Exploiting Conjunctive Queries in Description Logic Programs*

Thomas Eiter[1], Giovambattista Ianni[1,2], Thomas Krennwallner[1], and Roman Schindlauer[1,2]

[1] Institut für Informationssysteme 184/3, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
[2] Dipartimento di Matematica, Università della Calabria,
I-87036 Rende (CS), Italy.
{eiter,ianni,tkren,roman}@kr.tuwien.ac.at

Abstract. We present cq-programs, which enhance nonmonotonic description logics (dl-) programs by conjunctive queries (CQ) and union of conjunctive queries (UCQ) over Description Logics knowledge bases, as well as disjunctive rules. dl-programs had been proposed as a powerful formalism for integrating nonmonotonic logic programming and DL-engines on a clear semantic basis. The new cq-programs have two advantages: First, they offer increased expressivity by allowing general (U)CQs in the body. And second, this combination of rules and ontologies gives rise to strategies for optimizing calls to the DL-reasoner, by exploiting (U)CQ facilities of the DL-reasoner. To this end, we discuss some equivalences which can be exploited for program rewriting. Experimental results for a cq-program prototype show that this can lead to significant performance improvements.

1 Introduction

Rule formalisms that combine logic programming with other sources of knowledge, especially terminological knowledge expressed in Description Logics (DLs), have gained increasing interest in the past years. This process was mainly fostered by current efforts in the Semantic Web development of designing a suitable rules layer on top of the existing ontology layer. Such combinations of DLs and logic programming can be categorized in systems with (i) strict semantic integration and (ii) strict semantic separation, which amounts to coupling heterogeneous systems [1–4]. In this paper, we will concentrate on the latter, considering ontologies as external information with semantics treated independently from the logic program. Under this category falls [5, 2], which extends the answer-set semantics to so-called *dl-programs* (L, P), which consist of a DL part L and a rule part P that may query L. Such queries are facilitated by a special type of atoms, which also permit to enlarge L with facts imported from the logic program P, thus allowing for a bidirectional flow of information.

Since the semantics of logic programs is usually defined over a domain of explicit individuals, this approach may fail to derive certain consequences, which are implicitly contained in L. Consider a simplified version of an example from [6]:

* This work has been partially supported by the EC NoE REWERSE (IST 506779) and the Austrian Science Fund (FWF) project P17212-N04.

$L = \{father \sqsubseteq parent, \exists father.\exists father^{-}.\{Remus\}(Romulus), father(Cain, Adam),$
$father(Abel, Adam), hates(Cain, Abel), hates(Romulus, Remus)\},$
$P = \{BadChild(X) \leftarrow \mathrm{DL}[parent](X, Z), \mathrm{DL}[parent](Y, Z), \mathrm{DL}[hates](X, Y)\}.$

Apart from the explicit facts, L states that each *father* is also a *parent* and that Romulus and Remus have a common father. The single rule in P specifies that an individual hating a sibling is a *BadChild*. From this dl-program, $BadChild(Cain)$ can be concluded, but not $BadChild(Romulus)$, though it is implicitly stated that *Romulus* and *Remus* have the same father.

The reason is that, in a dl-program, variables must be instantiated over their Herbrand base (containing the individuals in L and P), and thus unnamed individuals like the father of Romulus and Remus, are not considered. In essence, dl-atoms only allow for building CQs that are *DL-safe* [6], which ensure that all variables in the query can be instantiated to named individuals. While this was mainly motivated by retaining decidability of the formalisms, unsafe CQs are admissible under certain conditions [1]. We thus pursue the following.

• We extend dl-programs by (U)CQs to L as first-class citizens in the language. In our example, to obtain the desired conclusion $BadChild(Romulus)$, we may use $P' = \{BadChild(X) \leftarrow \mathrm{DL}[parent(X, Z), parent(Y, Z), hates(X, Y)](X, Y)\}$, where the body of the rule is a CQ $\{parent(X, Z),\ parent(Y, Z),\ hates(X, Y)\}$ to L with distinguished variables X and Y.

Example 1. Both $r = BadParent(Y) \leftarrow \mathrm{DL}[parent](X, Y), \mathrm{DL}[hates](Y, X)$ and $r' = BadParent(Y) \leftarrow \mathrm{DL}[parent(X, Y), hates(Y, X)](X, Y)$ equivalently pick (some of) the bad parents. Here, in r the join between *parent* and *hates* is performed in the logic program, while in r' it is performed on the DL-side.

Since DL-reasoners including RACER, KAON2, and Pellet increasingly support answering CQs, this can be exploited to push joins between the rule part and the DL-reasoner, eliminating an inherent bottleneck in evaluating cq-programs.

• We present equivalence-preserving transformation rules, by which rule bodies and rules involving cq- or ucq-atoms can be rewritten.

• We report on some experiments with a prototype implementation of cq-programs using dlvhex and RACER. They show the effectiveness of the rewriting techniques, and that significant performance increases can be gained. These results are interesting in their own right, since they shed light on combining conjunctive query results from a DL-reasoner.

2 dl-Atoms with Conjunctive Queries

We assume familiarity with Description Logics (cf. [7]), in particular $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$.[3] A DL-KB L is a finite set of axioms in the respective DL. We denote logical consequence of an axiom α from L by $L \models \alpha$.

As in [5, 2], we assume a function-free first-order vocabulary Φ of nonempty finite sets $\mathcal{C}$ and $\mathcal{P}$ of constant resp. predicate symbols, and a set $\mathcal{X}$ of variables. As usual, a *classical literal* (or *literal*), l, is an atom a or a negated atom $\neg a$.

[3] We focus on these DLs because they underly OWL-Lite and OWL-DL. Conceptually, cq-programs can be defined for other DLs as well.

Syntax A *conjunctive query* (CQ) $q(\vec{X})$ is an expression $\{\vec{X} \mid Q_1(\vec{X}_1), Q_2(\vec{X}_2), \ldots, Q_n(\vec{X}_n)\}$, where each Q_i is a concept or role expression and each $\vec{X}_i$ is a singleton or pair of variables and individuals, and where $\vec{X} \subseteq \bigcup_{i=1}^{n} \text{vars}(\vec{X}_i)$ are its *distinguished* (or *output*) variables. A *union of conjunctive queries* (UCQ) $q(\vec{X})$ is a disjunction $\bigvee_{i=1}^{m} q_i(\vec{X})$ of CQs $q_i(\vec{X})$. Where it is clear from the context, we omit $\vec{X}$ from (U)CQs.

Example 2. In our opening example, $cq_1(X,Y) = \{parent(X,Z), parent(Y,Z), hates(X,Y)\}$ and $cq_2(X,Y) = \{father(X,Y), father(Y,Z)\}$ are CQs with distinguished variables X, Y, and $ucq(X,Y) = cq_1(X,Y) \vee cq_2(X,Y)$ is a UCQ.

We now define dl-atoms α of form $\text{DL}[\lambda; q](\vec{X})$, where $\lambda = S_1\ op_1\ p_1, \ldots, S_m\ op_m\ p_m$, $m \geq 0$, is a list of expressions $S_i\ op_i\ p_i$, where each S_i is either a concept or a role, $op_i \in \{\uplus, \cup, \cap\}$, and p_i is a predicate symbol matching S_i's arity, and where q is a (U)CQ with output variables $\vec{X}$ (in this case, α is called a *(u)cq-atom*), or $q(\vec{X})$ is a dl-query. Each p_i is an *input predicate symbol*; intuitively, $op_i = \uplus$ increases S_i by the extension of p_i, while $op_i = \cup$ increases $\neg S_i$; $op_i = \cap$ constrains S_i to p_i.

Example 3. The cq-atom $\text{DL}[parent \uplus p; parent(X,Y), parent(Y,Z)](X,Z)$ with output X, Z extends L by adding the extension of p to the property *parent*, and then joins *parent* with itself.

A *cq-rule* r is of the form $a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n$, where every a_i is a literal and every b_j is either a literal or a dl-atom. If $n = 0$ and $k > 0$, then r is a *fact*. A *cq-program* $KB = (L, P)$ consists of a DL-KB L and a finite set of cq-rules P.

Semantics For any CQ $q(\vec{X}) = \{Q_1(\vec{X}_1), Q_2(\vec{X}_2), \ldots, Q_n(\vec{X}_n)\}$, let $\phi_q(\vec{X}) = \exists \vec{Y} \bigwedge_{i=1}^{n} Q_i(\vec{X}_i)$, where $\vec{Y}$ are the variables not in $\vec{X}$, and for any UCQ $q(\vec{X}) = \bigvee_{i=1}^{m} q_i(\vec{X})$, let $\phi_q(\vec{X}) = \bigvee_{i=1}^{m} \phi_{q_i}(\vec{X})$. Then, for (U)CQ $q(\vec{X})$, the set of *answers of* $q(\vec{X})$ *on* L is the set of tuples $ans(q(\vec{X}), L) = \{\vec{c} \in \mathcal{C}^{|\vec{X}|} \mid L \models \phi_q(\vec{c})\}$.

Let $KB = (L, P)$ be a cq-program. Given the semantics of (U)CQs on L, defining the semantics of cq- and ucq-atoms w.r.t. a Herbrand interpretation I of the predicates in P (using constants from $\mathcal{C}$) in the same way as for dl-atoms is straightforward. We recall that a ground dl-atom $a = \text{DL}[\lambda; Q](\vec{c})$ is satisfied w.r.t. I, denoted $I \models_L a$, if $L \cup \lambda(I) \models Q(\vec{c})$, where $\lambda(I) = \bigcup_{i=1}^{m} A_i$ and

- $A_i(I) = \{S_i(\vec{e}) \mid p_i(\vec{e}) \in I\}$, for $op_i = \uplus$;
- $A_i(I) = \{\neg S_i(\vec{e}) \mid p_i(\vec{e}) \in I\}$, for $op_i = \cup$;
- $A_i(I) = \{\neg S_i(\vec{e}) \mid p_i(\vec{e}) \in I$ does not hold$\}$, for $op_i = \cap$.

Now, given a ground instance $a(\vec{c})$ of a (u)cq-atom $a(\vec{X}) = \text{DL}[\lambda; q](\vec{X})$ (i.e., all variables in $q(\vec{X})$ are replaced by constants), I satisfies $a(\vec{c})$, denoted $I \models_L a(\vec{c})$, if $\vec{c} \in ans(q(\vec{X}), L \cup \lambda(I))$. The notion of model and (strong) answer set of KB is then defined as usual (cf. [5, 2]).

Example 4. Let $KB = (L, P)$, where L is the well-known wine ontology[4] and P is as follows (P uses only atomic queries and may launch our rewritings):

[4] http://www.w3.org/TR/owl-guide/wine.rdf

$$
\begin{aligned}
v(L) \vee \neg v(L) \leftarrow\ & \mathrm{DL}[WhiteWine](W), \mathrm{DL}[RedWine](R), \mathrm{DL}[locatedIn](W, L), \\
& \mathrm{DL}[locatedIn](R, L), \mathit{not}\, \mathrm{DL}\big[locatedIn(L, L')\big](L). \\
\leftarrow\ & v(X), v(Y), X \neq Y. \qquad c \leftarrow v(X). \qquad \leftarrow \mathit{not}\, c. \\
del(W) \leftarrow\ & \mathrm{DL}[hasFlavor](W, wine{:}Delicate). \\
del\_r(W) \leftarrow\ & v(L), del(W), \mathrm{DL}[locatedIn](W, L).
\end{aligned}
$$

Informally, the first rule picks a largest region in which both red and white wine grow, and the next three rules make sure that exactly one such region is picked. The last rules choose the delicate wines in the region selected for visit.

KB has the following 3 strong answer sets (only positive facts from predicates del_r and v are listed): $\{del\_r(MountadamRiesling), v(AustralianRegion), \dots\}$, $\{del\_r(LaneTannerPinotNoir), del\_r(WhitehallLanePrimavera), v(USRegion), \dots\}$, and $\{del\_r(StonleighSauvignonBlanc), v(NewZealandRegion), \dots\}$.

The example in the introduction shows that cq-programs are more expressive than dl-programs in [5, 2]. Furthermore, answer set existence for KB and reasoning from the answer sets of KB is decidable if (U)CQ-answering on L is decidable, which is feasible for quite expressive DLs including $\mathcal{SHIQ}$ and fragments of $\mathcal{SHOIN}$, cf. [8–10]. Rosati's well-known $\mathcal{DL}$+*log* formalism [11, 1], and the more expressive hybrid MKNF knowledge bases [12, 13] are closest in spirit to dl- and cq-programs, since they support nonmonotonic negation and use constructions from nonmonotonic logics. However, their expressiveness seems to be different from dl- and cq-programs. It is reported in [12] that dl-programs (and hence also cq-programs) can not be captured using MKNF rules. In turn, the semantics of $\mathcal{DL}$+*log* inherently involves deciding containment of CQs in UCQs, which seems to be inexpressible in cq-programs.

3 Rewriting Rules for cq- and ucq-Atoms

As shown in Ex. 1, in cq-programs we might have different choices for defining the same query. Indeed, the rules r and r' are equivalent over any DL-KB L. However, r' performs the join on the DL side in a single call to the DL-reasoner, while r performs the join on the logic program side, over the results of two calls to the DL-reasoner. In general, making more calls is more costly, and thus r' may be computationally preferable. Furthermore, the result transferred by the single call in r' is smaller than the results of the two calls.

Towards exploiting such rewriting, we present some transformation rules for replacing a rule or a set of rules in a cq-program with another rule or set of rules, while preserving the semantics of the program (see Table 1). By (repeated) application of these rules, the program can be transformed into another, equivalent program. Note that ordinary dl-atoms $\mathrm{DL}[\lambda; Q](\vec{t})$, may be replaced by equivalent cq-atoms $\mathrm{DL}[\lambda; Q(\vec{t})](\vec{X})$, where $\vec{X} = \mathrm{vars}(\vec{t})$, to facilitate rewriting.

Query Pushing (A) By this rule, cq-atoms $\mathrm{DL}[\lambda; cq_1](\vec{Y_1})$ and $\mathrm{DL}[\lambda; cq_2](\vec{Y_2})$ in the body of a rule (A1) can be merged. In rule (A2), cq_1' and cq_2' are constructed

Query Pushing

$$r:\ a_1 \vee \cdots \vee a_k \leftarrow \mathrm{DL}[\lambda; cq_1](\vec{Y_1}), \mathrm{DL}[\lambda; cq_2](\vec{Y_2}), B. \tag{A1}$$

$$r':\ a_1 \vee \cdots \vee a_k \leftarrow \mathrm{DL}\big[\lambda; cq_1' \cup cq_2'\big](\vec{Y_1} \cup \vec{Y_2}), B. \tag{A2}$$

where $B = b_1, \ldots, b_m, not\, b_{m+1}, \ldots, not\, b_n$.

(In)equality Pushing

$$r:\ a_1 \vee \cdots \vee a_h \leftarrow \mathrm{DL}[\lambda; cq](\vec{Y}), Y_{i_1} \neq Y_{i_2}, \ldots, Y_{i_{2k-1}} \neq Y_{i_{2k}}, \quad Y_{i_{2k+1}} = Y_{i_{2k+2}}, \ldots, Y_{i_{2l-1}} = Y_{i_{2l}}, B. \tag{B1}$$

$$r':\ a_1 \vee \cdots \vee a_h \leftarrow \mathrm{DL}\big[\lambda; cq' \cup \{Y_{i_1} \neq Y_{i_2}, \ldots, Y_{i_{2k-1}} \neq Y_{i_{2k}}\}\big](\vec{Y}), B. \tag{B2}$$

where each $Y_{i_j} \in \vec{Y}$ for $1 \leq j \leq 2l$, and $B = b_1, \ldots, b_m, not\, b_{m+1}, \ldots, not\, b_n$.

Fact Pushing

$$\bar{P} = \left\{ \begin{array}{l} f(\vec{c_1}), f(\vec{c_2}), \ldots, f(\vec{c_l}), \\ a_1 \vee \cdots \vee a_k \leftarrow \mathrm{DL}\big[\lambda; \bigvee_{i=1}^{r} cq_i\big](\vec{Y}), f(\vec{Y'}), B. \end{array} \right\} \tag{C1}$$

$$\bar{P'} = \left\{ \begin{array}{l} f(\vec{c_1}), f(\vec{c_2}), \ldots, f(\vec{c_l}), \\ a_1 \vee \cdots \vee a_k \leftarrow \mathrm{DL}\Big[\lambda; \bigvee_{i=1}^{r} \Big(\bigvee_{j=1}^{l} cq_i \cup \{\vec{Y'} = \vec{c_j}\}\Big)\Big](\vec{Y}), B. \end{array} \right\} \tag{C2}$$

$\vec{c_1}, \ldots, \vec{c_l}$ are ground tuples, $\vec{Y'} \subseteq \vec{Y}$, and $B = b_1, \ldots, b_m, not\, b_{m+1}, \ldots, not\, b_n$.

Unfolding

$$\bar{P} = \left\{ \begin{array}{l} r_1 : a_1 \vee \cdots \vee a_i \leftarrow a'(\vec{Y}), B_1. \\ r_2 :\ H' \vee a'(\vec{Y'}) \leftarrow B_2. \end{array} \right\} \tag{D1}$$

$$\bar{P'} = \bar{P} \cup \big\{\, r_1' : H'\theta \vee a_1\theta \vee \cdots \vee a_i\theta \leftarrow B_2\theta, B_1\theta. \,\big\} \tag{D2}$$

$H' = a_1' \vee \cdots \vee a_j'$, and θ is the mgu of $a'(\vec{Y})$ and $a'(\vec{Y'})$ (thus $a'(\vec{Y}\theta) = a'(\vec{Y'}\theta)$);

Where $a'(\vec{Y})$ is not unifiable with $a'(\vec{Z}) \in H(r_1) \cup H'$, alternatively $\bar{P'} = \{r_1', r_2\}$.

Table 1. Equivalences

by renaming variables in cq_1 and cq_2 as follows. Let $\vec{Z_1}$ and $\vec{Z_2}$ be the non-distinguished (i.e., existential) variables of cq_1 resp. cq_2. Rename each $X \in \vec{Z_1}$ occurring in cq_2 and each $X \in \vec{Z_2}$ occurring in cq_1 to a fresh variable.

Query Pushing can be similarly done when one or both of cq_1, cq_2 is a UCQ ucq_1 resp. ucq_2; here, we simply distribute the subqueries and form a single UCQ.

Pushing of (In)equalities (B) If the DL-engine is used under the unique name assumption and supports (in)equalities in the query language, we can easily rewrite rules with equality ($=$) or inequality ($\neq$) in the body by pushing it to the cq-query. A rule of form (B1) can be replaced by (B2), where the CQ cq' results from cq by collapsing variables according to the equalities $Y_{i_{2k+1}} = Y_{i_{2k+2}}$, $\ldots$, $Y_{i_{2l-1}} = Y_{i_{2l}}$.

Example 5. Consider rule $r = bigwinery(M) \leftarrow \mathrm{DL}[Wine](W_1), \mathrm{DL}[Wine](W_2), W_1 \neq W_2, \mathrm{DL}[hasMaker](W_1, M), \mathrm{DL}[hasMaker](W_2, M)$. Here, we want to know all wineries producing at least two different wines. We can rewrite r, by Query and Inequality Pushing, to the rule r'

$$r':\ bigwinery(M) \leftarrow \mathrm{DL}\left[\begin{array}{c} Wine(W_1), Wine(W_2), W_1 \neq W_2 \\ hasMaker(W_1, M), hasMaker(W_2, M) \end{array}\right](M, W_1, W_2).$$

A similar rule works for a ucq-atom $\mathrm{DL}[\lambda; ucq](\vec{Y})$ in place of $\mathrm{DL}[\lambda; cq](\vec{Y})$.

Fact Pushing (C) Suppose we have a program with "selection predicates", i.e., facts which serve to select a specific property in a rule. We can push such facts into a ucq-atom and remove the selection atom from the rule body.

Example 6. Consider the program P, where we only want to know the children of *joe* and *jill*: $P = \{f(joe), f(jill), fchild(Y) \leftarrow \mathrm{DL}[isFatherOf](X,Y), f(X).\}$ We may rewrite the program to a more compact one with the help of ucq-atoms:

$$P' = \left\{ fchild(Y) \leftarrow \mathrm{DL}\left[\begin{array}{l}\{isFatherOf(X,Y), X = joe\} \vee \\ \{isFatherOf(X,Y), X = jill\}\end{array}\right](X,Y). \right\}$$

Such a rewriting makes sense in situations were *isFatherOf* has many tuples and thus would lead to transfer all known father child relationships.

Unfolding (D) Unfolding rules is a standard-method for partial evaluation of ordinary logic programs under answer set semantics. It can be also applied in the context of cq-programs, with no special adaptation. After folding rules with (u)cq-atoms in the body into other rules, subsequent Query Pushing might be applied. In this way, inference propagation can be shortcut.

The following results state that the rewritings preserve equivalence. Let $P \equiv_L Q$ denote that (L, P) and (L, Q) have the same answer sets.

Theorem 1. *For an $X \in \{A,B\}$ let r and r' be rules of form $(X1)$ and $(X2)$, respectively. Let (L, P) be a cq-program with $r \in P$. Then, $P \equiv_L (P \setminus \{r\}) \cup \{r'\}$.*

Theorem 2. *Let $\bar{P}$ be a set of cq-rules of form* (C1) (*resp.* (D1)) *and $\bar{P}'$ be a set of cq-rules of form* (C2) (*resp.* (D2)). *Then, $\bar{P} \equiv_L \bar{P}'$. For any set of cq-rules P such that $\bar{P} \subseteq P$, $P \equiv_L (P \setminus \bar{P}) \cup \bar{P}'$, where for* (D2) $\bar{P}' = \{r_1, r_2, r_1'\}$.

Based on these rules, we have developed an optimization algorithm, described in the extended version of this paper.[5] Further, more general rewriting rules (e.g., incorporating cost models) can be conceived, which we omit for space reasons.

4 Experimental Results

We have tested the rule transformations using the prototype implementation of the DL-plugin for dlvhex,[6] a logic programming engine featuring higher-order syntax and external atoms (see [14]), which uses RACER 1.9 as DL-reasoner (cf. [15]). To our knowledge, this is currently the only implemented system for such a coupling of nonmonotonic logic programs and Description Logics.

The tests were done on a P4 3GHz PC with 1GB RAM under Linux 2.6. As an ontology benchmark, we used the testsuite described in [16]. The experiments covered particular query rewritings (see Table 2) and version of the region program (Ex. 4) with the optimizations applied. We report only part of the results, shown in Fig. 1. Missing entries mean memory exhaustion during the evaluation.

In most of the tested programs, the performance boost using the aforementioned optimization techniques was substantial. Due to the size of the respective

[5] http://www.kr.tuwien.ac.at/staff/roman/papers/dlopt-ext.pdf
[6] http://www.kr.tuwien.ac.at/research/dlvhex/

VICODI program: (Fact Pushing)

$P_v = \{c(\mathit{vicodi{:}Economics}), c(\mathit{vicodi{:}Social}), v(X) \leftarrow \mathrm{DL}[\mathit{hasCategory}](X,Y), c(Y).\}$

SEMINTEC query: (Query Pushing)

$$P_{s_2} = \left\{ s_2(X,Y,Z) \leftarrow \begin{array}{l} \mathrm{DL}[\mathit{Man}](X), \mathrm{DL}[\mathit{isCreditCard}](Y,X), \mathrm{DL}[\mathit{Gold}](Y), \\ \mathrm{DL}[\mathit{livesIn}](X,Z), \mathrm{DL}[\mathit{Region}](Z) \end{array} \right\}$$

SEMINTEC costs: (Query Pushing, Functional Property)

$P_l = \{l(X,Y) \leftarrow \mathrm{DL}[\mathit{hasLoan}](X,Y), \mathrm{DL}[\mathit{Finished}](Y).\}$

hasLoan is an inverse functional property and $|\mathit{hasLoan}| = 682(n+1)$, $|\mathit{Finished}| = 234(n+1)$, where n is obtained from the ontology instance SEMINTEC_n.

Table 2. Some test queries

ontologies, in some cases the DL-engine failed to evaluate the original dl-queries, while the optimized programs did terminate with the correct result.

In detail, for the region program (upper left graph), we used the ontologies wine_0 through wine_9. There is a significant speedup, and in case of wine_9 only the optimized program could be evaluated. Most of the computation time was spent by RACER. We note that the result of the join in the body of the first rule had a size merely linear in the number of top regions L; a higher performance gain may be expected for ontologies with larger joins.

The VICODI test series revealed the power of Fact Pushing (see the upper right graph). While the unoptimized P_v could be evaluated only with VICODI_0 and _1, all ontologies VICODI_i, $0 \leq i \leq 4$, could be handled with the optimized program.

The SEMINTEC tests dealt with Query Pushing and show a significant evaluation speedup (see lower row of Fig. 1). P_{s_2} is from one of the benchmark queries in [16], while P_l tests the performance increase when pushing a query to a functional property. In both cases, we used the ontologies SEMINTEC_i, $0 \leq i \leq 4$, but could only complete the tests of the optimized programs on all SEMINTEC_n. The performance gain for P_l is in line with the constant join selectivity.

Future work will be to compare to realizations of cq-programs based on other DL-engines which host CQs, such as Pellet and KAON2, and to enlarge and refine the rewriting techniques.

References

1. Rosati, R.: Integrating Ontologies and Rules: Semantic and Computational Issues. In: Reasoning Web. LNCS 4126, Springer (2006) 128–151
2. Eiter, T., Ianni, G., Polleres, A., Schindlauer, R., Tompits, H.: Reasoning with Rules and Ontologies. In: Reasoning Web. LNCS 4126, Springer (2006) 93–127
3. Antoniou, G., Damásio, C.V., Grosof, B., Horrocks, I., Kifer, M., Maluszynski, J., Patel-Schneider, P.F.: Combining Rules and Ontologies: A survey. Tech. Rep. IST506779/Linköping/I3-D3/D/PU/a1, Linköping University (2005)
4. Pan, J.Z., Franconi, E., Tessaris, S., Stamou, G., Tzouvaras, V., Serafini, L., Horrocks, I., Glimm, B.: Specification of Coordination of Rule and Ontology Languages. Project Deliverable D2.5.1, KnowledgeWeb NoE (2004)

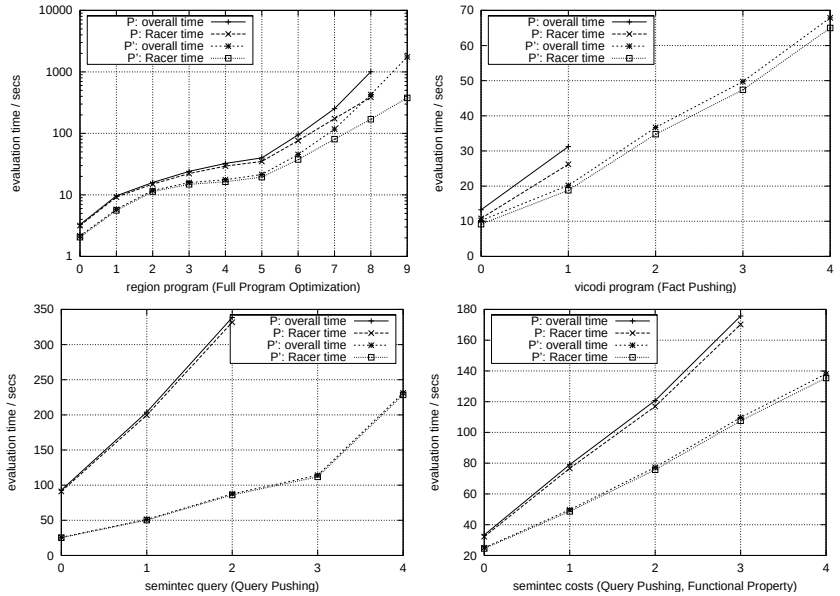

Fig. 1. Evaluation time for Ex. 4, VICODI, and SEMINTEC tests.

5. Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining Answer Set Programming with Description Logics for the Semantic Web. In Dubois, D., et al. eds.: Proceedings KR 2004, Morgan Kaufmann (2004) 141–151
6. Motik, B., Sattler, U., Studer, R.: Query Answering for OWL-DL with Rules. Journal of Web Semantics **3** (2005) 41–60
7. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)
8. Ortiz de la Fuente, M., Calvanese, D., Eiter, T.: Data Complexity of Answering Unions of Conjunctive Queries in SHIQ. In: Proceedings DL 2006 (2006) 62–73
9. Ortiz de la Fuente, M., Calvanese, D., Eiter, T., Franconi, E.: Characterizing Data Complexity for Conjunctive Query Answering in Expressive Description Logics. In: Proceedings AAAI-2006, AAAI Press (2006)
10. Glimm, B., Horrocks, I., Lutz, C., Sattler, U.: Conjunctive Query Answering for the Description Logic $\mathcal{SHIQ}$. In: Proc. IJCAI 2007, AAAI Press (2007) 399–404
11. Rosati, R.: $\mathcal{DL}$+*log*: Tight Integration of Description Logics and Disjunctive Datalog. In: Proceedings KR 2006, AAAI Press (2006) 68–78
12. Motik, B., Horrocks, I., Rosati, R., Sattler, U.: Can OWL and Logic Programming live together happily ever after? In: Proceedings ISWC-2006. LNCS 4273, Springer (2006) 501–514
13. Motik, B., Rosati, R.: A faithful Integration of Description Logics with Logic Programming. In: Proceedings IJCAI 2007, AAAI Press/IJCAI (2007) 477–482
14. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In: Proceedings IJCAI-2005, Professional Book Center (2005) 90–96
15. Haarslev, V., Möller, R.: RACER System Description. In: Proceedings IJCAR-2001. LNCS 2083, Springer (2001) 701–705
16. Motik, B., Sattler, U.: A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes. In Hermann, M., Voronkov, A., eds.: LPAR. LNCS 4246, Springer (2006) 227–241