# Query Rewriting Under Query Extensions for OWL 2 QL Ontologies

Tassos Venetis[1], Giorgos Stoilos[2], and Giorgos Stamou[1]

[1] School of Electrical and Computer Engineering
National Technical University of Athens
Zographou Campus, 15780, Athens, Greece

[2] Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, UK

**Abstract.** Conjunctive query answering is a key reasoning service for many ontology-based applications. With the advent of lightweight ontology languages, such as OWL 2 QL, several query answering systems have been proposed which compute the so called *UCQ rewriting* of a given query. It is often the case in realistic scenarios, that users refine their original queries, by e.g., extending them with new constraints and making them more precise. To the best of our knowledge, in such cases, all OWL 2 QL systems would need to recompute the rewriting of the refined query from scratch. In this paper we study the problem of computing the rewriting of the refined query by 'extending' the pre-computed rewriting and avoiding re-computation. We study the problem from a theoretical point of view and present a practical algorithm. Finally, we evaluate our implementation experimentally by comparing it against many state-of-the-art query rewriting systems, obtaining encouraging results.

## 1 Introduction

A key application of OWL ontologies is ontology-based data access (OBDA) [15], where an ontology is used to support query answering against distributed and/or heterogeneous data sources. A typical scenario would involve the use of an OWL ontology to answer conjunctive queries over RDF datasets. Due to the high complexity of answering conjunctive queries over OWL 2 DL ontologies [11, 6], prominent languages such as OWL 2 QL have been developed. OWL 2 QL (a well-known OWL 2 profile[3]) is based on the well-known Description Logic (DL) DL-Lite$_R$ [3, 1]. DL-Lite$_R$ is a member of the DL-Lite family [3, 1], a family of 'lightweight' ontology languages specifically designed to feature low theoretical complexity, and hence imply the existence of efficient query answering algorithms.

Query answering in the DL-Lite family is usually performed via a technique called *query rewriting*. According to this technique, given a *query* and a DL-Lite ontology, the query is 'rewritten' into a set of queries such that, the union

[3] `http://www.w3.org/TR/owl2-profiles/`

of the answers of the queries in the set over the input data and by discarding the input ontology is equal to the answers of the original query over the data and the ontology. In recent years, query rewriting over DL-Lite ontologies has drawn significant attention, and several different algorithms and systems have been proposed [3, 13, 4, 16].

Quite often in realistic data-access scenarios, users do not immediately ask the query that they want. More precisely, as has been shown in the Web literature [9, 8, 12], users usually first ask some 'general' query and then, according to the results they get back, refine it by adding further constraints, making their request more specific each time. Consequently, the actual (final) query might only be known after several refinements of the initial one. In a different (Semantic Web) motivating scenario, in order to assist users in constructing their queries, iterative and incremental techniques have also been proposed [18, 5, 17]. However, to the best of our knowledge, none of the query rewriting approaches that have been proposed in the literature is designed to work well in such scenarios. More precisely, in such cases all algorithms will (re)compute the entire rewriting of each of the refined queries from scratch.

In the current paper we study the following problem: Given a DL-Lite$_R$ ontology, a query and its rewriting (computed previously), and a new constraint to be added to the query, compute the rewriting of the new query by 'extending' the input rewriting and avoid computing it from scratch through the standard algorithms. First, we study the problem at a theoretical level and investigate whether it is possible to compute the rewriting of the extended query given any input rewriting. Unfortunately, the answer is negative and we explain how optimisation techniques employed by nearly all modern systems might compute rewritings that are not suitable for our task. Then, we present an algorithm for computing query rewritings under query extensions. Our algorithm is based on the well-known PerfectRef algorithm [3], which in its original version did not include any optimisations and hence is suitable for our purposes. For our algorithm to behave well we propose several optimisations that are specific to our case. Finally, we have implemented the algorithm and have conducted an experimental evaluation using the evaluation framework proposed in [13]. We have compared our techniques with many of the available query rewriting systems and our first results are encouraging given our preliminary implementation.

Our problem is also highly relevant to the field of databases, where it has been studied under the term *view adaptation* [7, 10]—that is, computing the materialisation of a re-defined materialised view. However, view adaptation has not been studied in the presence of database constraints—that is, under the presence of logical axioms. We also feel that this new approach and view of query rewriting opens an interesting line of research for scalable OBDA.

## 2 Preliminaries

*Description Logics* We assume that the reader is familiar with the basics of DL syntax, semantics and standard reasoning problems [2]. We next recapitulate the syntax of DL-Lite$_R$ [3] a prominent DL language that consists of the logical underpinnings of the QL profile of OWL 2[3] and is widely used in ontology-based data access.

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 following grammar, where $A \in \mathbf{C}$ and $R$ is a DL-Lite$_R$-role:

$$B := A \mid \exists R$$

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.

*Queries* We use standard notions of (function-free) term and variable. 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:

$$\{\vec{x} \mid \{\alpha_1, \ldots, \alpha_m\}\}$$

where each $\alpha_i$ is a concept or role atom 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 $\alpha_i$. 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. Finally, a variable that is either distinguished or appears in at least two different atoms $\alpha_i, \alpha_j$ with $i \neq j$ in $q$ is called *bound*, otherwise it is called *unbound*. We often abuse notation and use $q$ to refer to the set of its atoms, i.e., $\{\alpha_1, \ldots, \alpha_m\}$. For the rest of the paper, and without loss of generality, we will asume that queries are *connected* [6]. Finally, a union of conjunctive queries (UCQ) is a set of conjunctive queries.

A certain answer to a CQ $q$ w.r.t. $\mathcal{O}$ is a tuple $\vec{c} = (c_1, \ldots, c_n)$ of individuals s.t. $\mathcal{O}$ entails the FOL formula obtained by building the conjunction of all atoms $\alpha_i$ 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. $\mathcal{O}$. Given $q_1, q_2$ with distinguished variables $\vec{x}$ and $\vec{y}$, we say that $q_2$ *subsumes* $q_1$, if there exists a substitution $\theta$ from the variables of $q_2$ to the variables of $q_1$ such that the set $[\{ans(\vec{y})\} \cup q_2]_\theta$ is a subset of the set $\{ans(\vec{x})\} \cup q_1$, where $ans$ is in each case a predicate of the same arity as $\vec{x}$ ($\vec{y}$) not appearing in $q_1$ ($q_2$). Finally, $q_1$ is *equivalent* to $q_2$ if they subsume each other.

*Query Answering in DL-Lite$_R$* Query answering in DL-Lite$_R$ is performed with a technique known as *query rewriting* which, given a DL-Lite$_R$-TBox $\mathcal{T}$ and query $q$, computes a UCQ $u$, called a *UCQ rewriting* for $q, \mathcal{T}$, with the following property: for each ABox $\mathcal{A}$ s.t. $\mathcal{O} = \mathcal{T} \cup \mathcal{A}$ is consistent, the following holds:

$$\mathsf{cert}(q, \mathcal{O}) = \bigcup_{q' \in u} \mathsf{cert}(q', \mathcal{A}).$$

For a DL-Lite$_R$-TBox, a UCQ rewriting $u$ for $q, \mathcal{T}$ can be computed using the perfect reformulation algorithm (PerfectRef) described in [3]. The algorithm applies exhaustively a *reformulation* and a *condensation* step that generate new CQs; the process terminates when no new CQ is generated.

In the reformulation step the algorithm picks a CQ $q$, an atom in the CQ $\alpha \in q$ and an axiom $I \in \mathcal{T}$ and *applies* the axiom on the atom $\alpha$ of $q$ replacing it with a new atom and hence, creating a new CQ. This is performed with the function $\mathsf{gr}(\alpha, I)$, that takes as input an atom $\alpha$ and an axiom $I \in \mathcal{T}$ and returns a new atom. For a CQ $q$, $\alpha \in q$ and $I \in \mathcal{T}$, $\mathsf{gr}(\alpha, I)$ is defined as follows:

- if $\alpha = A(x)$ and
  i) $I = B \sqsubseteq A$, then $\mathsf{gr}(\alpha, I) = B(x)$;
  ii) $I = \exists P \sqsubseteq A$, then $\mathsf{gr}(\alpha, I) = P(x, y)$ for $y$ a new variable in $q$;
  iii) $I = \exists P^- \sqsubseteq A$, then $\mathsf{gr}(\alpha, I) = P(y, x)$ for $y$ a new variable in $q$.
- if $g = P(x, z)$, $z$ is unbound in $q$ and
  i) $I = A \sqsubseteq \exists P$, then $\mathsf{gr}(\alpha, I) = A(x)$;
  ii) $I = \exists S \sqsubseteq \exists P$, then $\mathsf{gr}(\alpha, I) = S(x, y)$ for $y$ a new variable in $q$;
  iii) $I = \exists S^- \sqsubseteq \exists P$, then $\mathsf{gr}(\alpha, I) = S(y, x)$, for $y$ a new variable in $q$.
- if $g = P(z, x)$, with $z$ unbound, then
  i) $I = A \sqsubseteq \exists P^-$, then $\mathsf{gr}(\alpha, I) = A(x)$;
  ii) $I = \exists S \sqsubseteq \exists P^-$, then $\mathsf{gr}(\alpha, I) = S(x, y)$, where $y$ is new in $q$;
  iii) $I = \exists S^- \sqsubseteq \exists P^-$, then $\mathsf{gr}(\alpha, I) = S(y, x)$, where $y$ is new in $q$.
- if $g = P(x, y)$ and
  i) $I = S \sqsubseteq P$ or $I = S^- \sqsubseteq P^-$, then $\mathsf{gr}(\alpha, I) = S(x, y)$;
  ii) $I = S \sqsubseteq P^-$ or $I = S^- \sqsubseteq P$, then $\mathsf{gr}(\alpha, I) = S(y, x)$.

If for some axiom $I$ and some atom $\alpha$ one of the above conditions holds, then we say that $I$ is *applicable* to $\alpha$, and applying $I$ to some $\alpha$ in some CQ $q$ creates a new CQ of the form $q[\alpha/\mathsf{gr}(\alpha, I)]$—that is, the new CQ contains the atom $\mathsf{gr}(\alpha, I)$ instead of $\alpha$.

In the condensation step a new query is generated from a query $q$ by applying to $q$ the most general unifier between two atoms $\alpha_1, \alpha_2$ of its body; the application of the condensation step is denoted by $\mathsf{reduce}(q, \alpha_1, \alpha_2)$.

## 3 Query Rewriting Under Query Extensions

In this section we present the design and implementation of an algorithm for computing the UCQ rewriting of a refined query from the UCQ rewriting of the initial query by avoiding the computation of the UCQ rewriting of the refined query from scratch as in any of the standard query rewriting algorithms. In the current paper we focus on refinements that involve additions of new atoms to the queries, which we call *extensions*. We leave other types of refinements like deletion of atoms or changes in the set of distinguished variables for future work.

In the following, we first study the problem at a theoretical level and give examples that highlight issues and difficulties and which also explain several of its technical parts; then, we present the algorithm in detail.

### 3.1 Algorithm Design

*Example 1.* Consider the following TBox about an academic domain and the CQ which retrieves all individuals that are students:

$$\begin{array}{rcl} \mathcal{T} & = & \{\mathsf{GradStudent} \sqsubseteq \mathsf{Student}, \mathsf{TennisPlayer} \sqsubseteq \mathsf{Athlete}\} \\ q & = & \{x \mid \{\mathsf{Student}(x)\}\} \end{array}$$

The set $u = \{q, q_1\}$, where $q_1 = \{x \mid \{\mathsf{GradStudent}(x)\}\}$ is a UCQ rewriting for $q, \mathcal{T}$ and can be computed by any state-of-the-art query rewriting algorithm.

Suppose, now that a user wants to extend the initial query and retrieve only those students that are also athletes—that is, issue the new query $q' = \{x \mid \{\mathsf{Student}(x), \mathsf{Athlete}(x)\}\}$. It can be easily checked that the UCQ $u' = \{q', q'_1, q'_2, q'_3\}$, with $q'_i$ defined as follows, is a UCQ rewriting for $q', \mathcal{T}$:

$$\begin{array}{rcl} q'_1 & = & \{x \mid \{\mathsf{Student}(x), \mathsf{TennisPlayer}(x)\}\}, \\ q'_2 & = & \{x \mid \{\mathsf{GradStudent}(x), \mathsf{Athlete}(x)\}\}, \\ q'_3 & = & \{x \mid \{\mathsf{GradStudent}(x), \mathsf{TennisPlayer}(x)\}\} \end{array}$$

which can again be computed using any rewriting algorithm for $q', \mathcal{T}$. $\diamondsuit$

Although $u'$ from the above example can be computed by any state-of-the-art query rewriting algorithm and system, when such an algorithm is applied over $q'$ it will 'repeat' all the work previously done for the atom $\mathsf{Student}(x)$, when it computed the UCQ rewriting for query $q$. Our motivation is that since all the work for $q$ has already been done, perhaps it is possible to compute the UCQ rewriting of the extended query, by computing a UCQ rewriting only for the new atom (i.e., for the atom $\mathsf{Athlete}(x)$) and then, by appropriately 'combining' the two rewritings. Using this approach we will perform only the additional work left to compute a UCQ rewriting for the extended query, modulo the overhead for combining the rewritings which we anticipate to be small.

To design a correct algorithm several important technical issues need to be resolved. To mention a few, firstly, we need to figure out what is the appropriate CQ of the new atom for which a UCQ rewriting should be computed (especially regarding the choice of distinguished variable) while then, how to appropriately combine the two UCQ rewritings.

**Definition 1.** *Let $q$ be a CQ and $\alpha$ an atom containing at least a variable of $q$. The* atom-query *for $\alpha$ w.r.t. $q$ is the CQ defined as follows $q_\alpha := \{\mathsf{var}(\alpha) \cap \mathsf{var}(q) \mid \{\alpha\}\}$.*

The distinguished variables of the atom-query for $\alpha$ w.r.t. $q$ are all the variables of $\alpha$ that also appear in $q$. The intuition is that, in the extended query $(q \cup \{\alpha\})$ these variables are bound and hence, when computing the UCQ rewriting for $\alpha$ in isolation, these should be treated as such. In our previous example, the atom-query for $\alpha$ w.r.t. $q$ is the CQ $q_\alpha = \{x \mid \{\mathsf{Athlete}(x)\}\}$ and its UCQ rewriting is the UCQ $u_\alpha = \{q_\alpha, q'_\alpha\}$, where $q'_\alpha = \{x \mid \{\mathsf{TennisPlayer}(x)\}\}$.

Having computed a UCQ rewriting $u_\alpha$ for the atom-query, we can use the two UCQs to compute a UCQ rewriting for the extended query. At this point it is important to see how the two rewritings should be combined. The obvious choice is to take the pair-wise union of the CQs for which there is an overlap between their variables. More precisely, for $u$ and $u_\alpha$ two UCQ rewritings and for $q_1 \in u, q_2 \in u_\alpha$ such that $\mathsf{avar}(q_2) \subseteq \mathsf{var}(q_1)$, a CQ of the form $\{\mathsf{avar}(q) \mid q_1 \cup q_2\}$ can be constructed. Again, the intuition behind the condition on the variables is that there must exist 'join'-points between the queries that are unified. Indeed, one can construct the UCQ rewriting $u'$ for $q', \mathcal{T}$ from Example 1 by following this procedure: from queries $q_1 \in u$ and $q'_\alpha \in u_\alpha$ we can obtain the CQ $q'_3$, while from $q_1 \in u$ and $q_\alpha \in u_\alpha$ we can obtain the CQ $q'_2$. However, as the following example shows this operation between the UCQs is not enough to give a complete UCQ rewriting for the extended query.

*Example 2.* Consider the following TBox and CQ:

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

The set $u = \{q, q_1\}$, where $q_1 = \{x \mid \{A(x)\}\}$ is a UCQ rewriting for $q, \mathcal{T}$. Consider now the addition of the atom $\alpha = B(y)$. The atom-query for $\alpha$ w.r.t. $q$ is the query $q_\alpha = \{y \mid B(y)\}$ and the UCQ $u_\alpha = \{q_\alpha, q^1_\alpha, q^2_\alpha\}$, where $q^1_\alpha = \{y \mid \{S(z,y)\}\}$ and $q^2_\alpha = \{y \mid \{R(z,y)\}\}$, is a UCQ rewriting for $q_\alpha, \mathcal{T}$.

Using the procedure described above we can compute the UCQ $u_\cup := \{q \cup q_\alpha, q \cup q^1_\alpha, q \cup q^2_\alpha\}$, which is indeed a sound UCQ rewriting for the query $q^+ := q \cup \{\alpha\}$. However, it is not complete; more precisely, any complete UCQ rewriting for $q^+, \mathcal{T}$ must contain the query $\{x \mid \{A(x)\}\}$; but, for all CQs $q' \in u_\alpha$ $\mathsf{avar}(q') \not\subseteq \mathsf{var}(q_1)$, hence $q_1$ is never used. ◇

In the previous example we observe that the missing query $(q_1)$ does exist in the UCQ rewriting of the initial query, but it cannot be added to the target UCQ using the union operation. This suggests that there is probably another type of interaction between the UCQ rewritings that we should consider. More precisely, we observe that apart from points where the UCQ rewritings should be unified, there exist points where the UCQs should be 'merged'. For example, in the previous case we can observe that the formula implied by the CQ $q^2_\alpha \in u_\alpha$ is in some sense already 'contained in' $q \in u$. This represents a point where the two UCQ rewritings actually 'merge'. Hence, the construction of the UCQ rewriting of the extended query should proceed by copying $q$ and all the CQs that are generated in the UCQ of the initial query 'after' $q$. Thus, in our previous example, $q_1$ should be copied (as is) to the computed UCQ. This in turn implies that the rewriting algorithm used to compute the UCQ rewriting of the initial query should keep track of the dependencies between the generated queries. As we will show in the next section, the aforementioned union (join) and merge operations are the two operations used to compute a UCQ rewriting of the extended query.

Another important open question is whether the above process can be performed using any computed UCQ rewriting for the initial query. Unfortunately, as the following example shows, this is not always possible. The problem is that optimisation techniques like subsumption checking, employed by many modern state-of-the-art query rewriting systems, can prune queries that are not going to be redundant in UCQ rewritings of the extended query.

*Example 3.* Consider the following TBox and CQ:

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

The UCQ $u := \{q, q_1\}$, where $q_1 = \{x \mid \{A(x)\}\}$, is a UCQ rewriting for $q, \mathcal{T}$. However, $q_1$ subsumes $q$, hence $q$ can be removed and the UCQ $\{q_1\}$ is also a UCQ rewriting for $q, \mathcal{T}$. Most modern systems are likely to return the latter UCQ rewriting.

Now suppose that we extend the original query by adding the new atom $B(y)$. Then, the new query is of the form $q^+ = \{x \mid \{A(x), R(x,y), B(y)\}\}$ and its UCQ rewriting consists of the set $\{q^+\}$. Unfortunately, it is not possible to compute this UCQ rewriting from the UCQ $\{q_1\}$. Intuitively, the problem is that query $q_1$, which is used to prune $q$, is no longer generated in the UCQ rewriting of the extended query; hence, in that context $q$ is not redundant. A UCQ rewriting for $q^+, \mathcal{T}$ can, however, be generated from $u$ (the UCQ without the subsumed query removed) and a UCQ rewriting for $q_\alpha = \{y \mid \{B(y)\}\}$, which consists of the UCQ $\{q_\alpha\}$. More precisely, from the union of $q \in u$ and $q_\alpha \in u_\alpha$ one obtains the query $q^+$. ♢

The previous example suggests that we should use an algorithm that does not employ such optimisation techniques. One such algorithm is the original PerfectRef algorithm. However, the absence of optimisation techniques compromises the practicality of the approach. More precisely, as has been shown by experimental evaluations [14], systems that do not use optimisations tend to compute very large UCQ rewritings. Hence, performing a pair-wise union of two large UCQ rewritings can be impractical. However, our intuition is that on the one hand, the UCQ rewriting of the atom-query is going to be rather small, while on the other hand, the two UCQ rewritings would have many 'merge' and few 'join' points, as the following example shows.

*Example 4.* Consider the following TBox and CQ:

$$\mathcal{T} = \{A_n \sqsubseteq A_{n-1}, \ldots, A_2 \sqsubseteq A_1, A_1 \sqsubseteq B, A_1 \sqsubseteq C\} \quad q = \{x \mid \{B(x)\}\}.$$

The set $u = \{q, q_1, \ldots, q_n\}$ where $q_i$ is a CQ of the form $\{x \mid \{A_i(x)\}\}$ is a UCQ rewriting for $q, \mathcal{T}$. Now suppose that we extend the query with atom $C(x)$ obtaining the new CQ $q^+ = \{x \mid \{B(x), C(x)\}\}$. Following our previous discussion, we can compute a UCQ rewriting for $q^+, \mathcal{T}$ by combining $u$ with a UCQ rewriting for $q_\alpha = \{x \mid \{C(x)\}\}$ w.r.t. $\mathcal{T}$, which in this case is the UCQ $u_\alpha = \{q_\alpha, q_1, \ldots, q_n\}$. However, after computing the union of $q \in u$ and $q_\alpha \in u_\alpha$ we immediately see that $q_1$ appears in both UCQ rewritings. Hence, at this point the two UCQs merge and all queries $q_i$ with $1 \leq i \leq n$ can be copied to the final UCQ and can be discarded from further processing. $\diamondsuit$

Concluding our analysis and design, we present yet another technical issue in the construction of a correct algorithm.

*Example 5.* Consider the following TBox and CQ:

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

The set $u = \{q, q_1, q_2\}$, where $q_1 = \{x \mid \{R(x,y)\}\}$ and $q_2 = \{x \mid \{A(x)\}\}$ is a UCQ rewriting for $q, \mathcal{T}$. Consider now the addition of the atom $\alpha = B(z)$. The atom-query for $\alpha$ w.r.t. $q$ is the query $q_\alpha := \{z \mid \{B(z)\}\}$ and its UCQ rewriting is $u_\alpha := \{q_\alpha\}$. We can observe that the only query with which $q_\alpha$ joins is the query $q$, however, a UCQ rewriting for $q \cup \{\alpha\}$ must contain the queries $q_1' = \{x \mid \{R(x,y), B(x)\}\}$ and $q_2' = \{x \mid \{A(x), B(x)\}\}$. The issue is that query $q_1$ is produced from $q$ by unifying $R(x,y)$ and $R(z,y)$ through a condensation step, and $z$, the common variable, is renamed to $x$. $\diamondsuit$

The above example suggests that in order to be able to compute a UCQ rewriting of an extended query from the UCQ rewriting of an initial one, the algorithm used to create the UCQ rewriting of the initial one should keep track of variable renamings preformed during the condensation step. If this is the case, then in the previous example, $q_\alpha$ can be joined with $q_1$ and $q_2$ in order to produce queries $q_1'$ and $q_2'$.

### 3.2 The UCQ Extension Algorithm

As detailed in the previous section, in order to produce a correct UCQ rewriting for an extended query, first and foremost, the algorithm that is used to compute the UCQ rewriting of the initial query must, on the one hand keep track of the dependencies between the generated queries while on the other hand, keep track of variable changes in the condensation step.

These changes are detailed in Algorithm 1, which presents ex-PerfectRef, an extended version of the standard PerfectRef algorithm. Unlike PerfectRef, ex-PerfectRef maintains a binary-relation $G$ over queries. A pair $\langle q, q' \rangle$ is in $G$ if $q'$ is generated from $q$ by an application of either a single reformulation or condensation step. Since $G$ can contain cycles, ex-PerfectRef also extracts and returns a *hierarchy* out of the computed dependency relation—that is, for each cycle a representative query is selected and then a transitively-reduced strict partial order of all the representative elements is constructed. The formal definition of the hierarchy function is given next.

**The function** hierarchy. Let $U$ be a set, let $K \subseteq U \times U$ be a binary relation over $U$ and let $S$ be a subset of $U$.

**Algorithm 1** ex-PerfectRef$(q, \mathcal{T})$

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

Initialise a UCQ $u := \{q\}$
Initialise a binary relation $G := \emptyset$
Initialise a mapping $\mu$ from CQs to unifications and set $\mu(q) := \emptyset$
**repeat**
  $u' := u$
  **for all** $q \in u'$ **do**
    **for all** $\alpha \in q$ **do**
      **for all** PI $I \in \mathcal{T}$ **do**
        **if** $I$ is applicable to $\alpha$ **then**
          $q' := q[\alpha/\mathsf{gr}(\alpha, I)]$
          $\mu(q') := \mu(q)$
          $u := u \cup \{q'\}$;
          $G := G \cup \{\langle q, q' \rangle\}$
        **end if**
      **end for**
    **end for**
    **for all** $\alpha_1, \alpha_2$ in $q$ **do**
      **if** there is a most general unifier $\sigma$ for $\alpha_1$ and $\alpha_2$ **then**
        $q' := \mathsf{reduce}(q, \alpha_1, \alpha_2)$
        $\mu(q') := \mu(q) \cup \{\sigma\}$
        $u := u \cup \{q'\}$
        $G := G \cup \{\langle q, q' \rangle\}$
      **end if**
    **end for**
  **end for**
**until** $u' = u$
**if** $G = \emptyset$ **then**
  **return** $(\mathsf{hierarchy}(u, \{\langle q, \{\mathsf{var}(q) \mid \{\}\}\rangle\}), \mu)$
**end if**
**return** $(\mathsf{hierarchy}(u, G), \mu)$

- $D \in U$ is *reachable* in $K$ from $C \in U$, written $C \rightsquigarrow_K D$, if $E_0, \ldots, E_n$ with $n \geq 0$ exist where $E_0 = C$, $E_n = D$ and $\langle E_i, E_{i+1} \rangle \in K$ for each $0 \leq i < n$.[4]
- A *hierarchy* of $S$ w.r.t. $K$ is a pair $\langle \mathcal{H}, \rho \rangle$ defined as follows:
  - Let $V \subseteq S$ be a minimal (w.r.t. set inclusion) set such that, it contains exactly one element from each set $\{C \mid C, D \in S,\ C \rightsquigarrow_K D$ and $D \rightsquigarrow_K C\}$. Then, $\mathcal{H}$ is the reflexive–transitive reduction of the relation $\{\langle C, D \rangle \in V \times V \mid C \rightsquigarrow_K D\}$.
  - $\rho : V \to 2^S$ is the function on $V$ such that $D \in \rho(C)$ if and only if $C \rightsquigarrow_K D$ and $D \rightsquigarrow_K C$.
- $\mathsf{hierarchy}(S, K)$ is a function that returns one arbitrarily chosen but fixed hierarchy of $S$ w.r.t. $K$.

[4] Note that, according to this definition, each $C \in U$ is reachable from itself.

**Algorithm 2** ExtendRewriting$(q, \alpha, \mathcal{T}, \langle\mathcal{H}, \rho\rangle, \mu)$

**Input:** A CQ $q$, an atom $\alpha$, a DL-Lite$_R$-TBox $\mathcal{T}$ and a hierarchy $\langle\mathcal{H}, \rho\rangle$ and mapping $\mu$ computed using Algorithm 1.

$u_\alpha :=$ PerfectRef$(\{\mathsf{var}(\alpha) \cap \mathsf{var}(q) \mid \{\alpha\}\}, \mathcal{T})$
Initialise a queue $Q$ with $Q := \{q_0\}$, where $q_0$ is the root in $\mathcal{H}$
$u := \emptyset$
**while** $Q \neq \emptyset$ **do**
    Remove the head $q_H$ from $Q$
    **for all** $q_{eq} \in \rho(q_H)$ **do**
        **for all** $q_\alpha \in u_\alpha$ **do**
            **if** isContainedIn$(q_\alpha, q_{eq})$ **then**
                **for all** $q''$ such that $q_{eq} \rightsquigarrow_{\mathcal{H}} q''$ **do**
                    Add $q''$ and all CQs in $\rho(q'')$ to $u$
                **end for**
            **else**
                $\mu_{eq} := \mu(q_{eq})$
                **if** containsAllVars$(q_\alpha, q_{eq}, \mu_{eq})$ **then**
                    Add $\{\mathsf{avar}(q) \mid q_{eq} \cup (q_\alpha)_{\mu_{eq}}\}$ to $u$
                    Add to the end of $Q$ each $q''$ such that $\langle q_{eq}, q''\rangle \in \mathcal{H}$
                **end if**
            **end if**
        **end for**
    **end for**
**end while**
$u := u \setminus \{\{\mathsf{var}(q) \mid \{\}\}\}$
**return** removeSubsumed$(u)$

Finally, the algorithm uses a mapping $\mu$ from CQs to variable mappings in order to keep track of the variable unifications that are conducted during the condensation step (Line 20). These are also copied to newly created CQs in the reformulation step (Line 11).

Having computed a UCQ rewriting for some query $q$ and TBox $\mathcal{T}$ in the form of a hierarchy $\langle\mathcal{H}, \rho\rangle$ using Algorithm 1, and tracked variable unifications using $\mu$, one can compute a UCQ rewriting for any extension of query $q$ with an atom $\alpha$. The algorithm uses the following functions to check that the two UCQ rewritings should be merged or to check using the variable mappings in $\mu$ computed by Algorithm 1 that two CQs can be joined (cf. Example 5).

**The function** isContainedIn. Let $q, q'$ be two CQs. Then, isContainedIn$(q', q)$ returns true if the CQ $\{\mathsf{avar}(q) \mid q \cup q'\}$ subsumes $q$; otherwise it returns false. The intuition is that all atoms in $q'$ already exist in $q$.

**The function** containsAllVars. Let $q, q'$ be two CQs and $\mu$ a set of variable mappings. Then, containsAllVars$(q', q, \mu)$ returns true if, for each $z \in \mathsf{avar}(q')$ there exists $x \in \mathsf{var}(q)$ such that, when considering the mappings in $\mu$ as a graph, we have $z \rightsquigarrow_\mu x$.

Algorithm 2 presents the algorithm in detail. The algorithm accepts as an input a CQ $q$, a new atom $\alpha$, a DL-Lite$_R$-TBox $\mathcal{T}$ and a hierarchy $\langle \mathcal{H}, \rho \rangle$ and a mapping $\mu$ computed using Algorithm 1 and it returns a UCQ for the query $\{\mathsf{var}(q) \mid q \cup \{\alpha\}\}$. It first computes a UCQ rewriting $u_\alpha$ for the atom-query $q_\alpha$ of $\alpha$ w.r.t. $q$ (Line 1). The UCQ rewriting for $q_\alpha$ explicates all implied information of the $\mathcal{T}$ about the atom $\alpha$, which is essential for the correctness of the algorithm.

Having a UCQ rewriting for the initial query and the atom-query for $\alpha$ w.r.t. $q$, the algorithm proceeds in combining the UCQs; it uses a queue $Q$ to perform a breadth-first search over the queries in $\mathcal{H}$ and either compute the union of the queries or copy queries from the UCQ rewriting of the initial query. More precisely, it picks a query $q_H$ from $Q$, a query $q_{eq}$ in the equivalence class $\rho(q_H)$ and a query $q_\alpha$ from the UCQ $u_\alpha$. If $\mathsf{isContainedIn}(q_\alpha, q_{eq}) = \mathsf{true}$, then the two UCQs merge and hence, all queries $q''$ that are reachable in $\mathcal{H}$ from $q_{eq}$ and all those queries in the equivalence class of $q''$ can be added to the target UCQ (Lines 9–11). Otherwise, the algorithm checks if the two CQs can be unified using function containsAllVars. If the function returns true then the union of the queries is sound, and is thus added to the target UCQ after appropriately renaming the variables of $q_\alpha$ if necessary; then, the successor query of $q_{eq}$ in $\mathcal{H}$ is added to $Q$ and the process continues. Finally, the algorithm applies subsumption checking in order to remove all the redundant queries and return a minimal UCQ rewriting for the extended query.

It can be shown that Algorithm 2 correctly computes a UCQ rewriting for an extended query, given a hierarchy computed using Algorithm 1.

## 4 Evaluation

We have developed a prototype tool for computing the rewriting of an extended conjunctive query based on Algorithms 2 and 1. Our implementation uses the implementation of PerfectRef that was developed and used in the experimental evaluation in [14].

We have compared our implementations with a number of available query rewriting systems. More precisely, our set of tools include the aforementioned implementation of PerfectRef,[5] Requiem [14], a resolution-based rewriting algorithm that uses subsumption to reduce the number of generated queries and Rapid [4], a recently developed highly-optimised DL-Lite$_R$ UCQ rewriting algorithm. For the evaluation we used the framework proposed in [14]. It consists of nine ontologies, namely $V$ that captures information about European history,[6] P1 and P5 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[7] ontology and A that is an ontology capturing information about abilities, disabilities and devices. Moreover, we also used the ontologies P5X, UX and AX that consist of normalised versions of the ontologies P5, U

[5] `http://www.cs.ox.ac.uk/projects/requiem/C.zip`
[6] `http://www.vicodi.org/`
[7] `http://swat.cse.lehigh.edu/projects/lubm/`

**Table 1.** Comparison between PerfectRef and ex-PerfectRef

| $\mathcal{O}$ | $Q$ | PerfectRef | | ex-PerfectRef | | $\mathcal{O}$ | $Q$ | PerfectRef | | ex-PerfectRef | | $\mathcal{O}$ | $Q$ | PerfectRef | | ex-PerfectRef | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $\sharp u$ | $t$ | $\sharp u$ | $t$ | | | $\sharp u$ | $t$ | $\sharp u$ | $t$ | | | $\sharp u$ | $t$ | $\sharp u$ | $t$ |
| P1 | 1 | 2 | 1 | 2 | 4 | S | 1 | 6 | 2 | 6 | 4 | V | 1 | 15 | 4 | 15 | 6 |
| | 2 | 3 | 2 | 3 | 5 | | 2 | 202 | 175 | 202 | 158 | | 2 | 11 | 9 | 11 | 9 |
| | 3 | 7 | 9 | 7 | 9 | | 3 | 1005 | 1113 | 1005 | 1109 | | 3 | 72 | 43 | 72 | 38 |
| | 4 | 16 | 21 | 16 | 24 | | 4 | 1548 | 945 | 1548 | 1920 | | 4 | 185 | 82 | 185 | 121 |
| | 5 | 32 | 55 | 32 | 54 | | 5 | 8693 | 8589 | 8693 | 23521 | | 5 | 150 | 167 | 150 | 181 |
| P5 | 1 | 14 | 4 | 14 | 6 | U | 1 | 5 | 3 | 5 | 5 | A | 1 | 783 | 561 | 783 | 277 |
| | 2 | 86 | 32 | 86 | 40 | | 2 | 286 | 166 | 286 | 173 | | 2 | 1812 | 1217 | 1812 | 710 |
| | 3 | 530 | 273 | 530 | 304 | | 3 | 1248 | 479 | 1248 | 593 | | 3 | 4763 | 1506 | 4763 | 2074 |
| | 4 | 3476 | 1576 | 3476 | 2663 | | 4 | 5359 | 1628 | 5359 | 3919 | | 4 | 7251 | 2336 | 7251 | 5288 |
| | 5 | 23744 | 26498 | 23744 | 188456 | | 5 | 9220 | 4038 | 9220 | 14451 | | 5 | 7885 | 347798 | - | - |
| P5X | 1 | 14 | 3 | 14 | 5 | UX | 1 | 5 | 3 | 5 | 4 | AX | 1 | 783 | 335 | 738 | 269 |
| | 2 | 86 | 34 | 86 | 38 | | 2 | 286 | 187 | 286 | 154 | | 2 | 1812 | 1249 | 1812 | 713 |
| | 3 | 530 | 282 | 530 | 301 | | 3 | 1248 | 484 | 1248 | 587 | | 3 | 4763 | 1403 | 4763 | 2089 |
| | 4 | 3476 | 1452 | 3476 | 2660 | | 4 | 5358 | 1615 | 5358 | 3995 | | 4 | 7251 | 2717 | 7251 | 5407 |
| | 5 | 23744 | 33248 | 23744 | 195478 | | 5 | 9220 | 4041 | 9220 | 14513 | | 5 | 7885 | 356756 | - | - |

and A. For each ontology, a set of five hand-crafted queries is proposed [14]. All experiments were conducted on a MacBook Pro with a 2.66GHz processor and 4GB of RAM with a time-out of 600 seconds.

In our first experiment we compared our extended ex-PerfectRef (i.e., Algorithm 1) against the standard implementation of PerfectRef. The goal is to assess the extent to which our extensions and changes affect the performance of the original algorithm.

Table 1 presents the results, where $\sharp u$ and $t$ denote the size of the computed UCQ and the execution time (in milliseconds). As can be seen from that table, the performance of ex-PerfectRef is generally worse than that of PerfectRef. This was not surprising due to the extensions that have been applied to the original algorithm. This difference is relatively small in queries such as $Q1$–$Q4$, while it is usually more acute in query $Q5$. Notably, for query Q5 in ontologies A and AX, ex-PerfectRef failed to terminate within the set time-out.

In our second experiment, we evaluated Algorithms 1 and 2 against other query rewriting algorithms. In the case of our system, we proceeded as follows: for each of the test ontologies and for each of the queries $Qi$, $1 \leq i \leq 5$ we removed an arbitrary selected atom $\alpha$ to obtain a (hypothetical) initial query $Qi^{-}$. Then, we first run the method ex-PerfectRef$(Qi^{-}, \mathcal{T})$ to compute a UCQ rewriting for $Qi^{-}, \mathcal{T}$ in the form of a hierarchy $\langle \mathcal{H}, \rho \rangle$ together with the variable unification mappings $\mu$, and then run the method ExtendRewriting$(Qi^{-}, \alpha, \mathcal{T}, \langle \mathcal{H}, \rho \rangle, \mu)$ to compute the UCQ rewriting for $Qi, \mathcal{T}$, as detailed in Algorithm 2. Since this process requires a query that contained at least two atoms, we did not consider query $Q1$ for some ontologies.

Table 2 presents the results from our second experiment. In that table, ex-PR refers to algorithm ex-PerfectRef executed for $Qi^{-}$ and $\mathcal{T}$, Ref refers to Algorithm 2 without the final redundancy elimination step, while *sub* refers to that step (Line 23 of Algorithm 2). Hence, $\sharp u_{\star}$ and $t_{\star}$ denote the size of the computed UCQ and the execution time (in milliseconds) for the respective code $\star$. Also P-Ref refers to algorithm PerfectRef. Note that, after the final redundancy elimination

**Table 2.** Results of Algorithms 1 and 2 compared with other UCQ rewriting systems

| $\mathcal{O}$ | $Q$ | Algorithms 1 & 2 | | | | | | | P-Ref | Requiem | Rapid |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $\sharp u_{\text{ex-PR}}$ | $\sharp u_{\text{Ref}}$ | $t_{\text{ex-PR}}$ | $t_{\text{Ref}}$ | $t_{sub}$ | $t_{\text{Ref}}+t_{sub}$ | $t_{\text{all}}$ | | | |
| V | 2 | 1 | 10 | 3 | 35 | 3 | 38 | 41 | 14 | 15 | 44 |
| | 3 | 3 | 72 | 4 | 82 | 45 | 127 | 131 | 124 | 63 | 66 |
| | 4 | 37 | 185 | 30 | 39 | 95 | 134 | 164 | 274 | 173 | 116 |
| | 5 | 120 | 30 | 184 | 7 | 11 | 18 | 202 | 392 | 93 | 108 |
| P1 | 2 | 2 | 2 | 5 | 2 | 1 | 3 | 8 | 4 | 6 | 10 |
| | 3 | 3 | 2 | 5 | 3 | 0 | 3 | 8 | 9 | 11 | 12 |
| | 4 | 7 | 2 | 11 | 3 | 0 | 3 | 14 | 31 | 27 | 17 |
| | 5 | 16 | 2 | 30 | 4 | 1 | 5 | 35 | 102 | 69 | 37 |
| P5 | 2 | 14 | 10 | 5 | 5 | 2 | 7 | 12 | 40 | 26 | 15 |
| | 3 | 86 | 13 | 47 | 15 | 3 | 18 | 65 | 299 | 245 | 26 |
| | 4 | 530 | 18 | 322 | 35 | 1 | 36 | 358 | 1328 | 1131 | 39 |
| | 5 | 3476 | 32 | 2685 | 105 | 3 | 108 | 2793 | 26781 | 7722 | 104 |
| P5X | 2 | 14 | 25 | 7 | 6 | 7 | 13 | 20 | 134 | 152 | 30 |
| | 3 | 86 | 79 | 59 | 17 | 35 | 52 | 111 | 630 | 1380 | 161 |
| | 4 | 530 | 399 | 406 | 73 | 61 | 134 | 540 | 6327 | 4161 | 1230 |
| | 5 | 3476 | 2649 | 2911 | 225 | 770 | 995 | 3906 | 342133 | 93234 | 6533 |
| S | 2 | 34 | 29 | 19 | 11 | 1 | 12 | 31 | 400 | 180 | 12 |
| | 3 | 193 | 33 | 255 | 15 | 4 | 19 | 274 | 1514 | 1095 | 16 |
| | 4 | 404 | 57 | 464 | 20 | 5 | 25 | 489 | 1490 | 1142 | 15 |
| | 5 | 2296 | 154 | 2551 | 52 | 3 | 55 | 2606 | 28829 | 8202 | 18 |
| U | 1 | 24 | 2 | 10 | 4 | 0 | 4 | 14 | 4 | 11 | 9 |
| | 2 | 41 | 18 | 19 | 10 | 0 | 10 | 29 | 427 | 158 | 9 |
| | 3 | 180 | 8 | 161 | 11 | 1 | 12 | 173 | 650 | 256 | 11 |
| | 4 | 205 | 59 | 99 | 51 | 4 | 55 | 154 | 2133 | 1234 | 14 |
| | 5 | 225 | 59 | 179 | 19 | 6 | 25 | 204 | 6453 | 3307 | 18 |
| UX | 1 | 24 | 5 | 8 | 5 | 0 | 5 | 13 | 3 | 13 | 8 |
| | 2 | 41 | 20 | 18 | 10 | 1 | 11 | 29 | 471 | 212 | 8 |
| | 3 | 180 | 39 | 183 | 13 | 6 | 19 | 202 | 1169 | 778 | 20 |
| | 4 | 205 | 35 | 105 | 54 | 3 | 57 | 162 | 7677 | 6975 | 17 |
| | 5 | 225 | 113 | 215 | 19 | 30 | 49 | 264 | 20863 | 20466 | 33 |
| A | 1 | 27 | 52 | 14 | 80 | 8 | 88 | 102 | 676 | 181 | 20 |
| | 2 | 783 | 71 | 305 | 22 | 7 | 29 | 334 | 1184 | 162 | 42 |
| | 3 | 4763 | 104 | 2072 | 117 | 11 | 128 | 2200 | 1476 | 231 | 97 |
| | 4 | 783 | 323 | 313 | 95 | 80 | 175 | 488 | 2774 | 340 | 179 |
| | 5 | 4763 | 624 | 2141 | 262 | 195 | 457 | 2598 | 342897 | 576 | 316 |
| AX | 1 | 27 | 67 | 15 | 81 | 15 | 96 | 111 | 527 | 233 | 31 |
| | 2 | 783 | 1490 | 356 | 93 | 648 | 741 | 1097 | 2210 | 1561 | 1269 |
| | 3 | 4763 | 4752 | 11296 | 188 | 10428 | 10616 | 21921 | 9774 | 12097 | 2132 |
| | 4 | 783 | 3355 | 338 | 153 | 3451 | 3604 | 3942 | 16608 | 9140 | 2846 |
| | 5 | 4763 | 36013 | 11459 | 934 | - | - | - | - | - | 60492 |

step, all systems returned UCQ rewritings of the same size (the same as the ones reported in [14]) and hence the numbers are not presented.

As we can observe from the table, compared to PerfectRef, the process of extending the UCQ rewriting of a query (column $t_{\mathsf{Ref}}+t_{sub}$) is much more efficient than computing the UCQ rewriting of the new query from scratch (column for PerfectRef). Even more interestingly, even when considering Algorithms 1 and 2 together (i.e., $t_{\mathsf{all}}$), the process is much more efficient than PerfectRef. In cases of queries containing a few atoms (usually queries Q1 and Q2) and having small rewritings (less than 30 queries), the total time is comparable, however in queries with large UCQ rewritings and large number of atoms, Algorithms 1 and 2 combined, manage to be several times and sometimes even 1 or 2 orders of magnitude faster than PerfectRef in computing the UCQ rewriting for $Qi, \mathcal{T}$. Such notable cases are queries Q3–Q5 in ontology P5 and P5X, all the queries in ontology S, queries Q3-Q5 in ontology U and UX, queries Q1, Q2, Q4 and Q5 in ontology A and finally queries Q1, Q2, and Q4 in ontology AX. An intuition behind this large improvement is that the brute-force (blind) application of the reformulation and condensation steps of PerfectRef is bound to be inefficient and not scale well in such cases. In our case though, Algorithm 1 first computes a UCQ rewriting for a *smaller* CQ (i.e., $Qi$) and then Algorithm 2 performs a much more guided breadth-first search, applying simple operations like set-union.

However, there are also two exceptions. Firstly, PerfectRef is faster in query Q3 ontology A. The reason is that the UCQ rewriting of the 'reduced' query $Qi^-$ is much larger (4763 CQs) than the UCQ rewriting of $Qi$ (104 CQs). That is, the extra atom in $Qi$ helps PerfectRef stop computation earlier and compute the small target UCQ rewriting fast, while Algorithm 2 begins the refinement process with a large number of CQs most of which are not going to produce CQs for $Qi, \mathcal{T}$. Finally, like PerfectRef, Algorithm 2 failed to terminate in query Q5 ontology AX. The reason is that the size of the UCQ computed by the Ref part of the algorithm, i.e., $\sharp u_{\mathsf{Ref}}$, is quite large and the final redundancy elimination method fails to terminate within the set time-out.

Interestingly, a similar good behaviour for Algorithms 1 and 2 combined can be observed even when compared to the much more optimised system Requiem. There are a few cases that Requiem is more efficient, especially for ontology A which, as mentioned above, seems to be problematic for Algorithm 2, however, we can observe that in most cases the behaviour of Algorithms 1 and 2 is much more robust and scales better in queries with a UCQ rewriting of increasing size. Again, this is due to the guided nature of the refinement algorithm, while Requiem, although it uses subsumption internally to remove redundant queries, applies the resolution rule in an unguided brute-force way.

Finally, even when compared to Rapid, a highly optimised and DL-Lite$_R$-tuned algorithm, although Rapid is in most cases faster than the overall execution time of our strategy, there are several cases that the performance of the two algorithms is comparable. Actually, in ontology P5X and ontology A query 2, it manages to be notably faster than Rapid. Furthermore, when restricted only to the refinement step (Algorithm 2), algorithm manages to be even closer to the performance of Rapid.

## 5 Conclusion

In the current paper we studied the following problem: Given a query, a UCQ rewriting for the query and some atom, can we compute a UCQ rewriting for the query extended with the additional atom by "extending" the input UCQ rewriting without computing a UCQ rewriting of the new query from scratch?

We studied the problem at a theoretical level and investigated whether it is possible to compute such a UCQ rewriting from any given UCQ rewriting for the initial query. Our results showed that this is not possible in general, especially when optimisations are used to prune queries from the UCQ rewriting of the initial query. Hence, we designed our refinement algorithm by using the PerfectRef algorithm, which, in its original version, did not include any optimisations. Although it is commonly accepted that an unoptimised rewriting algorithm would compute large UCQ rewritings, and hence, compromise the practicality of our method, we continued by developing new optimisation strategies and a careful strategy for computing the refinement. Subsequently, we implemented the proposed algorithm and evaluated it experimentally, obtaining several encouraging and interesting results. On the one hand, the refinement process is much more efficient than computing the UCQ rewriting of the extended query from scratch using most (if not all) state-of-the-art rewriting algorithms. On the other hand, even when considering the overall time of computing the UCQ rewriting of the initial query together with the time for the refinement, the method was more efficient and robust compared to PerfectRef and Requiem, the latter of which also employs several optimisations.

There are many interesting challenges for future work. Firstly, one could study a similar problem under different types of query refinements, such as, after removing an atom or after adding and/or removing distinguished variables. Secondly, our initial relatively naive and preliminary algorithm is definitely open for further optimisations. More precisely, it is currently unknown whether some redundant queries from the UCQ rewriting of the initial query can actually be removed. Finally, investigating whether such an approach can also be applied to optimised systems such as Rapid or to more expressive DLs like $\mathcal{EL}$ and $\mathcal{ELHI}$ using systems such as Requiem are also interesting issues.

## References

1. Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M.: The DL-Lite family and relations. Journal of Artificial Intelligence Research 36, 1–69 (2009)
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)

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: Proceedings of the 23rd International Conference on Automated Deduction (CADE 23), Polland. pp. 192–206 (2011)
5. Demidova, E., Zhou, X., Nejdl, W.: A probabilistic scheme for keyword-based incremental query construction. IEEE Transactions on Knowledge and Data Engineering 99 (2011)
6. Glimm, B., Horrocks, I., Lutz, C., Sattler, U.: Conjunctive query answering for the description logic $\mathcal{SHIQ}$. In: Proc. of International Joint Conference on Artificial Intelligence (IJCAI 2007) (2007)
7. Gupta, A., Mumick, I.S., Ross, K.A.: Adapting materialized views after redefinitions. In: Proceedings of ACM SIGMOD International Conference on Management of Data. pp. 211–222 (1995)
8. Jansen, B.J., Spink, A., Blakely, C., Koshman, S.: Defining a session on web search engines: Research articles. Journal of the American Society for Information Science and Technology 58, 862–871 (2007)
9. Jansen, B.J., Spink, A., Pedersen, J.: A temporal comparison of altavista web searching: Research articles. Journal of the American Society for Information Science and Technology 56, 559–570 (2005)
10. Mohania, M.: Avoiding re-computation: View adaptation in data warehouses. In: In Proceedings of 8 th International Database Workshop. pp. 151–165 (1997)
11. Ortiz, M., Calvanese, D., Eiter, T.: Data complexity of query answering in expressive description logics via tableaux. Journal of Automated Reasoning 41(1), 61–98 (2008)
12. Pass, G., Chowdhury, A., Torgeson, C.: A picture of search. In: Proceedings of the 1st international conference on Scalable information systems (InfoScale 06). ACM (2006)
13. Pérez-Urbina, H., Motik, B., Horrocks, I.: Tractable query answering and rewriting under description logic constraints. Journal of Applied Logic 8, 186–209 (2009)
14. Pérez-Urbina, H., Horrocks, I., Motik, B.: Efficient query answering for OWL 2. In: Proceedings of the International Semantic Web Conference (ISWC 09). pp. 489–504 (2009)
15. 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)
16. Rosati, R., Almatelli, A.: Improving query answering over DL-Lite ontologies. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR-10) (2010)
17. Tran, T., Cimiano, P., Rudolph, S., Studer, R.: Ontology-based interpretation of keywords for semantic search. In: Proceedings of the International Semantic Web Conference (ISWC 2007). vol. 4825, pp. 523–536 (2007)
18. Zenz, G., Zhou, X., Minack, E., Siberski, W., Nejdl, W.: From keywords to semantic queries-incremental query construction on the semantic web. Journal of Web Semantics 7, 166–176 (2009)