Engineering optimisations in query rewriting for OBDA

Jose Mora
Ontology Engineering Group, Departamento de Inteligencia Artificial, Facultad de Informatica,Universidad Politécnica de Madrid
Madrid, Spain
jmora@fi.upm.es

Oscar Corcho
Ontology Engineering Group, Departamento de Inteligencia Artificial, Facultad de Informatica,Universidad Politécnica de Madrid
Madrid, Spain
ocorcho@fi.upm.es

ABSTRACT

Ontology-based data access (OBDA) systems use ontologies to provide views over relational databases. Most of these systems work with ontologies implemented in description logic families of reduced expressiveness, what allows applying efficient query rewriting techniques for query answering. In this paper we describe a set of optimisations that are applicable with one of the most expressive families used in this context ($\mathcal{ELHIO}^{\neg}$). Our resulting system exhibits a behaviour that is comparable to the one shown by systems that handle less expressive logics.

1. INTRODUCTION

Ontology Based Data Access (OBDA) consists on superimposing a conceptual layer as a view to an underlying information system, which abstracts away from how that information is maintained in the data layer and provides inference capabilities [6]. Expressiveness comes at a cost in terms of computational complexity, and thus OBDA has traditionally been an expensive feature only interesting when the challenges to tackle were on par with the cost of the solution. Several languages have been proposed as interesting compromise solutions between expressiveness and tractability for OBDA: the *DL-Lite* family [5], which includes *DL-Lite*$_{core}$, *DL-Lite*$_{\mathcal{F}}$ and *DL-Lite*$_{\mathcal{R}}$; the QL profile of $OWL2$ [8]; some families in Datalog± [4]; and the $\mathcal{ELHIO}^{\neg}$ family [14].

In this paper we focus on $\mathcal{ELHIO}^{\neg}$ and propose a series of optimisations that can be carried out during the rewriting process, which we have implemented in our system kyrie. Our evaluation shows that the efficiency results obtained are comparable to those obtained in previous approaches for less expressive logics.

This paper is structured as follows: In section 2 we provide some background on DL families and approaches for OBDA query rewriting. In section 3 we introduce the algorithms involved in our approach. In section 4 we describe our optimisations for the query rewriting process for $\mathcal{ELHIO}^{\neg}$. In section 5 we evaluate the results obtained. Finally, in section 6 we draw some conclusions from the work performed and outline some foreseen further improvements in the rewriting process.

2. BACKGROUND

Now we briefly describe the most relevant logics used for OBDA, and review implementations that use these logics. They are presented here in a chronological order.

2.1 OBDA-related logics

In this section we cite logics with special relevance in OBDA. In the examples a refers to a constant (individual), B_i refer to basic concepts (classes) and R refers to roles (properties).

- The *DL-Lite* family diverged into logics *DL-Lite*$_{\mathcal{R}}$ and *DL-Lite*$_{\mathcal{F}}$, both extending *DL-Lite*$_{core}$, where concept inclusions are restricted to $B_1 \sqsubseteq B_2$ and $B_1 \sqsubseteq \neg B_2$. *DL-Lite*$_{\mathcal{R}}$ includes subsumption (ISA) and disjointness assertions between roles and *DL-Lite*$_{\mathcal{F}}$ includes functionality restrictions on roles. These logics are first-order reducible with a tractable complexity [5].
- The $OWL2$ QL profile was inspired by the *DL-Lite* family and designed to keep the complexity of rewriting low, considering first-order rewritability. As a summary of a more extensive comparison [1], the main differences with *DL-Lite* are the lack of number restrictions, functionality constraints and keys. Among the constructs in OWL 2 not supported in *DL-Lite* we can remark nominals, concepts of the form $\{a\}$.

- The $\mathcal{ELHIO}^{\neg}$ logic [14] is more expressive; it extends the expressiveness of *DL-Lite*$_\mathcal{R}$ by including basic concepts of the form $\{a\}$, $\top$, and $B_1 \sqcap B_2$, as well as axioms of the form $\exists R.B_1 \sqsubseteq B_2$. This logic is the only one in this section that does not present the first-order rewritability property, this means that depending on the query and the expressiveness in the ontology, the generated Datalog may contain recursive predicates, thus some queries cannot be unfolded into a union of conjunctive queries (UCQ) and must be rewritten to recursive Datalog when considering $\mathcal{ELHIO}^{\neg}$ ontologies. Despite that the computational complexity of the rewriting process remains tractable (PTime-complete).
- Finally, some families in Datalog± preserve the property of first-order rewritability to SQL equivalent languages while offering a greater expressiveness for rewritings to SQL or non-recursive Datalog, mainly because of the fact that Datalog± predicates are n-ary. Some of the Datalog paradigms that ensure decidability are chase termination, guardedness or stickiness, extended to weak-stickiness by Calì et al. [3]. These paradigms limit the loops that can be present in some Datalog to ensure decidability of the unfolding and thus first-order rewritability.

2.2 Related OBDA query rewriting approaches

These logics have been used in several approaches, starting with the perfect reformulation [5], which is implemented in **Quonto** and by Pérez-Urbina [14]. This approach accepts ontologies written in the *DL-Lite* family (*DL-Lite*$_{core}$, *DL-Lite*$_\mathcal{F}$ and *DL-Lite*$_\mathcal{R}$) and generates a UCQ as a result of the rewriting process. This approach was the first of a series and would inspire many others, usually generating UCQs and optimizing the query rewriting process.

The **RQR** algorithm in the REQUIEM system [14] accepts $\mathcal{ELHIO}^{\neg}$ ontologies and generates a rewriting using resolution with free selection (RFS) [2]. Both RQR and RFS form the basis for the algorithm presented here. RFS is proven to be correct and complete on Horn clauses [11] however due to space limitations RFS will not be explained, we refer the reader to previous citations. RQR reduces the number of useless factorizations in RFS, queries generated and processing time through several optimizations, the main one being the introduction of Skolem functions when an existential quantification occurs in the head of a clause, which was handled in previous approaches as a nameless variable. The output generated by this approach is again a UCQ. Resolution in REQUIEM is splitted in two steps, the first one is saturation, which generates a (possibly recursive) Datalog program, and the second one is unfolding, which unfolds this Datalog program to generate a UCQ. REQUIEM may produce a Datalog program for the output by simply skipping the latter stage. If the ontology includes recursion the UCQ cannot be complete, in this case the unfolding generates a (recursive) linear Datalog program (at most one intensional predicate per clause).

The previous approaches generate a large number of queries in the UCQ, as the generation of this UCQ from Datalog presents a combinatorial blowup that depends on the length of the query. **Presto** [16] addresses this problem on *DL-Lite*$_\mathcal{R}$, this does not include expressions of the form $\exists R.B$, which eases the search for *most-general subsumees* (MGS). MGS are used to remove existential join variables, to then remove unbound variables and redundant atoms. This way the query is recursively factorized and splitted, depending on the existential joins and the connectivity in the query. Presto obtains results that are several orders of magnitude faster in the query rewriting process than previous approaches. Depending on the ontology and the query, the resulting non-recursive Datalog is also normally briefer in the number of clauses, hiding the combinatorial explosion that would result from unfolding. As a result of the factorization, several parts of the query are rewritten into (potentially equivalent) subqueries.

Stamou et al. [17] take a different approach in the handling of Skolem functions. This approach has evolved into **Rapid** [7], which handles an expressiveness that falls between REQUIEM and Presto: expressions of the form $\exists R.B$ are allowed in REQUIEM but not in Presto; and in the case of Rapid they are allowed in the right hand side but not in the left hand side of subsumption axioms. Rapid applies two rules alternatively, *query shrinking* and *query unfolding*. *Query shrinking* removes a bound variable by unifying it with a functional term. Skolem functions are internally handled in this resolution rule, so they do not appear after applying it. *Query unfolding* replaces a set of atoms with its unfoldings, preserving the terms in the atoms (no functional terms are used). This strategy generates less subsumed (redundant) queries and it is possible to restrict the search for subsumed queries among subsets of the queries generated. The output is equivalent to REQUIEM for ontologies in the expressiveness handled by Rapid.

A similar approach using two different resolution steps (factorization and rewriting) in a stratified strategy is the one implemented in **Nyaya** [10]. During the factorization step the query is compacted with unifications that preserve the query semantics, and in the rewriting step the query is unfolded. An optional step removes redundant atoms in the queries. In Nyaya, the expressiveness is greater than in previous cases by allowing the use of n-ary predicates. However there is no statement about which additional ontological axioms could be covered with this. The body of the clauses is restricted to those that have only one atom. With this it is possible to identify atoms in the body of a query that are implied by some other atom in the body, what means that they can be eliminated, reducing the size of the query, the UCQ and the required processing. This approach is specially tailored at reducing the size of the UCQ that are generated in the process. Depending on the query, this optimization may provide much smaller queries, in size, width or length, which are respectively, the number of queries in the UCQ, the number of joins to be performed and the number of atoms in the perfect rewriting as explained in the evaluation done for this approach.

Another approach that should be mentioned is the one taken by **Venetis** et al. [18], based on perfect reformulation. In this approach, users normally pose a succession of queries, refining an initial conjunctive query by adding or removing atoms. In these cases it is possible to use partial results from previous rewritings in the new rewritings.

Finally, **Prexto** [15] modifies Presto by considering extensional constraints, concept and role disjointness assertions and role functionality assertions. Disjointness and role functionality assertions are considered when construcing the Datalog program along with subsumption. In the unfolding stage these assertions are considered again, along with the extensional constraints. These considerations allow reducing the combinatorial explosion usual in the unfolding of these Datalog programs and the size of the rewritten UCQ.

2.3 Resolution with free selection

Resolution with free selection (RFS) has a special relevance for the approach taken in kyrie, thus we introduce here some necessary terminology to explain this algorithm and advise the reader to consult the references [2, 13] for a more detailed presentation. For comparison purposes we can consider first binary resolution as an example:

$$\frac{C \vee A \qquad D \vee \neg B}{(C \vee D)\mu}$$

Where μ is the most general unifier (MGU) of atomic formulas A and B. We can see in this rule that two atoms are unified, one on each clause. A selection function selects atoms in those clauses, so that for two atoms to be unified they have to be selected by that selection function. A selection function for Horn clauses selects either the head of a clause or a non empty set of body atoms based on some criteria. This allows to prioritize the inferences in which the selected atoms are selected. Resolution with free selection for Horn clauses takes the form:

$$\frac{A \leftarrow B_1 \wedge \ldots \wedge (\underline{B_i}) \wedge \ldots \wedge B_n \qquad \underline{C} \leftarrow D_1 \wedge \ldots \wedge D_n}{(A \leftarrow B_1 \wedge \ldots \wedge (D_1 \wedge \ldots \wedge D_n) \wedge \ldots \wedge B_n)\mu}$$

Where μ is the MGU of atomic formulas B_i and C and the underlined atoms are the ones selected and resolved. The clause for which a body atom is selected is considered the main premise and the clause for which the head is selected is considered the side premise.

Saturation in resolution means applying the available resolution rules until no new clauses can be obtained. Saturation in resolution with free selection has been proved to be correct and complete for Horn clauses [11]. In this paper we see that (1) some of these resolution rules can be applied before having any query, (2) that subsumed clauses can be deleted optimizing the process and (3) that resolution with free selection has still a degree of freedom that allows to introduce some heuristics optimizing the process further, along with (4) other optimizations for query rewriting that can be done at the implementation level of this resolution method.

3. THE OBDA ALGORITHM FOR KYRIE

We describe the algorithm that our system (kyrie) implements, highlighting the optimizations performed. The rewriting algorithm is based on saturating resolution with free selection.

The ontology is preprocessed (algorithm 1) to save time in the rewriting of the queries. This is done by saturating the ontology with the selection function from REQUIEM (named `sfRQR` here), the result of this saturation is a logic program which contains functional terms. This saturation served in RQR to remove all clauses that contained functional terms, in this case those clauses have to be preserved until the query is available. `sfRQR` works properly for some specific types of clauses, thus auxiliary predicates are introduced to conform to these types of clauses. After applying `sfAux` non-recursive auxiliary predicates can be removed (line 2), these are the predicates selected by `sfAux` and the clauses that contain these predicates are the ones selected to be pruned after this saturation. Both saturation steps are explained further in section 4.1. Finally, usual subsumption checks (atoms and clauses) are performed.

The main algorithm performs a reachability test to remove the clauses in the preprocessed ontology that are not reachable by the query and a series of saturation steps with algorithm 3 and the previously explained selection functions. If the working mode is `Datalog` then the Datalog program is returned after another reachability test. If the working mode is `UCQ` then the unfolding is attempted for the predicates selected by `sfNonRec`, which are all predicates except one for each loop. For a loop we refer to a list of clauses $\gamma_1, \ldots, \gamma_n$ such that for every pair (γ_i, γ_{i+1}) there is some auxiliary predicate p_i such that $p_i \in body(\gamma_i), p \in head(\gamma_{i+1})$, and there is some p_n such that $p_n \in body(\gamma_n), p_n \in head(\gamma_1)$. A heuristic is applied to minimize the number of excluded predicates, those that appear on more loops are chosen first, breaking several loops with one predicate. At this stage of the process (no functional terms) and given the expressiveness handled (table 1 in [14]) a loop must contain unary and binary predicates to introduce new individuals (through variables in the body and not in the head of some clause). Only these loops are open (infinite) and produce the exclusion of some predicates from the unfolding.

Finally, the saturation algorithm (algorithm 3) uses subsumption checks to limit the explosion produced by blind resolution on all combinations of clauses. Two subsumption check steps are performed for each query before any other processing is done, as we see in line 9 and later loop. First `condensate` removes subsuming atoms in new clauses. Then the following loop removes subsumed clauses from the program. Subsumed clauses can be safely removed as soon as they are generated, this avoids the generation of other subsumed clauses and limits the explosion in the resolution. Please note that we use $a \succeq_s b$ to indicate that a subsumes b.

We will analyse specific parts of these algorithms in the remainder of the paper. Algorithm 1 (preprocessing) is explained in section 4.1. The subsumption checks (line 1 in algorithm 2 and loop in line 9 in algorithm 3) are the optimisations explained in section 4.2. In section 4.3 we see the benefits of using first the shortest clauses as in line 5 in algorithm 3. Finally, in section 4.4 we consider the cases in which the condition for line 3 in algoritmh 3 does not hold. This allows separating clauses in two sets, side premises and main premises, with all new resolved clauses as main premises.

Algorithm 1: kyrie preprocess algorithm

```
Input: $\mathcal{ELHIO}^{\neg}$ ontology $\Sigma$
Output: Preprocessed ontology $\Sigma$
1  $\Sigma = saturate(\mathtt{p}, sfRQR, \Sigma, \emptyset)$
2  $\Sigma = saturate(\mathtt{s}, sfAux, \Sigma, \emptyset)$
3  $\Sigma = removeSubsumed(condensate(\Sigma))$
4  return $\Sigma$
```

Algorithm 2: General kyrie algorithm

```
Input: Preprocessed $\mathcal{ELHIO}^{\neg}$ ontology $\Sigma$, UCQ $q$,
       working mode $mode \in \{\mathtt{Datalog}, \mathtt{UCQ}\}$
Output: Rewritten query $q_\Sigma$
1  $q = removeSubsumed(condensate(q))$
2  $\Sigma_q = reachable(\Sigma, q)$
3  $q_\Sigma = saturate(\mathtt{s}, sfRQR, q, \Sigma_q)$
4  $q_\Sigma = saturate(\mathtt{s}, sfAux, q_\Sigma, \emptyset)$
5  $q_\Sigma = reachable(q_\Sigma)$
6  if $mode = \mathtt{Datalog}$ then
7      return $q_\Sigma$
8  end
9  $\Sigma_q = \{q_i \in q_\Sigma \mid head(q_i) \neq head(q)\}$
10 $q_\Sigma = \{q_i \in q_\Sigma \mid head(q_i) = head(q)\}$
11 $q_\Sigma = saturate(\mathtt{u}, sfNonRec, q_\Sigma, \Sigma_q)$
12 return $q_\Sigma$
```

Algorithm 3: kyrie saturation algorithm `saturate`

```
Input: Working mode $mode$, selection function $sf$,
       Datalog program $q$, optional Datalog clauses $\Sigma$
Output: Datalog program $q_\Sigma$
1  $pending = \mathtt{new}\ SortedQueue(q, \mathtt{shortestFirst})$
2  $done = \mathtt{new}\ Queue()$
3  if $\Sigma = \emptyset$ then $\Sigma \equiv done$
4  while $\neg pending.isEmpty()$ do
5      $q_i = pending.pop()$
6      $done.push(q_i)$
7      forall the $q_j \in \Sigma$ do
8          $Q_{i,j} = resolve(q_i, q_j, selectionFunction)$
9          forall the $q_k \in Q_{i,j}$ do
10             $q_k = condensate(q_k)$
11             if $\forall q \in pending \cup done . q \not\preceq_s q_k$ then
12                 $done = \{q \in done \mid q_k \not\preceq_s q\}$
13                 $pending = \{q \in pending \mid q_k \not\preceq_s q\}$
14                 $pending.push(q_k)$
15             end
16         end
17     end
18 end
19 if $mode \neq \mathtt{u}$ then
20     $done = done \cup \Sigma$
21 end
22 if $mode \neq \mathtt{p}$ then
23     $done = prune(sf, done)$
24 end
25 return $done$
```

4. OPTIMISATIONS APPLIED IN KYRIE

We illustrate the optimizations described in this section with an ad-hoc ontology, Hospital.ttl[1].

4.1 Optimisation 1. Ontology preprocessing

Introduction: The preprocessing stage only depends on the ontologies used for rewriting, it is independent of the queries issued to the system. Among previous OBDA approaches, only Venetis [18] specifies some preprocessing.

Background: REQUIEM saturates RFS. In those inferences both premises may be clauses derived exclusively from the ontology. Since the query is not necessary for these inference steps, they can be performed before the query is available and only once for all queries.

Approach: We use RFS with `sfRQR` as in REQUIEM with all the optimizations included in algorithm 3. We must simply consider that the corresponding reasoning to remove the skolem functions cannot finish until the query is available, thus all clauses must still be preserved (working mode `p`).

This resolution needs auxiliary predicates to keep the classes of clauses in the types defined in table 1 in [14]. However, next time (line 3 in algorithm 2) this selection function will only be applied to clauses that will act as side premises, due to the separation between query clauses and ontology clauses done at that moment. Therefore, we can remove some auxiliar predicates, more precisely the only ones that need to be preserved are one for each loop according to the heuristic explained in section 3. Simply note that in this case we may have loops that introduce individuals with functions, without mixing unary and binary predicates.

Removing auxiliary predicates allows performing some additional inferences in the preprocessing and reducing the number of clauses that are generated and could be generated in later stages. This reduces the search space, the inferences performed and thus the time needed for the rewriting at the cost of some additional space needed to store some clauses.

Example: $\exists presents.Disease \sqsubseteq \exists suffers.Disease$ is an axiom that produces the clauses:

```
          AUX$4(x)  ←  Disease(y), presents(x,y)
     Disease(f12(x))  ←  AUX$4(x)
  suffers(x,f12(x))  ←  AUX$4(x)
```

Through resolution we can obtain the equivalent set:

```
     Disease(f12(x))  ←  Disease(y), presents(x,y)
  suffers(x,f12(x))  ←  Disease(y), presents(x,y)
```

Conclusion: Auxiliary predicates have no correspondence in the ABox and only serve as "proxies" in the backward chaining from the predicates in the head (`Disease`, `suffers`) to the predicates in the body (`Disease`, `presents`). Through saturation all these inferences are performed, which means that the auxiliary predicates are no longer needed and can be safely removed. This saturation is made possible by another resolution process before any query is available. Both processes reduce later inferences.

[1]`http://delicias.dia.fi.upm.es/~jmora/kyrie/evaluation/Hospital.ttl`

4.2 Query subsumption check

Introduction: Query subsumption checks consist in checking whether a part of the query being rewritten is subsumed by another. When this happens one of the two parts can be removed. Clause subsumption checks are usually performed by checking clauses by pairs, checking whether one subsumes the other and removing the subsumed clause. Clause condensation, is performed by checking subsumption between atoms in a clause. In this case the subsuming atom is removed, since they are grouped by conjunctions.

Background: previous approaches perform similar checks for atom subsumption, we can find `condensation` in REQUIEM, `DeleteRedundantAtoms` in Presto, the *shrinking* rule in Rapid and `factorization` and some optimizations in Nyaya.

With respect to the clause subsumption check, REQUIEM performs this in a separate stage after the resolution has finished by checking all clause pairs. When using the "F" mode, REQUIEM does also perform a "full" subsumption check [2]. This means that during resolution newly derived resolvents may be deleted if they are subsumed by old or processed clauses. However, this subsumption check is not performed the other way around wrt old and new clauses. Unlike REQUIEM, we perform the subsumption check in both directions. Rapid performs a similar subsumption check, but the generation of subsumed clauses is reduced and more controlled, what allows limiting the check to subsets of the generated clauses. This check is performed after each unfolding step. In the case of Nyaya, subsumed clauses are removed with the elimination step, which is optionally performed after every rewriting step. Presto produces a factorized Datalog where subsumption check becomes less tractable, and thus there is no clause subsumption check. Atom subsumption for atoms that share some variable is considered with the MGS.

Objectives: The hypotheses for the "full" subsumption check implemented in REQUIEM [2] are that (1) the certain answers obtained from the logic program remain unaltered despite removing subsubmed clauses independently on their provenance and (2) this subsumption check pays off in terms of efficiency despite the computational cost that it represents. We propose an optimization that consists in performing the subsumption check among all clauses generated as part of the resolution process (line 9 in algorithm 3). In our context the clever handling of the unfolding sets done in Rapid cannot be added in a straightforward way, since the Datalog used is not linear, which means that unfolding sets are not composed of atoms but of conjunctions of atoms. In our case, new clauses are generated using RFS. Every time that a new clause is generated, its subsumption is checked and subsumed clauses are removed immediately after their generation.

The rationale for the efficiency gain is that the avoidance of the inferences in which the subsumed clause would participate, which due to recursion avoids the generation of a whole tree of clauses. This pays off for the cost of subsumption checks in most cases and specially in most complex cases where there are many atoms in the body of the query and clauses in the ontology. Additionally, all later steps and stages have a lesser load and require less time, which includes this subsumption check, less clauses generated means less clauses to check for subsumption.

Verification: The first hypothesis is immediate considering the equivalence of programs after removing or adding subsumed clauses [12, 9]. We can see this has no impact on the certain answers: Given two clauses γ_1 and γ_2 such that γ_1 subsumes γ_2, by definition of subsumption $\gamma_1 \succeq_s \gamma_2 \Rightarrow \exists\mu.head(\mu\gamma_1) = head(\gamma_2) \land body(\mu\gamma_1) \subseteq body(\gamma_2)$ where μ is a unifier from the variables in γ_1 to the terms in γ_2, so we can see $\gamma_1 \models \gamma_2$. The certain answers for a given query q over a OBDA system composed of a TBox Σ over a database D, as $\langle\Sigma, D\rangle$, are defined as the set containing exactly every tuple α of constants in $\langle\Sigma, D\rangle$ such that $\Sigma \cup D \cup q \models Q_h(\alpha)$ where Q_h is the predicate in the head of q. With this definition we can see that by preserving satisfiability we preserve the set of certain answers for a given logic program and satisfiability is preserved in all regards, remember $\gamma_1 \models \gamma_2 \Rightarrow \{\alpha \mid \Gamma\cup\gamma_1\cup\gamma_2 \models Q_h(\alpha)\} = \{\alpha \mid \Gamma\cup\gamma_1 \models Q_h(\alpha)\}$ for any Γ and Q_h. The second hypothesis is verified in the evaluation section.

Example: Given the following resolution:

```
            Q(x)   ←  Patient(x), SickPerson(x)
   SickPerson(x)   ←  Patient(x)
          ∴ Q(x)   ←  Patient(x), Patient(x)
```

We have that the result can be condensed into the query clause `Q(x) ← Patient(x)`. This query subsumes the original query, which can be discarded, avoiding other resolution steps and the generation of other queries, for example a new query would be generated with the following resolution step:

```
            Q(x)   ←  Patient(x), SickPerson(x)
   SickPerson(x)   ←  Condition(y), suffers(x, y)
          ∴ Q(x)   ←  Patient(x), Condition(y),
                      suffers(x, y)
```

By avoiding the further use of this subsumed query clause, we prevent the generation of all the clauses that could be derived from it, recursively.

4.3 Prioritizing some inferences

Introduction: A resolution strategy may specify an absolute, partial or absent order in which the inferences should be performed. Resolution with free selection establishes a partial order with the selection function by means of the specification of the atoms that can participate in the resolution (selected atoms) and the omission of the remaining atoms. In this case we include some additional ordering criteria in a heuristical attempt to produce subsuming clauses earlier.

Background: Two of the state-of-the-art systems specify the order of inference steps during resolution: REQUIEM and Rapid. Selection functions in REQUIEM specify a partial ordering, while in Rapid clauses are selected more carefully:

- *shrinking* and *unfolding* rules are applied alternatively.
- shorter clauses are considered first for unification, since those are the ones that will more likely subsume others.

- the base clause for the inference is always a query clause, as in the optimized version of Nyaya.

Objective: In kyrie we add this optimisation over the classical RFS, as specified in algorithm 3 line 1. This optimisation can be included in any system where the resolution does not have any specific order. This optimisation is specially relevant when combined with the subsumption check, since it increases the likelihood for early production of subsuming clauses, and thus increasing the impact of the removal of subsumed clauses. If the subsumption check is not performed and subsumed clauses are eliminated after resolution then the order in which clauses are generated during resolution is irrelevant for any optimisation purposes.

Examples: Check the example in the previous section. By using and producing shorter clauses first we increase the probability of producing subsuming clauses and thus the impact of subsumption check is enhanced.

4.4 Constraining the searches

Introduction: There are two main searches for clauses within the resolution described so far. First, to apply a resolution rule the clauses that fulfill the roles of main premise and side premise must be found. Second, once a resolution step has been performed, according to section 4.2 subsumed and subsuming clauses are searched for each newly generated clause.

Background: During resolution we have two different types of clauses depending on their provenance: those coming from the ontology and those that have the query as an ancestor. These two groups are easily differentiable: any clause derived from the query will keep the query predicate Q_h as the head predicate. The role of these two groups of clauses during resolution is also clear when both are available: query clauses will act as main premises, because their head cannot be unified; ontology clauses will act as side premises, because all the inferences that could be done with ontology clauses as main premises are among ontology clauses, and thus could and were performed in the preprocessing.

This optimisation is applicable at an implementation level to most systems and sometimes it is tacitly applied. Depending on the order and specificity of the resolution strategy, different methods can be applied in the implementation, from storing two different lists of clauses instead of one to indexing the clauses depending on the head predicate.

Approach: The search for main and side premises can be restricted to these two sets of clauses that can be separated before starting resolution. After preprocessing, all generated clauses are again query clauses, thus subsumption check needs only be performed among query clauses, saving checks of ontology clauses with all other clauses.

Limitations: This cannot be the applied in unfolding with recursion. In the case of a recursive Datalog program, again a small set of recursive predicates are selected, as explained in section 4.1. Clauses with these predicates as head predicates are considered to be "subqueries", acting only as main premises and avoiding an infinite unfolding.

5. EVALUATION

We have performed an empirical evaluation comparing the results of REQUIEM, Presto, Rapid, Nyaya and kyrie. The full results for this evaluation are available online[2].

The evaluation has been performed on cold run, by restarting the application after every query, except for Presto. Each query has been run a minimum of five times per system and the results averaged. The hardware used in the evaluation is a Intel® Core™2 6300 @1.86GHz with 2GB of RAM, Windows® XP and Java™version 1.6.0_33. We have measured the times and number of clauses generated for the ontologies traditionally used in the state of the art along with the usual sets of five queries for each of the ontologies provided [14].

We have also extended some ontologies, the most complex ones and interesting in this evaluation, with additional axioms. This allows checking the impact that a little difference in expressiveness and only a few axioms can have on the results of query rewriting. The ontologies with these additional axioms are: "AXE", "AXEb", "P5XE" and "UXE".

For example we have expanded AX into AXE by adding a single axiom. $\exists AUX0^{-}.\exists AUX1 \sqsubseteq Quadriplegia$, even though $AUX0$ and $AUX1$ are not very descriptive names, this is an axiom that could fit in the semantics of the ontology, considering $AUX0$ is a subproperty of *isAffectedBy* and $AUX1$ is a subproperty of *isAssistedBy*. The impact of this single axiom is specially noticeable in the 5th query, where we can see that the rewriting time and clauses generated in the UCQ decrease significantly for kyrie.

Due to space limitations we limit the results included in the paper to those obtained with ontologies "A", "AX" and "AXE" in table 1. We refer the reader to the online results where it is possible to compare these systems with more ontologies and additional considerations as the time that preprocessing requires and the number of clauses in the preprocessed ontology. The size of the preprocessed ontologies remains at reasonable sizes, being P5XE the worst case with a factor of $5.\overline{851}$ and three cases with a factor of 1.

The results we obtained are mixed. Runtimes for kyrie are in many cases comparable with Rapid, which is the fastest algorithm. Times are even shorter than Rapid in some cases for the generation of the Datalog rewriting. The results in the number of clauses generated for the UCQ are the same for the ontologies that fall in the intersection of the different expressivenesses that the different systems handle and the main difference relies on the time that is needed to generate these rewritings.

However, with the most expressive ontologies we can see the difference in the size of the rewritings. When comparing REQUIEM and kyrie we can see they generate a different number of clauses in the UCQ when the Datalog generated is recursive and the unfolding can only be partial. In this case kyrie aims at reducing the number of different predicates that can be found in the head of some clause, which can be considered as "unfolding as much as possible". REQUIEM

[2]`http://delicias.dia.fi.upm.es/~jmora/kyrie/evaluation/`

		REQUIEM (F mode)				Presto				Rapid (D&F modes)				Nyaya		kyrie			
		Datalog		UCQ		Datalog		UCQ		Datalog		UCQ		Only UCQ		Datalog		UCQ	
$\mathcal{O}$	q	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size
	1	31	87	278	27	18	54	118	402	12	54	12	27	1417	247	15	42	53	27
	2	37	76	90	50	19	33	28	103	9	33	31	50	12268	92	12	31	40	50
A	3	50	77	128	104	25	33	47	104	9	33	53	104	86245	104	28	32	72	104
	4	40	79	443	224	18	44	122	492	24	60	81	224	47285	454	25	39	106	224
	5	75	77	1062	624	28	38	471	624	12	38	140	624	1578491	624	43	37	512	624
	1	47	127	743	41	37	69	353	782	18	69	21	41	1752	555	12	57	72	41
	2	47	116	4750	1431	37	51	2181	1781	15	51	234	1431	16838	1737	9	49	1512	1431
AX	3	62	126	44256	4466	62	57	30503	4752	15	57	693	4466	132846	4741	21	56	17599	4466
	4	59	119	63465	3159	42	69	34847	7100	21	85	596	3159	112717	6564	18	64	7149	3159
	5	81	126	10024743	39941	N/A	N/A	N/A	N/A	18	72	6137	32921	N/A	N/A	43	71	1012887	32921
	1	50	130	787	41	40	69	350	782	15	69	21	41	N/A	N/A	15	57	68	41
	2	47	119	4937	1431	37	51	2115	1781	18	51	228	1431	N/A	N/A	15	49	1490	1431
AXE	3	62	129	43843	4466	72	57	30631	4752	25	57	693	4466	N/A	N/A	24	56	17378	4466
	4	59	122	62403	3159	40	69	34840	7100	22	85	587	3159	N/A	N/A	25	64	6962	3159
	5	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	15	72	5996	32921	N/A	N/A	109	103	7262	3159

Table 1: Extract of some evaluation results

chooses to unfold partially the Datalog available, generating linear Datalog, which means that each clause contains at most one intensional database predicate. When comparing Rapid and kyrie we can see that kyrie generates a different number of clauses in more cases than when compared with REQUIEM. This is due to the axioms included in the extended ontologies, which are ignored by Rapid since they fall out of the expressiveness handled by it. The difference in query five with ontology “AXE” is specially remarkable: in this case kyrie generates a UCQ that is about 10% the size of the Rapid UCQ by including one single axiom, as explained before.

The optimizations done are specially evident in the time that the generation of the UCQ takes when compared with REQUIEM, however, they are more significant when comparing the Datalog results. REQUIEM performs an optimization stage that removes subsumed clauses after the UCQ has been generated, but in the case of kyrie this is done along with resolution (section 4.2), therefore subsumed clauses have already been removed from this Datalog program. The time needed to generate the Datalog rewriting for some queries is on par with Rapid, being lesser or greater depending on the case. The same can be said about the size of this Datalog rewriting.

6. CONCLUSIONS

In this paper we have focused on the optimizations that can be done on a query rewriting process based on RFS (as it is the case of REQUIEM), which allows handling expressive logics. We do not add additional constraints to the problem or reduce the expressiveness that is handled, what could ease the process from a computational point of view. Instead we focus on implementational details that can be extrapolated to other approaches, as already pointed out, and that have a strong impact on the efficiency of the process, as can be seen in the evaluation performed.

We have shown that a significant increase in the efficiency for query rewriting can be obtained with these optimisations. The optimisations performed impose no additional restrictions on the properties of the input, either on expressiveness of the ontology, shape of the queries or additional data to be used in the process. The evaluation has been performed on the usual ontologies and queries without adding any restriction or property to the input. This means that the optimisations presented can be performed along with other optimisationss that may require this kind of assumptions on the input. Moreover, the impact of the optimisations performed in this paper may serve to judge and to put into context the impact of other optimisations that may impose restrictions on the input or require additional information.

Finally, the difference between the results obtained as Datalog and those obtained as a UCQ is astounding in most of the cases. Obviously a UCQ is simpler to manage than a Datalog program by any underlying system.On the contrary Datalog provides a more compact representation, which may be more convenient for systems that can handle this expressiveness. It would be interesting to have some statistical data about the expressiveness that underlying systems can handle and measurements about the efficiency in query answering UCQs and Datalog programs. Without these data the relative relevance of the results for the rewriting to Datalog and the rewriting to UCQ is left at the discretion of the reader.

Acknowledgements. We would like to kindly thank authors of previous systems (1) Rapid, (2) Presto and (3) Nyaya for their help in the usage of their systems and respectively (1) publishing the code, (2) sharing the binaries and (3) sharing the code. Special thanks to Héctor Pérez-Urbina for releasing the original REQUIEM code.

The work presented in this paper has been funded by an PIF grant (Personal Investigador en Formación) from the UPM (Universidad Politécnica de Madrid) (RR01/2008) and by the Spanish national project myBigData (TIN2010-17060).

7. REFERENCES

[1] A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyaschev. The DL-Lite family and relations. *J. Artif. Int. Res.*, 36(1):1–69, 2009.

[2] L. Bachmair and H. Ganzinger. Resolution theorem proving. *Handbook of automated reasoning*, 1:19–99, 2001.

[3] A. Calì, G. Gottlob, and A. Pieris. Query answering under non-guarded rules in datalog+/-. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, P. Hitzler, and T. Lukasiewicz, editors, *Web Reasoning and Rule Systems*, volume 6333, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[4] A. Calì, G. Gottlob, and A. Pieris. New expressive languages for ontological query answering. In W. Burgard and D. Roth, editors, *AAAI*. AAAI Press, 2011.

[5] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *Journal of Automated Reasoning*, 39(3):385–429, Oct. 2007.

[6] D. Calvanese, D. Lembo, M. Lenzerini, A. Poggi, and R. Rosati. Ontology-based database access. Technical report, CiteSeerX, 2007.

[7] A. Chortaras, D. Trivela, and G. Stamou. Optimized query rewriting for OWL 2 QL. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, volume 6803, pages 192–206. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[8] B. Cuenca Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel Schneider, and U. Sattler. OWL 2: The next step for OWL. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(4):309–322, Nov. 2008.

[9] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing*, number 3569 in Lecture Notes in Computer Science, pages 61–75. Springer Berlin Heidelberg, Jan. 2005.

[10] G. Gottlob, G. Orsi, and A. Pieris. Ontological queries: Rewriting and optimization (extended version). *arXiv:1112.0343*, Dec. 2011.

[11] C. Lynch. Oriented equational logic programming is complete. *Journal of Symbolic Computation*, 23(1):23–45, Jan. 1997.

[12] M. J. Maher. Equivalences of logic programs. In E. Shapiro, editor, *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 410–424. Springer Berlin Heidelberg, Jan. 1986.

[13] H. Pérez-Urbina. Tractable query answering for description logics via query rewriting. *A thesis submitted for the degree of Doctor of Philosophy*, 2009.

[14] H. Pérez-Urbina, I. Horrocks, and B. Motik. Efficient query answering for OWL 2. In *The Semantic Web - ISWC 2009*, volume 5823 of *Lecture Notes in Computer Science*, pages 489–504. Springer, 2009.

[15] R. Rosati. Prexto: Query rewriting under extensional constraints in DL-Lite. In E. Simperl, P. Cimiano, A. Polleres, O. Corcho, and V. Presutti, editors, *The Semantic Web: Research and Applications*, volume 7295 of *Lecture Notes in Computer Science*, pages 360–374. Springer Berlin / Heidelberg, 2012.

[16] R. Rosati and A. Almatelli. Improving query answering over DL-Lite ontologies. In F. Lin, U. Sattler, and M. Truszczynski, editors, *Proceedings of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning*. AAAI Press, 2010.

[17] G. Stamou, D. Trivela, and A. Chortaras. Progressive semantic query answering. In *The 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, page 112, 2010.

[18] T. Venetis, G. Stoilos, and G. Stamou. Query rewriting under query extensions for OWL 2 QL ontologies. In *The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)*, page 59, 2011.