2 is of the form $P(z, v)$ with $z \mapsto x \in \theta$ (again if z is mapped to a different variable than x, this would imply that q_1 has two role atoms both having y as a second argument and y would not be unbound in q_1). Consequently, we have $\alpha_{0\theta} = P(u, v)_\theta = P(z, v)_\theta = a_{2\theta}$, which implies that the two atoms unify. Now, let the substitution $\sigma_1 = \{z \mapsto u\}$. As mentioned before, θ maps both z and u to x; hence, for the defined σ_1 we have $q_{2\sigma_1 \circ \theta} = q_{2\theta}$ and hence $q_{2\sigma_1}$ also subsumes q_1. Consequently, all previous conditions apply to $q_{2\sigma_1}$—that is, either $q_{2\sigma_1}$ subsumes q_1', or I is applicable to $q_{2\sigma_1}$ and the result subsumes q_1'. If neither of these is the case then again an atom $P(z', v)$ with $z' \neq z$ must exist in $q_{2\sigma_1}$, such

that it unifies with $P(u, v)$ and a σ_2 can be constructed such that $q_{2\sigma_1\circ\sigma_2}$ subsumes q_1. Since, there is only a finite number of atoms in a query, at some point some $\lambda = \{z_i \mapsto u \mid i \geq 1\}$ exists such that I is applicable to $q_{2\lambda}$ and the result subsumes q'_1. Note also that each σ_i is independent from the other and hence they can be applied in an arbitrary order over q_2.

4. If $\alpha = P(y, x)$, y is unbound in q_1, and I is of the form $C \sqsubseteq \exists P^-$, then $\beta = C(x)$ and $\alpha_0 = P(v, u)$ with $\{v \mapsto y, u \mapsto x\} \subseteq \theta$. Following a similar reasoning as in the previous case we can deduce that, either v is also unbound in q_2 so I is applicable to q_2 and the result subsumes q'_1, or it is not and in this case a list of atoms $P(v, z_i)$ exists such that for $\lambda = \{z_i \mapsto u\}$, I is applicable to $q_{2\lambda}$ and the result subsumes q'_1. □

Lemma 2 *Let q be a CQ that contains only one body atom and let κ be a substitution such that some axiom I is applicable to q_κ producing q'. Then I is also applicable to q and for q'' the result we have $q''_\kappa = q'$ (modulo renaming of fresh variables).*

Proof If κ does not change any variables of q then the property follows trivially. If it does, then we show the property by performing a case analysis on the form of q.

1. q is of the form $\{x \mid A(x)\}$. Then, κ can be of the form $\{x \mapsto y\}$ and I is of the form $C \sqsubseteq A$, with C a concept. In this case, $q' = \{y \mid C(y)\}$, while I is also applicable to q and the result is $q'' = \{x \mid C(x)\}$. If C is an atomic concept then clearly $q''_\kappa = q'$. If C is of the form $\exists R$ for R a role, then $q' = \{y \mid R(y, z_1)\}$ and $q'' = \{x \mid R(x, z_2)\}$, for z_1, z_2 fresh variables and again (modulo renaming of fresh variables) $q''_\kappa = q'$.
2. q is of the form $\{x \mid R(x, y)\}$. Then we have two cases depending on I.

(a) If I is of the form $C \sqsubseteq \exists R$, then κ cannot map y to x (and vice versa), as this would make x (y) bound in q_κ. Hence, κ (possibly) maps x and/or y to different variables. In any case $q' = \{x_\kappa \mid C(x_\kappa)\}$. Clearly, I is applicable to q and again as in case 1. we can deduce that $q''_\kappa = q'$.
(b) If I is of the form $P \sqsubseteq R$ and κ is as in the previous case, then the claim follows by a similar argument. However, κ can be of the form $\{y \mapsto x\}$ (or $\{x \mapsto y\}$). Then, $q' = \{x \mid P(x, x)\}$ (or $q' = \{y \mid P(y, y)\}$). Clearly, I is also applicable to q and the result is $q'' = \{x \mid P(x, y)\}$ for which $q''_\kappa = \{x \mid P(x, x)\}$ (or $q''_\kappa = \{y \mid P(y, y)\}$). In either case $q''_\kappa = q'$.

3. q is of the form $\{x, y \mid R(x, y)\}$ and I is of the form $P \sqsubseteq R$. This case is similar to case 2(b).

Theorem 1 *Let q be a CQ, let $\mathcal{T}$ be a TBox, let α be an atom such that $\mathsf{var}(\alpha) \cap \mathsf{var}(q) \neq \emptyset$ and let $\mathcal{G}$ be a reformulation-closed rewriting graph for $q, \mathcal{T}$. Let $\mathcal{G}'$ be the graph returned by Algorithm 1 when applied to $\mathcal{G}, \alpha$ and $\mathcal{T}$; then $\mathcal{G}'$ is a reformulation-closed rewriting graph for $q \cup \{\alpha\}, \mathcal{T}$.*

Proof First, note that ex-PerfectRef extends PerfectRef, hence for a given query q and TBox $\mathcal{T}$ it computes a rewriting graph for $q, \mathcal{T}$. Moreover, since PerfectRef applies exhaustively reformulation and reduction and does not prune computed queries, the output of the extended algorithm is reformulation-closed.

Let $\mathcal{G} = \langle u, \mathcal{H}, m\rangle$ be the input of Algorithm 1 and let $\mathcal{G}' = \langle u', \mathcal{H}', m'\rangle$ be its output. Moreover, for the following of the proof, let $jv = \mathsf{var}(\alpha) \cap \mathsf{avar}(q)$, let $\mathcal{G}_\alpha = \langle u_\alpha, \mathcal{H}_\alpha, m_\alpha\rangle$ be the rewriting graph computed by Algorithm 1 in line 1, and let u_i be a UCQ rewriting computed after i steps of the PerfectRef algorithm when applied to $q \cup \{\alpha\}, \mathcal{T}$—that is, after generating i queries by an application of the reformulation or the reduction step.

Correctness of the claim is strongly based of the following property:

($\star$): For all $i \geq 0$ and for all q_j in u_i a vertex q_h in $\mathcal{G}$ and a vertex q_α in $\mathcal{G}_\alpha$ exist such that, for $\kappa = m(q_h)$ and $nv = \mathsf{avar}(q_h) \cup (\mathsf{avar}(q) \cap \mathsf{var}((q_\alpha)_\kappa))$ we have $\mathsf{canBeJoined}(q_h, \kappa, jv) = \mathsf{true}$, and one of the following conditions hold:

1. $\{nv \mid q_h \cup (q_\alpha)_\kappa\}$ subsumes q_j, or
2. for some $\sigma \in \mathsf{mergeCQs}((q_\alpha)_\kappa, q_h)$ some CQ q' exists such that $q_h \rightsquigarrow_\mathcal{G} q'$ and $\{nv \mid q'\}_{\mu'}$, where $\mu' = \mathsf{buildSubst}(\sigma, m(q'))$ subsumes q_j.

Assume for now that Property ($\star$) holds. We will show that $\mathcal{G}' = \langle u', \mathcal{H}', m'\rangle$ is a reformulation-closed rewriting graph for $q \cup \{\alpha\}, \mathcal{T}$.

First, we show that u' is a UCQ rewriting for $q \cup \{\alpha\}, \mathcal{T}$. Assume that when applied to $q \cup \{\alpha\}$ and $\mathcal{T}$ the PerfectRef algorithm terminates after n steps and that u_n is the computed UCQ rewriting. But then, by Property ($\star$), we know that upon termination of Algorithm 1, for each CQ $q_n \in u_n$ either a CQ of the form $\{nv \mid q_h \cup (q_\alpha)_\kappa\}$ or a CQ of the form $\{nv \mid q'\}_{\mu'}$ that subsumes q_n has been added to u'. Moreover, functions canBeJoined, buildSubst, and the condition $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(q') \cup \mathsf{dom}(m(q'))$ of the respective algorithms ensure that every CQ produced by our algorithm can also be produced by the PerfectRef algorithm. Consequently, u' is both sound and complete, thus it is indeed a UCQ rewriting for $q \cup \{\alpha\}, \mathcal{T}$.

Now we show that $\mathcal{G}'$ is reformulation-closed. First, by definition of the mappings $m'(q'_n)$ for each CQ added to $\mathcal{G}'$ it follows that m' satisfies the conditions in Definition 3. Furthermore, upon termination of the algorithm also the relation $\mathcal{H}'$ satisfies the property of Definition 3. Moreover, again it can be easily seen that the algorithm will create and set as a top element of $\mathcal{G}'$ the CQ $q \cup \{\alpha\}$ with $m'(q \cup \{\alpha\}) = \emptyset$, while for all top elements q_i in $\mathcal{G}'$ we will also have $m(q_i) = \emptyset$ as they inherit the mapping of q (we elaborate more on this in the proof of Property ($\star$) later on). In addition, $\mathcal{H}'$ is closed under restricted reduction; if for some $q_h \in u, \kappa = m(q_h)$ and $q_\alpha \in u_\alpha$ we have some $\sigma \in \mathsf{mergeCQs}((q_\alpha)_\kappa, q_h)$, then the algorithm first adds $q^+ = \{nv \mid q_h \cup (q_\alpha)_\kappa\}$ to $\mathcal{G}'$ (line 13) and then adds $\langle q^+, \{nv \mid (q_h)\}_\sigma\rangle$ to $\mathcal{G}'$ (line 15). Then, it copies the sub-graph of $\mathcal{H}$ below q_h, but $\mathcal{H}$ is also reformulation-closed. The only interesting case is if for some q' such that $q_h \rightsquigarrow_\mathcal{G} q'$ we have $z \mapsto x \in \sigma$ and $z \mapsto x' \in m(q')$. The latter implies that there exists q'' such that $\langle q'', q'\rangle \in \mathcal{H}$ and $R(x', y), R(z, y)$ are atoms in q'' while $q' = q''_{z\mapsto x'}$, i.e., every occurence of z in q'' has been mapped to x'. In the new graph $\mathcal{H}'$, the algorithm will rather contain the query q''_σ, i.e., q''_σ will now contain atoms $R(x', y)$, $R(x, y)$ instead of $R(x, y)$, $R(z, y)$, i.e., occurences of z have been mapped to x. Subsequently, a child of q''_σ needs to be created. For $\mathcal{H}'$ to be reformulation-closed the child needs to be a query where now x, rather than z, is mapped to x'. This is accomplished by function $\mathsf{buildSubst}$. More precisely, since $z \mapsto x \in \sigma$ and $z \mapsto x' \in m(q')$, the function returns a mapping μ that contains $z \mapsto x$ and $x \mapsto x'$ which is applied to q' creating the appropriate query. Consequently, $\mathcal{G}'$ is reformulation-closed for $q \cup \{\alpha\}, \mathcal{T}$.

Now, we give the omitted proof of Property ($\star$).

To show this property we use induction over the number of steps i that algorithm $\mathsf{ex\text{-}PerfectRef}$ has completed when applied to $q \cup \{\alpha\}, \mathcal{T}$. For a CQ q we will use the notation m_q to denote the substitution $m(q)$.

Base Case ($i = 0$): On the one hand, when algorithm $\mathsf{ex\text{-}PerfectRef}$ is applied to $q \cup \{\alpha\}, \mathcal{T}$, at step 0 it initialises a UCQ u_0 to contain the vertex $q \cup \{\alpha\}$. On the other hand, in line 1 Algorithm 1 computes $\mathcal{G}_\alpha$ for $q_\alpha = \{\mathsf{var}(\alpha) \cap \mathsf{var}(q) \mid \alpha\}$ and $\mathcal{T}$ which uses $\mathsf{ex\text{-}PerfectRef}$ and hence at least contains q_α as a vertex. Moreover, since $\mathcal{G}$ is reformulation-closed for $q, \mathcal{T}$, then q appears as a top element in $\mathcal{G}$ with $m_q = \emptyset$. Hence, by construction of q_α and jv we have $\mathsf{canBeJoined}(q, m_q, jv) = \mathsf{true}$ and also $\{nv \mid q \cup (q_\alpha)_{m_q}\}$ is exactly $q \cup \{\alpha\}$. Consequently, condition 1 is satisfied and hence also Property ($\star$) is satisfied at step $i = 0$.

Induction Step: Assume that Property ($\star$) holds at step i for each $q_j \in u_i$. Subsequently, suppose that at step $i + 1$ the algorithm produces the UCQ u_{i+1} by applying a reformulation or a reduction step on some query q_j in u_i producing a query q_{i+1}. Moreover, by induction hypothesis, vertices q_h in $\mathcal{G}$ and q_α in $\mathcal{G}_\alpha$ exist such that Property ($\star$) is satisfied—that is, for $\kappa = m_{q_h}$, we have $\mathsf{canBeJoined}(q_h, \kappa, jv) = \mathsf{true}$ and either the query $\{nv \mid q_h \cup (q_\alpha)_\kappa\}$ (called q^+ in the following) subsumes q_j, or for some $\sigma \in \mathsf{mergeCQs}((q_\alpha)_\kappa, q_h)$ and some q' such that $q_h \rightsquigarrow_\mathcal{G} q'$, we have that $\{nv \mid q'\}_{\mu'}$ (called q^m in the following) subsumes q_j. We now distinguish between two cases according to whether q_{i+1} was produced by a reformulation or a reduction step.

If a reduction step was applied to q_j to produce q_{i+1}, then it is well-known by the properties of reduction that q_j subsumes q_{i+1}. Hence, by the transitivity of the subsumes relation and the induction hypothesis q_{i+1} is also subsumed by either q^+ or q^m. Consequently, Property ($\star$) is still satisfied.

Otherwise, assume that a reformulation step was applied to q_j to produce q_{i+1}. This implies that some axiom I is applied to some body atom of q_j producing q_{i+1}. By induction hypothesis, either q^+ or q^m subsume q_j. If either of these queries subsumes q_{i+1}, then Property ($\star$) is satisfied. If neither query subsumes q_{i+1}, then by Lemma 1, either (i) I is applicable to them and the result subsumes q_{i+1} or (ii) for $n \geq 1$ atoms of the form $P(u, v)$, $P(z_i, v)$ or $P(v, u)$, $P(v, z_i)$ exist in q^+ or in q^m such that for $\lambda = \{z_i \mapsto u \mid 1 \leq i \leq n\}$, q^+_λ or q^m_λ subsumes q_j, I is applicable to q^+_λ or q^m_λ and the result subsumes q_{i+1}. We now examine separately the cases that q^+ or q^m subsume q_j.

1. Assume that q^+ subsumes q_j and also that we have case (i) from above—that is, I is applicable to $q^+ = \{nv \mid q_h \cup (q_\alpha)_\kappa\}$. Since reformulation applies an axiom to a single body atom of a query at each time, this implies that I is either applicable to some body atom of q_h and the application creates a CQ of the form $\{nv \mid q'_h \cup (q_\alpha)_\kappa\}$, or it is applicable to some body atom of $(q_\alpha)_\kappa$ and the application creates a CQ of the form $\{nv \mid q_h \cup q'_\alpha\}$. (Note that reformulation does not change the distinguished variables of the query) In the former case, since I is applicable to q_h producing q'_h, q_h is a vertex in $\mathcal{G}$ and $\mathcal{G}$ is reformulation-closed, then $\langle q_h, q'_h\rangle \in \mathcal{H}$ with $m_{q'_h} = \kappa$ and $\mathsf{avar}(q_h) = \mathsf{avar}(q'_h)$. Moreover, $\mathsf{canBeJoined}(q'_h, m_{q'_h}, jv) = \mathsf{true}$; this is because I is applicable to $\{nv \mid q_h \cup (q_\alpha)_\kappa\}$, hence its application can only remove variables that do not appear in $(q_\alpha)_\kappa$. Consequently, condition 1 is satisfied and thus also Property ($\star$). Now we study the case when I is applicable to $(q_\alpha)_\kappa$: First, note that q_α contains exactly one body atom; this follows straightforwardly by the fact that the query for which $\mathcal{G}_\alpha$ is computed contains exactly one atom and by the fact that the $\mathsf{PerfectRef}$ calculus always produces a new query that contains at most the same number of atoms as one from which it is produced. Hence, by Lemma 2, I is also applicable to q_α and for the result q''_α we have $(q''_\alpha)_\kappa = q'_\alpha$. But since $q_\alpha \in u_\alpha$ and $\mathcal{G}_\alpha$ is reformulation-closed, then $q''_\alpha \in u_\alpha$. Consequently, again condition 2. is satisfied thus also Property ($\star$).
   Assume now that we have case (ii) from above—that is, for $n \geq 1$ there are atoms of the form $P(u, v)$, $P(z_i, v)$ or of the form $P(u, v)$, $P(z_i, v)$ in q^+ such that for $\lambda = \{z_i \mapsto u \mid 1 \leq i \leq n\}$, I is applicable to $q^+_\lambda = \{nv_\lambda \mid (q_h)_\lambda \cup (q_\alpha)_{\kappa\circ\lambda}\}$ and the new CQ produced by applying I on q^+_λ, called q_f, subsumes q_{i+1}. We will show that our algorithm can compute queries

q^+_λ and q_f. First note that the existence of atoms of the above form in q^+ implies that several restricted reduction steps are applicable between body atoms $P(z_i, v)$ and the atom $P(u, v)$ in q'_j. Moreover, since $(q_\alpha)_\kappa$ contains a single atom, then these reductions are either applicable over two atoms from q_h or between an atom from q_h and the atom in $(q_\alpha)_\kappa$. In the former case, an application of one of these reductions would produce the CQ $(q_h)_{\lambda_i} \cup (q_\alpha)_{\kappa \circ \lambda_i}$. Since graph $\mathcal{G}$ is reformulation-closed and by the form of these reductions, also $\langle q_h, q'_h \rangle \in \mathcal{G}$, with $q'_h = (q_h)_{\lambda_i}$ and $m_{q'_h} = \kappa \circ \lambda_i$; hence, $\mathsf{canBeJoined}(q'_h, m_{q'_h}, jv) = \mathsf{true}$. If none of these reductions is performed between $(q_\alpha)_\kappa$ and q_h, then for λ as defined before we will have $q_h \rightsquigarrow_\mathcal{G} (q_h)_\lambda$ with $m_{(q_h)_\lambda} = \kappa \circ \lambda$ and hence also $\mathsf{canBeJoined}((q_h)_\lambda, m_{(q_h)_\lambda}, jv) = \mathsf{true}$. Then, axiom I is applicable to $(q_h)_\lambda$ and since $\mathcal{G}$ is reformulation-closed we also have $\langle (q_h)_\lambda, q'' \rangle \in \mathcal{H}$ and $m_{q''} = m_{q_{h\lambda}}$. Finally, $\{nv' \mid q'' \cup (q_\alpha)_{m_{q''}}\}$ for $nv' = \mathsf{avar}(q'') \cup (\mathsf{avar}(q) \cap \mathsf{var}((q_\alpha)_{m_{q''}}))$ is q_f and it subsumes q_{i+1}, hence Property ($\star$) is satisfied.

Assume now that for some $k \le n$, $(q_\alpha)_\kappa$ contains atom $P(z_k, u)$ and one reduction step involves $(q_\alpha)_\kappa$ and atoms from q_h. In this case since $(q_\alpha)_\kappa$ contains one atom then this reduction actually eliminates this atom from q^+ and hence $q^+_{\lambda_k}$ is of the form $\{\mathsf{avar}(q)_{\lambda_k} \mid (q_h)_{\lambda_k}\}$. Consequently, applying the rest of reductions the final CQ q^+_λ would be of the form $\{\mathsf{avar}(q)_\lambda \mid (q_h)_\lambda\}$ and q_f can be produced by applying I on this CQ. Now, since $\mathcal{G}$ is reformulation-closed all reductions but λ_k have been applied on queries in $\mathcal{G}$ starting from q_h, hence a CQ $(q_h)_\nu$, with $\nu = \lambda \setminus \lambda_k$ exists in $\mathcal{G}$ such that $q_h \rightsquigarrow_\mathcal{G} (q_h)_\nu$. Moreover, since $(q_h)_\nu$ does not contain the atom that is in $(q_\alpha)_\kappa I$ is applicable to it and hence $\langle (q_h)_\nu, q'' \rangle \in \mathcal{H}$. Then, by definition of function $\mathsf{mergeCQs}$, we have that $\mathsf{mergeCQs}((q_\alpha)_\nu, (q_h)_\nu)$ contains $\{\lambda_k\}$. Hence, the CQ q''_{λ_k} is actually q_f that subsumes q_{i+1} and Property ($\star$) is satisfied again.

2. Assume now that q^m, i.e., $\{nv \mid q'\}_{\mu'}$ where $q_h \rightsquigarrow_\mathcal{G} q'$, $\mu' = \mathsf{buildSubst}(\sigma, m(q'))$ and $\sigma \in \mathsf{mergeCQs}((q_\alpha)_\kappa, q_h)$ subsumes q_j. Again by Lemma 1 either I is applicable to q^m and the result (call it q_f) subsumes q_{i+1} or further restricted reduction steps are applicable to q^m producing a CQ over which I is applicable and the result (call it again q_f) subsumes q_{i+1}.

   Since $q'_{m(q')} = q'$ and since μ' is constructed from σ and $m(q')$, then $\{nv \mid q'\}_{\mu'}$ is like q' but perhaps with some variable $z \in \mathsf{var}(q')$ renamed due to a mapping $z \mapsto y \in \sigma$. Moreover, since $\{nv \mid q'\}_{\mu'}$ does not mention z (it has been renamed by μ'), then neither of the previous steps (application of I or restricted reductions) does involve z. Now, due to μ' either q' has fewer bound variables than $\{nv \mid q'\}_{\mu'}$ or μ' frees a bound variable that is not free in q' after which some axiom might be applicable. In the former case (q' has fewer bound variables), at least the same sequence of reformulation or restricted reduction steps as to q^m are applicable to q'. Since these do not involve variable z, after applying them, a CQ q'' such that $\{nv \mid q''\}_{\mu''} = q_f$, where $\mu'' = \mathsf{buildSubst}(\sigma, m(q''))$ is generated, as μ'' built from σ will perform the same variable renaming on z. In addition, since $\mathcal{G}$ is reformulation-closed q'' is in $\mathcal{G}$. Now consider the latter case, i.e., μ' frees a bound variable in q'. This implies that μ' induces a mapping $\{z \mapsto x\}$ that frees some bound variable in q'. Consequently, a restricted reduction with unifier $\sigma' = \{z \mapsto x\}$ is applicable and since $\mathcal{G}$ is reformulation-closed we have that $\langle q', q'_{\sigma'} \rangle \in \mathcal{H}$. Subsequently, the same sequence of reformulation and restricted reduction steps as to q^m are applicable to $q'_{\sigma'}$. Hence again, we will have a query q'' such that $\{nv \mid q''\}_{\mu''} = q_f$, where $\mu'' = \mathsf{buildSubst}(\sigma, m(q''))$ is produced in $\mathcal{G}$. In any case Property ($\star$) is satisfied. □

References

1. Acciarri A, Calvanese D, De Giacomo G, Lembo D, Lenzerini M, Palmieri M, Rosati R (2005) Quonto: querying ontologies. In: Proceedings of the 20th international conference on artificial intelligence (AAAI-05)
2. Artale A, Calvanese D, Kontchakov R, Zakharyaschev M (2009) The DL-Lite family and relations. J Artif Intell Res 36:1–69
3. Calì A, Gottlob G, Lukasiewicz T (2009) A general datalog-based framework for tractable query answering over ontologies. In: Proceedings of the twenty-eigth ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems, PODS 2009, pp 77–86
4. Calì A, Gottlob G, Pieris A (2009) Tractable query answering over conceptual schemata. In: Proceedings of the 28th international conference on conceptual modeling (ER 2009), pp 175–190
5. Calì A, Gottlob G, Pieris A (2010) Advanced processing for ontological queries. Proceedings of the VLDB Endowment 3(1):554–565
6. Calì A, Gottlob G, Pieris A (2012) Ontological query answering under expressive entity-relationship schemata. Inf Syst 37(4):320–335
7. Calvanese D, De Giacomo G, Lembo D, Lenzerini M, Rosati R (2006) Data complexity of query answering in description logics. In: Proceedings of the 10th international conference on principles of knowledge representation and reasoning (KR 06), pp 260–270
8. Calvanese D, De Giacomo G, Lembo D, Lenzerini M, Rosati R (2007) Tractable reasoning and efficient query answering in description logics: the DL-Lite family. J Autom Reasoning 39(3):385–429
9. Chortaras A, Trivela D, Stamou G (2011) Optimized query rewriting in OWL 2 QL. In: Proceedings of the 23rd international conference on automated deduction (CADE 23), Polland, pp 192–206
10. Cuenca Grau B, Horrocks I, Motik B, Parsia B, Patel-Schneider P, Sattler U (2008) OWL 2: the next step for OWL. J Web Semantics (JWS) 6(4), 309–322
11. Gottlob G, Orsi G, Pieris A (2011) Ontological queries: rewriting and optimization. In: Proceedings of the 27th international conference on data engineering (ICDE 2011), pp 2–13

12. Gottlob G, Orsi G, Pieris A (2011) Ontological query answering via rewriting. In: Proceedings of the 15th international conference on advances in databases and information systems, Springer-Verlag, pp 1–18
13. Gottlob G, Schwentick T (2012) Rewriting ontological queries into small nonrecursive datalog programs. In: Proceedings of the 13th international conference on principles of knowledge representation and reasoning (KR 2012)
14. Gupta A, Mumick IS, Rossl KA (1995) Adapting materialized views after redefinitions. In: Proceedings of ACM SIGMOD international conference on management of data, pp 211–222
15. Horrocks I, Patel-Schneider PF, van Harmelen F (2003) From SHIQ and RDF to OWL: the making of a web ontology language. J Web Semantics 1(1):7–26
16. Jansen BJ, Spink A, Blakely C, Koshman S (2007) Defining a session on web search engines: research articles. J Am Soc Inf Sci Tech 58:862–871
17. Jansen BJ, Spink A, Pedersen J (2005) A temporal comparison of altavista web searching: research articles. J Am Soc Inf Sci Tech 56:559–570
18. Kikot S, Kontchakov R, Zakharyaschev M (2011) On (in)tractability of OBDA with OWL 2 QL. In: Proceedings of the 24th international workshop on description logics (DL 2011)
19. Kontchakov R, Lutz C, Toman D, Wolter F, Zakharyaschev M (2010) The combined approach to query answering in DL-Lite. In: Proceedings of the 20th international conference principles of knowledge representation and reasoning (KR 2010)
20. Lutz C (2008) The complexity of conjunctive query answering in expressive description logics. In: Proceedings of the 4th international joint conference on automated reasoning, IJCAR 2008, pp 179–193
21. Lutz C, Toman D, Wolter F (2009) Conjunctive query answering in the description logic EL using a relational database system. In: Proceedings of the 21st international joint conference on artificial intelligence (IJCAI 2009), pp 2070–2075
22. Mohania M (1997) Avoiding re-computation: view adaptation in data warehouses. In: Proceedings of 8 th international database workshop, pp 151–165
23. Motik B, Cuenca Grau B, Horrocks I, Wu Z, Fokoue A, Lutz C (2009) OWL 2 web ontology language profiles
24. Orsi G, Pieris A (2011) Optimizing query answering under ontological constraints. PVLDB 4(11):1004–1015
25. Ortiz M, Calvanese D, Eiter T (2008) Data complexity of query answering in expressive description logics via tableaux. J Autom Reasoning 41(1):61–98
26. Pass G, Chowdhury A, Torgeson C (2006) A picture of search. In: Proceedings of the 1st international conference on scalable information systems (InfoScale 06). ACM (2006)
27. Pérez-Urbina H, Horrocks I, Motik B (2009) Efficient query answering for OWL 2. In: Proc. of the international semantic web conference (ISWC2009), pp 489–504
28. Pérez-Urbina H, Motik B, Horrocks I (2010) Tractable query answering and rewriting under description logic constraints. J Appl Logic 8(2):186–209
29. Poggi A, Lembo D, Calvanese D, De Giacomo G, Lenzerini M, Rosati R (2008) Linking data to ontologies. J Data Semantics X, 133–173
30. Rodriguez-Muro M, Calvanese D (2012) High performance query answering over DL-Lite ontologies. In: Proceedings of the 13th international conference principles of knowledge representation and reasoning, KR 2012
31. Rosati R (2012) Query rewriting under extensional constraints in DL-Lite. In: Proceedings of the international workshop on description logics, DL-2012
32. Rosati R, Almatelli A (2010) Improving query answering over DL-Lite ontologies. In: Proceedings of the international conference on principles of knowledge representation and reasoning (KR-10)
33. Venetis T, Stoilos G, Stamou G (2011) Query rewriting under query extensions for OWL 2 QL ontologies. In: Proceedings of the 7th international workshop on scalable semantic web Knowledge base systems (SSWS2011), Germany