Incremental Query Rewriting for OWL 2 QL

Tassos Venetis, Giorgos Stoilos, and Giorgos Stamou

School of Electrical and Computer Engineering
National Technical University of Athens, Greece

1 Introduction

A key application of Description-Logic based ontologies is Ontology-Based Data Access (OBDA) [9]. In such scenarios a TBox is used to describe the schema of the application while answers to conjunctive queries reflect both the schema and the data. Unfortunately, it is well-known that conjunctive query (CQ) answering over expressive Description Logics (DLs) is of very high computational complexity [7, 5].

The need for efficient query answering has motivated the development of (families of) *lightweight* ontology languages, such as the DL-Lite family [3, 2]. Query answering in these languages is usually performed via a technique called *query rewriting*. According to this technique, a query and a DL-Lite ontology are transformed into a union of conjunctive queries (often called a *UCQ rewriting*) such that, the answers of the union of conjunctive queries over the input data and discarding the ontology are precisely the answers of the original query over the data and the ontology.

In the last years a large number of different algorithms and systems for computing rewritings for DL-Lite ontologies has been presented. Examples of such systems are QuOnto [1], Requiem [8], Presto [10], Nyaya [6], and Rapid [4]. Roughly speaking, all systems apply a set of equivalence-preserving transformations over the input query and TBox producing new queries until a fix-point is reached. In several previous approaches [3, 8, 6] this process is largely brute-force, in the sense that the algorithm iterates over the currently computed set of queries, over the atoms of the query and over the TBox axioms, and if some of the rules of the algorithm applies then a new query is generated.

It was shown recently that given a query q, a TBox $\mathcal{T}$ and an atom α of q, a UCQ rewriting for $q, \mathcal{T}$ can be computed by first computing a UCQ rewriting u^- for query $q \setminus \{\alpha\}$ (i.e., q without the atom α) and then 'extending' this rewriting with additional information from $\mathcal{T}$ that *only* regards α [11]. Using this idea we present a novel algorithm for computing a UCQ rewriting for queries over DL-Lite$_R$-TBoxes[1] incrementally. Roughly speaking, given a query q with atoms $\alpha_1, \ldots, \alpha_n$ the algorithm first computes UCQ rewritings u_i for 'special' queries that contain only a single body atom α_i. Finally, these UCQs are iteratively 'combined' until a UCQ rewriting for the input query has been computed.

Compared to several previous approaches our algorithm is significantly guided. At each step all the knowledge of $\mathcal{T}$ that regards a single atom α_i is 'materialised'

[1] DL-Lite$_R$ is a popular member of the DL-Lite family [3].

into u_i and is used to extend the currently computed UCQ. Our approach also shows that the process of rewriting (at least for DL-Lite) can be largely performed in parallel by 'decomposing' q into parts and processing them separately, which to the best of our knowledge, was previously unknown.

Furthermore, to further increase the efficiency of the algorithm, we additionally present a list of optimisations which considerably decrease its computation time. Many of the optimisations are intended to increase the efficiency of our final backward-subsumption (redundancy elimination) algorithm.

Finally, we have implemented the proposed algorithm and optimisations and we have compared them against several available state-of-the-art systems. Our results show that computing a UCQ rewriting incrementally is in the vast majority of cases more efficient than all systems. More precisely, our algorithm requires less time and computes the smallest UCQ rewriting in nearly all ontologies. Interestingly, when compared to the original DL-Lite algorithm [3], which also uses the same technique to compile knowledge from $\mathcal{T}$, our algorithm is several orders of a magnitude faster, which shows the benefits of the more guided approach.

An extended version of the paper with detailed proofs of correctness can be found online.[2]

2 Preliminaries

Let $\mathbf{C}$, $\mathbf{R}$, and $\mathbf{I}$ be countable, pairwise disjoint sets of *atomic concepts*, *atomic roles*, and *individuals*. A *DL-Lite$_R$-role* is either an atomic role P or its *inverse* P^-. *DL-Lite$_R$-concepts* are defined inductively by the grammar $B := A \mid \exists R$, where $A \in \mathbf{C}$ and R is a DL-Lite$_R$-role. A *DL-Lite$_R$-TBox* is a finite set of axioms of the form $B_1 \sqsubseteq B_2$ or $B_1 \sqcap B_2 \sqsubseteq \bot$, with $B_{(i)}$ DL-Lite$_R$-concepts and $\bot$ the *bottom* concept that is empty in all interpretations, or of the form $R_1 \sqsubseteq R_2$ with $R_{(i)}$ DL-Lite$_R$-roles. 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 $\mathcal{O} = \mathcal{T} \cup \mathcal{A}$ consists of a TBox and an ABox.

A *conjunctive query* (CQ) q is an expression of the form $\{\vec{x} \mid \{\alpha_1, \ldots, \alpha_m\}\}$ where $\{\alpha_1, \ldots, \alpha_m\}$ is called the *body* of the query with α_i a concept or role atom of the form $A(t)$ or $R(t, t')$ (for t, t' function-free terms and A, R atomic) and $\vec{x} = (x_1, \ldots, x_n)$ is a tuple of variables called the *distinguished* (or answer) variables, each appearing in at-least some atom α_i. The remaining variables of q are called *undistinguished*. We use $\mathsf{var}(q)$ to denote all the variables appearing in q and $\mathsf{avar}(q)$ to denote all its distinguished variables. We often abuse notation and use q to refer to the set of its atoms, i.e., $\{\alpha_1, \ldots, \alpha_m\}$. Hence, for β an atom and q a CQ, $q \cup \{\beta\}$ denotes a new CQ that consists of the atoms of q plus β and the same distinguished variables as q. For the rest of the paper, and without loss of generality, we will assume that queries are *connected* [5]. Finally, a union of conjunctive queries (UCQ) is a set of CQs.

Given CQs q_1, q_2 with distinguished variables $\vec{x}$ and $\vec{y}$, respectively, we say that q_2 *subsumes* q_1, if there exists a substitution θ from the variables of q_2 to the

[2] `http://image.ece.ntua.gr/~gstoil/main.pdf`

variables of q_1 such that the set $[\{Q(\vec{y})\} \cup q_2]_\theta$ is a subset of the set $\{Q(\vec{x})\} \cup q_1$, where Q is a predicate of the same arity as $\vec{x}$ and $\vec{y}$ that does not appear in q_1 or q_2. Finally, for a UCQ u and CQ q, we say that q is *redundant* in u if another query in u exists that subsumes q; otherwise it is called *non-redundant*.

For a DL-Lite$_R$-TBox, a UCQ rewriting u for $q, \mathcal{T}$ can be computed using the *perfect reformulation* algorithm (PerfectRef) [3]. The algorithm applies exhaustively a *reformulation* and a *reduction* step that generate new CQs; the process terminates when no new CQ is generated. More precisely, in the reformulation step the algorithm picks a CQ q, an atom α in the body of the CQ and an axiom I in $\mathcal{T}$ and *applies* the axiom on α replacing it with a new atom. For example, for the query $q_1 = \{x \mid \{R(x, y), A(y)\}\}$ and the axiom $I_1 = \exists R \sqsubseteq A$, applying I_1 on atom $A(y)$ produces the new CQ $q_2 = \{x \mid \{R(x, y), R(z, y)\}\}$, where z is a 'fresh' variable. 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_2 above generates query $q_3 := \{x \mid \{R(x, y)\}\}$.

Let $G = \langle U, E\rangle$ be a graph. For $a, b \in U$ we say that b is *reachable* from a, written $a \rightsquigarrow_G b$, if $c_0, \ldots, c_n$ with $n \geq 0$ exist where $c_0 = a$, $c_n = b$ and $\langle c_i, c_{i+1}\rangle \in E$ for each $0 \leq i < n$. An element $c \in U$ is called *top* in G if for each $c' \in U$ we have $c \rightsquigarrow_G c'$.

3 Extending Query Rewritings

It has been shown in [11] that given a CQ q, a rewriting u for $q, \mathcal{T}$ and an atom α, a UCQ rewriting for the query $q' = q \cup \{\alpha\}, \mathcal{T}$ can be computed by re-using the previously computed (given) information for q. Roughly speaking, the algorithm computes a UCQ rewriting u_α for a query q_α that consists only of the atom α and then extends the queries in u with atoms of the queries from u_α. The following example illustrates this idea.

Example 1. Consider the following DL-Lite$_R$-TBox and CQ:

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

and the UCQ rewriting $u = \{q, q_1\}$ where $q_1 = \{x \mid \{\mathsf{Professor}(x)\}\}$ for $q, \mathcal{T}$ computed using PerfectRef. Assume now, that q is extended in order to retrieve only those individuals that teach students—that is, q is extended to $q' = \{x \mid \{\mathsf{teaches}(x, y), \mathsf{Student}(y)\}\}$. In order to compute a UCQ rewriting for $q', \mathcal{T}$ the algorithm presented in [11] proceeds as follows.

First, it constructs the query $q_\alpha = \{y \mid \{\mathsf{Student}(y)\}\}$ that consists of the single body atom α and its distinguished variables are the common variables between α and q. Then, a UCQ rewriting $u_\alpha = \{q_\alpha, q'_\alpha\}$ for $q_\alpha, \mathcal{T}$ is computed using PerfectRef, where $q'_\alpha = \{y \mid \{\mathsf{teaches}(z, y)\}\}$ for z a fresh variable. Subsequently, the algorithm initialises an empty UCQ u' and iterates over the sets u and u_α constructing and adding new queries to u' as follows:

1. The atoms of q_α are added to q; hence, query q' is added to u'.

2. The atoms of q'_α are added to q; hence, query $q'_1 = q \cup \{\mathsf{teaches}(z, y)\}$ is added to u'.
3. The algorithm identifies that the body atom of q'_α can be unified into the body of q; the result (i.e., CQ q) is added to u'. Additionally, since after this unification CQ q is part of the target UCQ u' all queries that are produced in u due to q also need to be added; hence, query q_1 is also added to u'.
4. No query is generated from q_1 and q_α (or q'_α) since q_1 does not contain all the distinguished variables of q_α (or q'_α), i.e., $\mathsf{avar}(q_\alpha) \not\subseteq \mathsf{var}(q_1)$.

It can be verified that the set $u' = \{q', q'_1, q, q_1\}$ is a UCQ rewriting for $q', \mathcal{T}$. $\diamondsuit$

Intuitively, the above approach is possible because the process of rewriting is to a large extend 'local' with respect to the atoms of a query. For example, the application of reformulation on some query atom is independent from the other atoms of the query, hence the information from $\mathcal{T}$ that regards α can be materialised and then used to extend the queries in u. The only exception is the reduction step where two different atoms are unified. This step was introduced in [3] because an axiom might only be applicable to a reduction of some query—that is, after reduction the reformulation procedure can continue. To tackle these cases the algorithm in [11] checks whether a query from u_α can be 'absorbed' ('merged') into a query q_i from u. Note, however, that the algorithm does not apply exhaustively all possible unifications as done in the original reduction step. In contrast, it unifies a query q_α into a query q in such a way that the queries that are (possibly) produced in u due to q can still be produced. This is similar to the *factorisation* optimisation [6]. Our algorithm uses the following function.

Function mergeCQs**:** Let q, q' be two queries. Then, function $\mathsf{mergeCQs}(q', q)$ returns a substitution σ defined as follows: (i) if there exists $\alpha \in q' \cap q$, then σ is the identity substitution; (ii) 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 pair-wise different, then $\sigma = \{z \mapsto x\}$; otherwise, $\sigma = \emptyset$.

In Example 1, for q'_α and q we have $\mathsf{mergeCQs}(q'_\alpha, q) = \{z \mapsto x\}$, hence q as well as all queries that are produced in u due to q (i.e., q_1) are added to the result. To accomplish the latter, however, the algorithm needs to be aware of the dependencies of the queries in the given (pre-computed) UCQ u. To capture this information, instead of a UCQ, the algorithm accepts as input a graph $\mathcal{G}$ of queries which encodes the dependencies between the queries in u.

Definition 1. *Let q be a CQ and let $\mathcal{T}$ be a DL-Lite$_R$-TBox. A* rewriting graph *for $q, \mathcal{T}$ is a directed 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 each node $q_i \in u$ is labelled with a substitution $m(q_i)$. Moreover, $\mathcal{G}$ satisfies the following properties: (i) 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, and (ii) 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 $m(q_2) = m(q_1) \circ \sigma$.*

Algorithm 1 IncrementalRew$(q, \mathcal{T})$

input: A CQ q and a DL-Lite$_R$-TBox $\mathcal{T}$.

1: Let S be the set of body atoms in q
2: Remove an atom α from S s.t. $\mathsf{var}(\alpha) \cap \mathsf{avar}(q) \neq \emptyset$
3: Set $av := \mathsf{var}(\alpha) \cap \mathsf{avar}(q)$ and $c_v := \mathsf{var}(\alpha)$
4: $\mathcal{G}_i := \mathsf{ex\text{-}PerfectRef}(\{av \mid \{\alpha\}\}, \mathcal{T})$
5: **while** $S \neq \emptyset$ **do**
6: Remove an atom α' from S s.t. $\mathsf{var}(\alpha') \cap c_v \neq \emptyset$
7: $jv := c_v \cap \mathsf{var}(\alpha')$
8: $av := av \cup (\mathsf{var}(\alpha') \cap \mathsf{avar}(q))$
9: $\mathcal{G}_\alpha := \mathsf{ex\text{-}PerfectRef}(\{jv \mid \{\alpha'\}\}, \mathcal{T})$
10: $\mathcal{G}' := \langle u', \mathcal{H}', m' \rangle$ for an empty UCQ u', binary relation $\mathcal{H}'$ and mapping m'
11: Let q_i be a top CQ in $\mathcal{G}_i$ and q_α a top CQ in $\mathcal{G}_\alpha$
12: $\mathsf{joinGraphs}(q_i, q_\alpha, \mathcal{G}', \mathcal{G}_i, \mathcal{G}_\alpha, av, jv)$
13: $c_v := c_v \cup \mathsf{var}(\alpha')$
14: $\mathcal{G}_i := \mathcal{G}'$
15: **return** $\mathsf{removeRedundant}(\mathcal{G})$

A rewriting graph for a query q over a DL-Lite$_R$-TBox can be easily computed by a straightforward extension of the PerfectRef algorithm, which we call ex-PerfectRef. The details of the algorithm have been presented in [11].

4 An Incremental Query Rewriting Algorithm

The previous results show that a rewriting for a given (fixed) query over some TBox can be computed incrementally by considering one of its atoms at a time. For example, for query $q' = \{x \mid \{\mathsf{teaches}(x, y), \mathsf{Student}(y)\}\}$ of Example 1 we can fist select atom $\alpha_1 := \mathsf{teaches}(x, y)$ compute a UCQ rewriting u_{α_1} for $q_{\alpha_1} = \{x \mid \{\mathsf{teaches}(x, y)\}\}$ (which consists of the set $\{q, q_1\}$ of the example) and then pick the last atom, compute a UCQ u_{α_2} for $q_{\alpha_2} := \{y \mid \{\mathsf{Student}(y)\}\}$ and finally extend u_{α_1} with atoms of queries from u_{α_2} as shown in Example 1. In general, given a (fixed) query one can pick one of its atoms, compute a rewriting (graph) for it, and then iteratively add the rest of its atoms by extending the previously computed rewriting. When all the atoms have been processed a UCQ rewriting for the given query would have been computed. In contrast to our previous work [11], at each step this algorithm should produce a rewriting graph out of the input rewriting graph instead of a UCQ in order to be able to iteratively process all the atoms.

This idea is illustrated in Algorithm 1. 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 $\mathcal{G}_i$ for the query $\{\mathsf{var}(\alpha) \cap \mathsf{avar}(q) \mid \{\alpha\}\}$ (line 4). Hence, initially a rewriting graph for a query that contains only atom α of q, variables $c_v := \mathsf{var}(\alpha)$ of q and distinguished variables $av := \mathsf{var}(\alpha) \cap \mathsf{avar}(q)$ of q have been computed. Then, the algorithm selects one-by-one the remaining atoms and extends the previously computed rewriting graph (lines 5–14). More

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

input: Rewriting graphs $\mathcal{G}' = \langle u', \mathcal{H}', m' \rangle$, $\mathcal{G} = \langle u, \mathcal{H}, m \rangle$ and $\mathcal{G}_\alpha = \langle u_\alpha, \mathcal{H}_\alpha, m_\alpha \rangle$ and two sets of variables jv and av.

1: $\kappa := m(q_h)$
2: **if** canBeJoined(q_h, κ, jv) **then**
3: Create CQ $q_c := \{av \mid q_h \cup (q_\alpha)_\kappa\}$, set $m'(q_c) := \kappa$ and add q_c to u'
4: Set $\sigma :=$ mergeCQs(q_α, q_h)
5: **if** $\sigma \neq \emptyset$ **then**
6: Add $\langle q_c, (q_h)_\sigma \rangle$ to $\mathcal{G}'$
7: **for all** q' s.t. $q_h \rightsquigarrow_{\mathcal{G}} q'$ **do**
8: Set $m'(\{av \mid q'\}_\sigma) := \kappa \circ \sigma$
9: **for all** $\langle q', q'' \rangle \in \mathcal{G}$ **do** Add $\langle \{av \mid q'\}_\sigma, \{av \mid q''\}_\sigma \rangle$ to $\mathcal{G}'$
10: **for all** $\langle q_\alpha, q' \rangle \in \mathcal{G}_\alpha$ **do**
11: Create CQ $q'_c := \{av \mid q_h \cup (q')_\kappa\}$, set $m'(q'_c) := \kappa$ and add $\langle q_c, q'_c \rangle$ to $\mathcal{G}'$
12: joinGraphs$(q_h, q', \mathcal{G}', \mathcal{G}, \mathcal{G}_\alpha, av, jv)$
13: **for all** $\langle q_h, q' \rangle \in \mathcal{G}$ **do**
14: **if** canBeJoined$(q', m(q'), jv)$ **then**
15: Create CQ $q'_c := \{av \mid q' \cup (q_\alpha)_\kappa\}$, set $m'(q'_c) := \kappa$ and add $\langle q_c, q'_c \rangle$ to $\mathcal{G}'$
16: joinGraphs$(q', q_\alpha, \mathcal{G}', \mathcal{G}, \mathcal{G}_\alpha, av, jv)$

precisely, at the beginning of the i-th iteration the algorithm has computed a rewriting graph $\mathcal{G}_i$ for a query q_i that contains i atoms of q, c_v contains the variables of q that appear in q_i, while av the distinguished variables of q that appear in q_i. Hence, it picks another atom α' such that some of its variables also appear in c_v (line 6), it adds the variables of α' that are distinguished in q to av (line 8), it computes a rewriting graph $\mathcal{G}_\alpha$ for the query $\{\mathsf{var}(\alpha') \cap c_v \mid \{\alpha'\}\}$ (line 9) and then, it joins $\mathcal{G}$ with $\mathcal{G}_{\alpha'}$ using function joinGraphs (line 12) storing the result to $\mathcal{G}'$. Finally, after processing all atoms of the query it uses the well-known redundancy elimination algorithm proposed in [8] to remove the redundant (subsumed) queries (line 15).

Function joinGraphs is shown in Algorithm 2. Intuitively, this algorithm computes the Cartesian product of the two input rewriting graphs (modulo cases where queries should be merged). 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 $q' \cup q_\alpha$ (for-loop in line 13). Similarly, for q a vertex in $\mathcal{G}$ and $\langle q_\alpha, q'_\alpha \rangle \in \mathcal{G}_\alpha$ (for-loop in line 10). In addition the algorithm also checks whether a CQ from $\mathcal{G}_\alpha$ can be merged into some CQ q_h from $\mathcal{G}$ (line 4). If this is the case then the queries that have been produced in $\mathcal{G}$ due to q_h are copied to the new rewriting graph (see lines 7–9).

Note that the graphs can be cyclic but standard graph-traversal algorithms can be used to guarantee termination.

Example 2. Consider the TBox $\mathcal{T}$ and CQ $q' = \{x \mid \{\mathsf{teaches}(x, y), \mathsf{Student}(y)\}\}$ of Example 1. A run of Algorithm 1 is the following:

(1.) First, it selects atom α s.t. $\mathsf{var}(\alpha) \cap \mathsf{avar}(q') \neq \emptyset$ (line 2). The only atom that satisfies this condition is $\alpha = \mathsf{teaches}(x, y)$. Subsequently, algorithm

ex-PerfectRef is executed for $q = \{x \mid \{\mathsf{teaches}(x, y)\}\}$ and $\mathcal{T}$. This creates the rewriting graph $\mathcal{G}_i = \langle u, \mathcal{H}, m\rangle$, where $u = \{q, q_1\}$ is as defined in Example 1, $\mathcal{H} = \{\langle q, q_1\rangle\}$ and $m(q) = m(q_1) = \emptyset$ (line 4). At this point $c_v = \{x, y\}$ and $av = \{x\}$.

(2.) Next, the algorithm picks another atom α' of q' s.t. $\mathsf{var}(\alpha') \cap c_v \neq \emptyset$. One such atom is $\alpha' = \mathsf{Student}(y)$. Hence, using again algorithm ex-PerfectRef it computes for $q_\alpha = \{y \mid \{\mathsf{Student}(y)\}\}$ and $\mathcal{T}$ (line 9) the rewriting graph $\mathcal{G}_\alpha = \langle u_\alpha, \mathcal{H}_\alpha, m_\alpha\rangle$, where $u_\alpha = \{q_\alpha, q'_\alpha\}$ is as defined in Example 1, $\mathcal{H} = \{\langle q_\alpha, q'_\alpha\rangle\}$ and $m(q_\alpha) = m(q'_\alpha) = \emptyset$. Subsequently, it calls algorithm joinGraphs with parameters q, q_α, $\mathcal{G}'$, $\mathcal{G}_i$ (as computed in the previous step), $\mathcal{G}_\alpha$ and the variable sets $av = \{x\}$ and $jv = \{y\}$ in order for $\mathcal{G}'$ to reflect the new graph. This algorithm proceeds as follows: first, it selects q from $\mathcal{G}_i$ and q_α from $\mathcal{G}_\alpha$ and creates the query $q' = \{x \mid \{\mathsf{teaches}(x, y), \mathsf{Student}(y)\}$ (by adding atoms of q_α to q) (line 3). Then, it proceeds to the child of q_α, (i.e., to q'_α) and it creates the CQ $q'_1 = \{x \mid \{\mathsf{teaches}(x, y), \mathsf{teaches}(z, y)\}\}$ (by adding atoms of q'_α to q) (line 11). Moreover, it also adds the relation $\langle q', q'_1\rangle$ to $\mathcal{G}'$. Subsequently, a recursive call to joinGraphs is made with first two parameters q and q'_α. In this call, in line 4, $\mathsf{mergeCQs}(q'_\alpha, q)$ returns $\{z \mapsto x\}$, hence tuples $\langle q'_1, q\rangle$ and $\langle q, q_1\rangle$ are added to $\mathcal{G}'$, and $m(q) = m(q_1) = \{z \mapsto x\}$. (Note that $q_\sigma = q$ and $q_{1\sigma} = q_1$) Then, the algorithm returns from the recursive call and proceeds in line 13 to the child of q (i.e., q_1) but $\mathsf{canBeJoined}(q_1, m(q_1), jv)$ returns false for the reasons explained in Example 1 item 4. Hence, the algorithm terminates and we have $\mathcal{G}' = \langle u', \mathcal{H}', m'\rangle$ where $u' = \{q', q'_1, q, q_1\}$ (as defined in Example 1), $\mathcal{H}' = \{\langle q', q'_1\rangle, \langle q'_1, q\rangle\langle q, q_1\rangle\}$ and $m(q') = m(q'_1) = \emptyset$, $m(q) = m(q_1) = \{z \mapsto x\}$. ♢

5 Optimisations

5.1 Optimising the Last Iteration

As explained earlier, Algorithm 2 computes the cartesian product between two rewriting graphs. The structure of the computed graph is important while processing the atoms of the query, however, it is not important after processing the *last* atom of the input query. Consequently, in the last iteration, Algorithm 1 can call a simplified version of Algorithm 2 that constructs a set of CQs rather than a rewriting graph. Algorithm 3 depicts the simplified algorithm. Roughly speaking, it is obtained from Algorithm 2 by, removing the for-loop starting in line 13, computing for the last selected atom α a set u_α rather than a rewriting graph $\mathcal{G}_\alpha$, and adding the computed queries to a UCQ rather than a graph.

5.2 Optimising Redundancy Elimination

In line 15 Algorithm 1 applies the well-known redundancy elimination algorithm from [8]. As it has been shown by several experimental evaluations [8, 4], this method usually does not perform well in practice, because it consists of several loops over the (potentially large) set of computed CQs. In order to improve the

Algorithm 3 OptimisedExtensionStep($\mathcal{G}, u_\alpha, jv, av$)

Input: A rewriting graph $\mathcal{G} = \langle u, \mathcal{H}, m \rangle$, a UCQ u_α, and two sets of variables.

```
1: Initialise a queue Q with a top element in G
2: Initialise a UCQ U := ∅
3: while Q ≠ ∅ do
4:    Remove the head q of Q and let κ := m(q)
5:    if canBeJoined(q, κ, jv) then
6:       Add {av | q ∪ (q_α)_κ} to U
7:       for all q_α ∈ u_α do
8:          σ := mergeCQs(q_α, q)
9:          if σ ≠ ∅ then
10:            for all q' s.t. q ⇝_G q' do Add {av | q'}_σ to U
11:         else Add each q' such that ⟨q, q'⟩ ∈ G to Q
12: return U
```

performance of this method our algorithm uses the following two approaches. First, it tries to identify queries that, if added to the final rewriting, they are going to be redundant. Clearly, such queries need not be added, hence reducing the size of the set over which algorithm removeRedundant would be executed. Secondly, it also tries to identify queries that are going to be non-redundant. Such queries can then be excluded from the final check. Our algorithm identifies such queries as follows.

In the last iteration and before calling Algorithm 3 it executes the standard subsumption checking algorithm over $\mathcal{G}$ and stores all subsumption relations. Note that, the size of $\mathcal{G}$ at this point is expected to be significantly smaller than that of the final UCQ, hence the algorithm should behave well in practice. Then, when executing Algorithm 3 it identifies redundant queries as follows:

- In line 10, it adds a query $\{av \mid q'\}_\sigma$ to U *only if* for q the subsumer of q' (if it exists) $\{av \mid q\}$ is not already in U.
- Let q selected in line 4. If a subsumer q' of q exists such that, either $\{av \mid q'\}$ is already in U, or $q' \subseteq q$, $m(q') = m(q)$, and canBeJoined$(q', m(q'), jv) =$ true, then the algorithm 'skips' q—that is, it adds each q'' such that $\langle q, q'' \rangle \in \mathcal{G}$ to Q and it continues with the next CQ.

Also, Algorithm 3 is modified to identify non-redundant queries as follows:

- At the beginning it initialises an empty set NR of non-redundant queries.
- In line 10, if $\{av \mid q'\}_\sigma = \{av \mid q'\}$ and q' is non-redundant in u it adds $\{av \mid q'\}_\sigma$ to NR.
- In line 6, it adds $\{av \mid q \cup (q_\alpha)_{m(q)}\}$ to NR if none of the predicates in q_α appear in any CQ in u and if for each $q'_\alpha \in u_\alpha$ we have 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 removeRedundant as follows: All queries that are in the set NR are excluded from redundancy checking.

Table 1: Comparison between PerfectRef, Nyaya, Rapid, and versions of IQAROS

$\mathcal{O}$	Q	Size of UCQ						UCQ Computation Time						Overall Rewriting Time					
		PR	Nyaya	Rapid	Inc$_1$	Inc$_2$	Inc$_3$	PR	Nyaya	Rapid	Inc$_1$	Inc$_2$	Inc$_3$	PR	Nyaya	Rapid	Inc$_1$	Inc$_2$	Inc$_3$
P5	1	6	6	6	6	6	6	1	14	7	1	1	2	1	14	7	1	2	2
	2	11	10	10	10	10	10	15	128	14	7	3	3	17	128	15	8	4	4
	3	22	13	13	13	13	13	256	726	22	76	19	19	261	726	23	78	21	21
	4	45	15	15	15	15	15	1828	1889	33	288	173	166	1830	1889	36	291	178	169
	5	90	16	16	16	16	16	32255	16062	75	838	306	308	32270	16062	77	841	310	310
P5X	1	14	14	14	14	14	14	0	12	10	0	0	1	1	12	10	0	1	1
	2	86	66	25	81	25	25	2	130	23	2	3	1	13	170	26	6	4	3
	3	530	374	127	413	133	74	36	540	92	24	6	12	150	1415	135	95	19	36
	4	3476	2475	636	2070	670	393	656	1672	343	187	46	122	5876	3842	1181	313	283	445
	5	23744	17584	3180	10352	3352	2057	41454	15095	2061	828	214	371	326400	142580	5252	2817	1078	1233
S	1	6	6	6	6	6	6	0	15	6	0	0	0	0	15	6	0	0	0
	2	202	3	2	204	12	2	12	11	9	12	4	3	34	12	9	12	4	3
	3	1005	7	4	864	96	4	190	46	14	60	8	7	677	48	14	65	9	7
	4	1548	5	4	1428	84	4	254	34	14	104	9	6	889	34	14	116	10	7
	5	8693	13	8	6048	672	8	8216	159	36	1018	227	93	54252	163	37	1146	236	93
U	1	2	2	2	2	2	2	0	25	9	1	1	2	1	25	9	1	1	2
	2	189	1	1	190	5	1	24	7	19	12	3	4	32	7	19	12	3	4
	3	296	4	4	300	20	4	112	172	13	77	5	7	144	172	14	79	5	7
	4	1763	2	2	1688	45	2	826	15	17	253	8	10	1500	15	17	258	8	10
	5	3418	11	10	3375	90	10	2680	107	18	527	17	20	5083	108	19	582	18	20
UX	1	5	5	5	5	5	5	0	24	11	1	1	2	0	24	11	1	2	2
	2	286	1	1	287	7	1	14	6	13	10	4	4	31	6	13	11	4	4
	3	1248	12	12	1260	84	12	118	166	20	80	10	11	534	166	21	104	21	11
	4	5385	5	5	5137	129	5	829	15	17	201	11	13	6354	15	17	243	15	13
	5	9220	26	25	8955	225	25	2625	115	26	427	31	53	19622	120	30	593	67	53
A	1	402	248	27	357	77	77	24	1231	18	17	5	10	55	1304	18	18	11	14
	2	103	93	54	103	54	54	124	4928	43	39	12	16	127	4967	45	39	43	17
	3	104	105	104	104	104	104	656	35451	97	173	103	106	661	35491	97	177	328	107
	4	492	455	333	471	320	320	1237	17121	170	170	58	52	1297	17511	208	197	130	55
	5	624	-	624	624	624	624	355571	-	383	3412	258	620	355872		384	3667	491	637
AX	1	783	556	41	794	431	431	30	1282	26	18	4	10	135	1649	26	24	8	13
	2	1812	1738	1546	1812	1653	1545	141	4493	649	57	26	30	892	5588	1191	752	772	92
	3	4763	4742	4466	4763	4466	4466	707	34032	1694	186	48	125	8244	51352	2225	10018	8006	491
	4	7251	6565	4497	7229	6639	4479	1282	16569	1247	192	37	45	12782	36460	2785	4891	3579	304
	5	78885	-	32956	78885	74025	32960	319681	-	3810	4361	665	1276	-	-	60006	-	-	26770

Note that, some of the conditions above might seem rather strict. However, as shown by our experimental evaluation these are usually satisfied in practice and they can indeed be very effective. Moreover, their implementation overhead can be noticeable in some cases, however, their benefits in several difficult scenarios greatly outperforms it.

6 Evaluation

We have implemented Algorithms 1–3 in a prototype tool called IQAROS[3] and have compared it against PerfectRef [3], Nyaya [6], Requiem [8], and Rapid [4].[4] Regarding IQAROS we included three versions; the first one (Inc$_1$) implements Algorithms 1 and 2 without any optimisations; the second one (Inc$_2$) uses Algorithm 3 instead of Algorithm 2 when it adds the last atom of the query; the

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

[4] We were not able to obtain Presto as it is not publicly available. We also do not present Requiem due to space limitations and since Rapid outperforms it.

third one (Inc_3) refines Inc_2 by also implementing the various optimisations detailed in the previous section. For the evaluation we used the relatively standard framework proposed in [8], however, we did not include results for ontologies V and P1 since they are rather trivial for all systems. Experiments were conducted on a MacBook Pro with a 2.66GHz processor and 4GB of RAM, with a time-out of 600 seconds.

Table 1 presents the results for each system, where the columns annotated as "Size of UCQ" present the size of the computed UCQ before the final redundancy elimination (after redundancy elimination all systems return the same UCQ, as the ones reported in [8]), while the rest present the computation time before and after redundancy elimination (measured in milliseconds).

First we compare the different versions of IQAROS. We observe that the size of the computed UCQs decreases from Inc_1 to Inc_3. The difference between Inc_1 and Inc_2 is justified by the fact that the latter uses the simpler algorithm (Algorithm 3) which does not compute the Cartesian product between the graphs. The benefits of using this algorithm are also reflected in the computation times of Inc_2 compared to Inc_1. Inc_3 computes the smallest UCQ of the three versions due to its techniques for eliminating redundant queries. Regarding execution time Inc_3 performs similarly to Inc_2 and sometimes slightly worse, due to the overhead of implementing the various optimisations. However, when considering the total time the benefits of the optimisations become apparent. Inc_3 is significantly faster in ontologies A and AX and is actually the only configuration of IQAROS that can process query 5 in AX in only 27 seconds. This is heavily due to the optimisation of tracking non-redundant queries.

Compared to PerfectRef, and Nyaya, all versions of IQAROS (even Inc_1) are much faster, in some cases even for several orders of a magnitude. Moreover, Inc_2 and Inc_3 compute significantly smaller UCQs. Since in their core all these systems are based on the same approach for materialising knowledge from $\mathcal{T}$, we concluded that this improvement is due to the incremental rewriting strategy that provides a much more guided and localised strategy compared to the blind brute-force application of the reformulation and reduction steps. Also Nyaya supports n-arry predicates and its factorisation step is significantly more involved that our merge function.

Compared to Rapid, Inc_3 (the fastest of the three configurations) computes similarly small UCQs with some small exceptions (either against or in favor) in queries 3–5 in ontology P5X, in queries 1 and 3 in ontology A and in queries 1, 2, 4 and 5 in ontology AX. Moreover, Rapid is notably faster[5] 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 milliseconds. In all the other cases 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 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 optimisations used in Inc_3.

[5] We consider a system to be 'notably faster' if it is faster for more than 20 milliseconds.

7 Conclusion

In the current paper we presented a novel algorithm for query rewriting over DL-Lite$_R$ ontologies. The algorithm is based on a novel approach that processes each atom separately and then combines the results to compute a final UCQ rewriting. It is significantly guided and our experimental evaluation showed that it is generally faster than all available systems known to us.

We feel that our techniques have several important practical and theoretical consequences and give opportunities for future work. First, we strongly feel that this approach can be used in other First-Order rewritable languages, like *Linear-Datalog*$^{\pm}$ [6], and there is strong evidence that the resulting system would exhibit good performance. Even in non-First-Order rewritable languages one could perhaps still exploit parts of this technique to increase the efficiency of the rewriting algorithms. Moreover, our results show that the rewriting process (at-least for DL-Lite) can largely be performed in parallel and such techniques can be further investigated.

Acknowledgments Work supported by project EUscreen (ECP-2008-DILI-518002) within EU's eContentplus Programme. Giorgos Stoilos is supported by a Marie Curie FP7-Reintegration-Grants within European Union's Seventh Framework Programme (FP7/2007-2013) under REA grant agreement 303914.

References

1. Acciarri, A., Calvanese, D., Giacomo, G.D., Lembo, D., Lenzerini, M., Palmieri, M., Rosati, R.: Quonto: Querying ontologies. In: Proc. of AAAI-05. (2005)
2. Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M.: The DL-Lite family and relations. Journal of Artificial Intelligence Research 36, 1–69 (2009)
3. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The DL-Lite family. Journal of Automated Reasoning 39(3), 385–429 (2007)
4. Chortaras, A., Trivela, D., Stamou, G.: Optimized query rewriting in OWL 2 QL. In: Proc. of CADE-23 (2011)
5. Glimm, B., Horrocks, I., Lutz, C., Sattler, U.: Conjunctive query answering for the description logic $\mathcal{SHIQ}$. In: Proc. of IJCAI 2007 (2007)
6. Gottlob, G., Orsi, G., Pieris, A.: Ontological queries: Rewriting and optimization. In: Proc. of ICDE 2011 (2011)
7. Lutz, C.: The complexity of conjunctive query answering in expressive description logics. In: Proc. of IJCAR 08. pp. 179–193 (2008)
8. Pérez-Urbina, H., Horrocks, I., Motik, B.: Efficient Query Answering for OWL 2. In: Proc. of ISWC-09 (2009)
9. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. Journal on Data Semantics X, 133–173 (2008)
10. Rosati, R., Almatelli, A.: Improving query answering over DL-Lite ontologies. In: Proc. of KR-10 (2010)
11. Venetis, T., Stoilos, G., Stamou, G.: Query rewriting under query extensions for OWL 2 QL ontologies. In: Proc. of SSWS-11, Bonn, Germany (2011)