Rewriting Ontological Queries into Small Nonrecursive Datalog Programs

Georg Gottlob
Department of Computer Science
University of Oxford
georg.gottlob@cs.ox.ac.uk

Thomas Schwentick
Fakultät für Informatik
TU Dortmund
thomas.schwentick@udo.edu

Abstract

We consider the setting of ontological database access, where an A-box is given in form of a relational database D and where a Boolean conjunctive query q has to be evaluated against D modulo a T-box Σ formulated in DL-Lite or Linear Datalog$^{\pm}$. It is well-known that (Σ, q) can be rewritten into an equivalent nonrecursive Datalog program P that can be directly evaluated over D. However, for Linear Datalog$^{\pm}$ or for DL-Lite versions that allow for role inclusion, the rewriting methods described so far result in a nonrecursive Datalog program P of size exponential in the joint size of Σ and q. This gives rise to the interesting question of whether such a rewriting necessarily needs to be of exponential size. In this paper we show that it is actually possible to translate (Σ, q) into a polynomially sized equivalent nonrecursive Datalog program P.

1 Introduction

1.1 Motivation

This paper is about query rewriting in the context of ontological database access. Query rewriting is an important new optimization technique specific to ontological queries. The essence of query rewriting, as will be explained in more detail below, is to compile a query and an ontological theory (usually formulated in some description logic or rule-based language) into a target query language that can be directly executed over a relational database management system (DBMS). The advantage of such an approach is obvious. Query rewriting can be used as a preprocessing step for enabling the exploitation of mature and efficient existing database technology to answer ontological queries. In particular, after translating an ontological query into SQL, sophisticated query-optimization strategies can be used to efficiently answer it. However, there is a pitfall here. If the translation inflates the query excessively and creates from a reasonably sized ontological query an enormous exponentially sized SQL query (or SQL DDL program), then even the best DBMS may be of little use.

The above problem has motivated a flourishing research activity at the cutting edge between the fields of description logic (DL) and database theory. Specific query languages with tractable query-answering have been designed, including *OWL 2 QL* (Grau et al. 2008; Pérez-Urbina, Horrocks, and Motik 2009), which is an OWL-based equivalent of the description logic *DL-Lite*$_{\mathcal{R}}$ (Calvanese et al. 2007). While experimental and commercial ontological database management systems have existed and have been used for a number of years (e.g. (Chong et al. 2005; Acciarri et al. 2005; Virgilio et al. 2011)), they have been problematic with respect to performance, and there is an agreement that ontological query optimization methods have not yet been developed to their full potential. In fact, there is a general feeling that many issues of ontology querying are still not well understood, and that deep theoretical research is necessary to better understand some fundamental issues of query rewriting. In this spirit, the present paper tries to shed light on a relevant problem in this context: Is it at all possible to translate an ontological conjunctive query into a polynomially sized SQL (or SQL-DDL) query?

1.2 Main Result

We show that polynomially sized query rewritings into nonrecursive Datalog exist in specific settings. Note that nonrecursive Datalog can be efficiently translated into SQL with view definitions (SQL DDL), which, in turn, can be directly executed over any standard DBMS. Our results are — for the time being — of theoretical nature and we do not claim as of now that they will lead to better practical algorithms. A smart implementation of our new query translation, making use of various Datalog optimization techniques, shall be the subject of interesting future research. Our main result applies to the setting where ontological constraints are formulated in terms of *tuple-generating dependencies (tgds)*, and we make heavy use of the well-known *chase* procedure (Maier, Mendelzon, and Sagiv 1979; Johnson and Klug 1984). For definitions, see Section 2. The result after chasing a tgd set Σ over a database D is denoted by $chase(D, \Sigma)$.

Consider a set Σ of tgds and a database D over a joint signature $\mathcal{R}$. Let q be a Boolean conjunctive query (BCQ) issued against (D, Σ). We would like to transform q into a nonrecursive Datalog query P such that $(D, \Sigma) \models q$ iff $D \models P$. We assume here that P has a special propositional goal $goal$, and $D \models P$ means that $goal$ is derivable from P when evaluated over D. Let us define an important property of classes of tgds.

Definition 1.1. Polynomial witness property (PWP). *The PWP holds for a class $\mathcal{C}$ of tgds if there exists a polynomial γ such that, for every finite set $\Sigma \subseteq \mathcal{C}$ of tgds and each BCQ q, the following holds: for each database D, whenever $(D, \Sigma) \models q$, then there is a sequence of at most $\gamma(|\Sigma|, |q|)$ chase steps whose atoms already entail q.*

Our main technical result, which is more formally stated and proven in Section 3, is as follows.

Main Theorem. *Let Σ be a set of tgds from a class $\mathcal{C}$ enjoying the PWP and let a denote the maximum arity of any predicate symbol occurring in Σ. Then each BCQ q can be rewritten in polynomial time into a nonrecursive Datalog program P of size polynomial in the joint size of q and Σ, such that for every database D, $(D, \Sigma) \models q$ if and only if $D \models P$. Moreover, the arity of P is bounded by $\max(a + 1, 3)$, in case a sufficiently large linear order can be accessed in the database, or otherwise by $O(\max(a + 1, 3) \cdot \log m)$, where m is the joint size of q and Σ.*

1.3 Other Results

From this result, and from already established facts, a good number of further rewritability results for other formalisms can be derived. In particular, we can show that conjunctive queries based on other classes of tgds or description logics can be efficiently translated into nonrecursive Datalog. Among these formalisms are: linear tgds, originally defined in (Calì, Gottlob, and Lukasiewicz 2009) and equivalent to inclusion dependencies, various major versions of the well-known description logic DL-Lite (Calvanese et al. 2007; Poggi et al. 2008), OWL 2 QL, and sticky tgds (Calì, Gottlob, and Pieris 2011) as well as sticky-join tgds (Calì, Gottlob, and Pieris 2010b; 2010c). We will just give an overview and very short explanations of how each of these rewritability results follows from our main theorem. A more detailed treatment is planned for a future journal version of this paper.

1.4 Structure of the Paper

The rest of the paper is structured as follows. In Section 2 we state a few preliminaries and simplifying assumptions. In Section 3, we give a rather detailed proof sketch of the main result. Section 4 contains the other results following from the main result. A brief overview of related work concludes the paper in Section 5.

This paper extends a shorter version presented at *DL 2011*, (Gottlob and Schwentick 2011).

2 Preliminaries and Assumptions

We assume the reader to be familiar with the terminology of relational databases and the concepts of *conjunctive query (CQ)* and *Boolean conjunctive query (BCQ)*. For simplicity, we restrict our attention to Boolean conjunctive queries q. However, our results can easily be reformulated for queries with output, see Remark 3 after the proof of Theorem 3.1.

Given a relational schema $\mathcal{R}$, a *tuple-generating dependency (tgd)* σ is a first-order formula of the form $\forall \vec{X} \forall \vec{Y}\, \Phi(\vec{X}, \vec{Y}) \rightarrow \exists \vec{Z}\, \Psi(\vec{X}, \vec{Z})$, where $\Phi(\vec{X}, \vec{Y})$ and $\Psi(\vec{X}, \vec{Z})$ are conjunctions of atoms over $\mathcal{R}$, called the *body* and the *head* of σ, denoted $body(\sigma)$ and $head(\sigma)$, respectively. We usually omit the universal quantifiers in tgds. Such σ is satisfied in a database D for $\mathcal{R}$ iff, whenever there exists a homomorphism h that maps the atoms of $\Phi(\vec{X}, \vec{Y})$ to atoms of D, there exists an extension h' of h that maps the atoms of $\Psi(\vec{X}, \vec{Z})$ to atoms of D. All sets of tgds are finite here. We assume in the rest of the paper that every tgd has exactly one atom and at most one existentially quantified variable in its head. A set of tgds is in *normal form* if the head of each tgd consists of a single atom. It was shown in (Calì, Gottlob, and Kifer 2008, Lemma 10) that every set Σ of TGDs can be transformed into a set Σ' in normal form of size at most quadratic in $|\Sigma|$, such that Σ and Σ' are equivalent with respect to query answering. The normal form transformation shown in (Calì, Gottlob, and Kifer 2008) can be achieved in logarithmic space. It is, moreover, easy to see that this very simple transformation preserves the polynomial witness property.

For a database D for $\mathcal{R}$, and a set of tgds Σ on $\mathcal{R}$, the set of *models* of D and Σ, denoted $mods(D, \Sigma)$, is the set of all (possibly infinite) databases B such that (i) $D \subseteq B$ and (ii) every $\sigma \in \Sigma$ is satisfied in B. The set of *answers* for a CQ q to D and Σ, denoted $ans(q, D, \Sigma)$, is the set of all tuples $\underline{a}$ such that $\underline{a} \in q(B)$ for all $B \in mods(D, \Sigma)$. The *answer* for a BCQ q to D and Σ is *yes* iff the empty tuple is in $ans(q, D, \Sigma)$, also denoted as $D \cup \Sigma \models q$.

Note that, in general, query answering under tgds is undecidable (Beeri and Vardi 1981), even when the schema and tgds are fixed (Calì, Gottlob, and Kifer 2008). Query answering is, however, decidable for interesting classes of tgds, among which are those considered in the present paper.

The *chase* procedure was introduced to enable checking implication of dependencies (Maier, Mendelzon, and Sagiv 1979), and later also for checking query containment (Johnson and Klug 1984). It is a procedure for repairing a database relative to a set of dependencies, so that the result of the chase satisfies the dependencies. By "chase", we refer both to the chase procedure and to its output. The chase comes in two flavors: *restricted* and *oblivious*, where the restricted chase applies tgds only when they are not satisfied (to repair them), while the oblivious chase always applies tgds (if they produce a new result). We focus on the oblivious one, since it makes proofs technically simpler. The *(oblivious) tgd chase rule* defined below is the building block of the chase.

TGD CHASE RULE. Consider a database D for a relational schema $\mathcal{R}$, and a tgd σ on $\mathcal{R}$ of the form $\Phi(\vec{X}, \vec{Y}) \rightarrow \exists \vec{Z}\, \Psi(\vec{X}, \vec{Z})$. Then, σ is *applicable* to D if there exists a homomorphism h that maps the atoms of $\Phi(\vec{X}, \vec{Y})$ to atoms of D. Let σ be applicable to D, and h_1 be a homomorphism that extends h as follows: for each $X_i \in \vec{X}$, $h_1(X_i) = h(X_i)$; for each $Z_j \in \vec{Z}$, $h_1(Z_j) = z_j$, where z_j is a fresh null value (i.e., a Skolem constant) different from all nulls already introduced. The *application of σ* on D adds to D the atom $h_1(\Psi(\vec{X}, \vec{Z}))$ if not already in D (which is possible when $\vec{Z}$ is empty). ■

The chase algorithm for a database D and a set of tgds Σ consists of an exhaustive application of the tgd chase rule in a breadth-first (level-saturating) fashion, which leads as result to a (possibly infinite) chase for D and Σ. Formally, the *chase of level up to* 0 of D relative to Σ, denoted $chase^0(D, \Sigma)$, is defined as D, assigning to every atom in D the *(derivation) level* 0. For every $k \geq 1$, the *chase of level up to* k of D relative to Σ, denoted $chase^k(D, \Sigma)$, is constructed as follows: let $I_1, \ldots, I_n$ be all possible images of bodies of tgds in Σ relative to some homomorphism such that (i) $I_1, \ldots, I_n \subseteq chase^{k-1}(D, \Sigma)$ and (ii) the highest level of an atom in some I_i is $k-1$; then, perform every corresponding tgd application on $chase^{k-1}(D, \Sigma)$, choosing the applied tgds and homomorphisms in a linear and lexicographic order, respectively, and assigning to every new atom the *(derivation) level* k. The *chase* of D relative to Σ, denoted $chase(D, \Sigma)$, is thus the limit of $chase^k(D, \Sigma)$ for $k \to \infty$.

The (possibly infinite) chase relative to tgds is a *universal model*, i.e., there exists a homomorphism from $chase(D, \Sigma)$ onto every $B \in mods(D, \Sigma)$ (Deutsch, Nash, and Remmel 2008; Calì, Gottlob, and Kifer 2008). This result implies that BCQs q over D and Σ can be evaluated on the chase for D and Σ, i.e., $D \cup \Sigma \models q$ is equivalent to $chase(D, \Sigma) \models q$.

A *chase sequence* of length n based on D and Σ is a sequence of n atoms such that each atom is either from D or can be derived via a single application of some rule in Σ from previous atoms in the sequence. If S is such a chase sequence and q a conjunctive query, we write $S \models q$ if there is a homomorphism from q to the set of atoms of S.

We assume that every database has two constants, 0 and 1, that are available via the unary predicates $Zero$ and One, respectively. Moreover, each database has a binary predicate Neq such that Neq(a, b) is true precisely if a and b are distinct values.

We finally define *N-numerical databases*. Let D be a database whose domain does not contain any natural numbers. We define D_N as the extension of D by adding the natural numbers $0, 1, \ldots, N$ to its domain, a unary relation Num that contains exactly the numbers $1, \ldots, N$, binary order relations $Succ$ and $<$ on $0, 1, \ldots, N$, expressing the natural successor and "$<$" orders on N, respectively.[1] We refer to D_N as the *N-numerical extension* of D, and, a so extended database as *N-numerical database*. We denote the total domain of a numerical database D_N by $\text{dom}_N(D)$ and the non-numerical domain (still) by $\text{dom}(D)$. Standard databases can always be considered to be N-numerical, for some large N by the standard type *integer*, with the $<$ predicate (and even arithmetic operations). A number $maxint$ corresponding to N can be defined.

3 Main Result

Our main result is more formally stated as follows:

Theorem 3.1. *Let $\mathcal{C}$ be a class of tgds in normal form, enjoying the polynomial witness property and let γ be the polynomial bounding the number of chase steps (with $\gamma(n_1, n_2) \geq \max(n_1, n_2)$, for all naturals n_1, n_2). For each set $\Sigma \subseteq \mathcal{C}$ of tgds and each Boolean CQ q, one can compute in polynomial time a nonrecursive Datalog program P of polynomial size in $|\Sigma|$ and $|q|$, such that, for every database D it holds $D, \Sigma \models q$ if and only if $D \models P$. Furthermore:*

(a) For N-numerical databases D, where $N \geq \gamma(|\Sigma|, |q|)$, the arity of P is $\max(a+1, 3)$, where a is the maximum arity of any predicate symbol occurring in Σ;

(b) otherwise (for non-numerical databases), the arity of P is $\mathcal{O}(\max(a+1, 3) \cdot \log \gamma(|\Sigma|, |q|))$, where a is as above.

We note that N is polynomially bounded in $|\Sigma|$ and $|q|$ by the polynomial γ that only depends on $\mathcal{C}$.

The rest of this section is dedicated to a proof, in form of a detailed proof sketch, of Theorem 3.1. This proof sketch should provide substantial insight into how our translation works and why it is correct. A fully formal proof requires many more pages and will be given in the journal version of the present paper.

Proof. Our detailed proof sketch starts with an illustration of its high-level idea. We then introduce some notation, conventions, and simplifying assumptions (which we can make without loss of generality). This is followed by the proper proof sketch which is illustrated by a running example.

High-level idea of the proof. We first describe the high level idea of the construction of a nonrecursive Datalog program P of arity $a + k + 4$, where k is the maximum number of tuples in any left hand side of a chase rule. We explain afterwards how the arity can be reduced to $\max(a + 1, 3)$. The program P checks whether there is a chase sequence $S = t_1, \ldots, t_N$ with respect to D and Σ and a homomorphism h from q to (the set of atoms of) S. To this end, P consists of one large rule r_{goal} of polynomial size in N and some shorter rules that define auxiliary relations and will be explained below.

The aim of r_{goal} is to guess the chase sequence S *and* the homomorphism h at the same time. We recall that N does not depend on the size of D but only on $|\Sigma|$ and $|q|$ and thus r_{goal} can well be as long as the chase sequence and q together. One of the advantages of this approach is that we only have to deal with those null values that are actually relevant for answering the query. Thus, at most N null values need to be represented.[2]

One might try to obtain r_{goal} by just taking one atom A_i for each tuple t_i of S and one atom for each atom of q and somehow test that they are consistent. However, it is not clear how consistency could possibly be checked in a purely conjunctive fashion.[3] There are two ways in which disjunctive reasoning is needed. First, it is not a priori clear on which previous tuples, tuple t_i will depend. Second, it is not a priori clear to which tuples of S the atoms of q can be mapped.

[1] Of course, if $\text{dom}(D)$ already contains some natural numbers we can add a fresh copy of $\{0, 1, \ldots, N\}$ instead.

[2] We recall that we assume without loss of generality that every tgd has at most one existentially quantified variable in its head.

[3] Furthermore, of course, there are no relations to which the atoms A_i could possible be matched.

To overcome these challenges we use the following basic ideas.

(1) We represent the tuples of S (and the required tuples of D) in a symbolic fashion, utilizing the numerical domain. Thus, for example, an atom $R_1(a, b)$ can be identified by the value triple $\langle 1, a, b\rangle$; a tuple $R_4(a, b, \perp_2)$, where $\perp_2$ is a null-value, can be represented by the quadruple $\langle 4, a, b, 2\rangle$. This shall just give the reader a first idea of how to encode atoms. Our actual encoding (given below in the proof) is based on this idea, but is slightly more involved, as it also contains a bit f that discriminates between original database atoms ($f = 0$) and new atoms derived during the chase ($f = 1$). Moreover, the actual encoding applies to "normalized" atoms that are all of the same arity.

(2) We let P compute auxiliary predicates that allow us to express disjunctive relationships between the tuples in S.

Example 3.2. *We shall illustrate the proof idea with a very simple running example, shown in Figure 1.*

A possible chase sequence in this example is shown in Figure 2(a). The mapping $X \mapsto a$ and $Y \mapsto g$, maps $R_5(X, Y)$ to t_5 and $R_3(Y, X)$ to t_6, thus satisfying q. Before we describe the proof idea in more detail, we fix some notation and convenient conventions.

Notation and conventions. Let $\mathcal{C}$ be a class of tgds enjoying the PWP, let Σ be a set of tgds from $\mathcal{C}$, and let q be a BCQ. Let $R_1, \dots R_m$ be the predicate symbols occurring in Σ or in q. We denote the number of tgds in Σ by ℓ.

Let $N := \gamma(|\Sigma|, |q|)$ where γ is as in Definition 1.1, thus N is polynomial in $|\Sigma|$ and $|q|$. By definition of N, if $(D, \Sigma) \models q$, then q can be witnessed by a chase sequence Γ of length $\leq N$. Our assumption that $\gamma(n_1, n_2) \geq \max(n_1, n_2)$, for every n_1, n_2, guarantees that N is larger than (i) the number of predicate symbols occurring in Σ, (ii) the cardinality $|q|$ of the query, and (iii) the number of rules in Σ.

For the sake of a simpler presentation, we assume that all relations in Σ have the same arity a and all rules use the same number k of tuples in their body. The latter can be easily achieved by repeating tuples, the former by filling up shorter tuples by repeating the first tuple entry. Furthermore, we only consider chase sequences of length N. Shorter sequences can be extended by adding tuples from D.

Example 3.3. *Example 3.2 thus translates as illustrated in Figure 3. The (extended) chase sequence is shown in Figure 2 (b). The query q is now satisfied by the mapping $X \mapsto a$, $Y \mapsto g$, $U \mapsto g$, $V \mapsto a$, thus mapping $R_5(X, Y, X)$ to t_5 and $R_3(Y, X, Y)$ to t_6.*

Proof (continued). On an abstract level, the atoms that make up the final rule r_{goal} of P can be divided into three groups serving three different purposes. That is, r_{goal} can be considered as a conjunction $r_{\text{tuples}} \wedge r_{\text{chase}} \wedge r_{\text{query}}$. Each group is "supported" by a sub-program of P that defines relations that are used in r_{goal}, and we refer to these three subprograms as P_{tuples}, P_{chase} and P_{query}, respectively.

- The purpose of r_{tuples} is basically to lay the ground for the other two. It consists of N atoms that allow to guess the symbolic encoding of a sequence $S = t_1, \dots, t_N$.
- The atoms of r_{chase} are designed to verify that S is an actual chase sequence with respect to D.
- Finally, r_{query} checks that there is a homomorphism from q to S.

P_{tuples} and r_{tuples}. We continue with an explanation of the symbolic representation of tuples underlying r_{tuples}.

The symbolic representation of the tuples t_i of the chase sequence S uses numerical values to encode null values, predicate symbols R_i (by i), tgds $\sigma_j \in \Sigma$ (by j) and the number of a tuple t_i in the sequence (that is: i).

In particular, the symbolic encoding uses the following numerical parameters.[4]

- r_i to indicate the relation R_{r_i} to which the tuple belongs;
- f_i to indicate whether t_i is from D ($f_i = 0$) or yielded by the chase ($f_i = 1$);
- Furthermore, $x_{i1}, \dots, x_{ia}$ represent the attribute values of t_i as follows. If the j-th attribute of t_i is a value from $\text{dom}(D)$ then x_{ij} is intended to be that value, otherwise it is a null represented by a numeric value.

Since each rule of Σ has at most one existential quantifier in its head, at each chase step, at most one new null value can be introduced. Thus, we can unambiguously represent the null value (possibly) introduced in the j-th step of the chase by the number j. In particular, all null values introduced in a chase sequence (of length N) can indeed be represented by elements of the numerical domain.

The remaining parameters s_i and $c_{i1}, \dots, c_{ik}$ are used to encode information about the tgd and the tuples (atoms) in S that are used to generate the current tuple. More precisely,

- s_i is intended to be the number of the applied tgd σ_{s_i} and
- $c_{i1}, \dots, c_{ik}$ are the tuple numbers of the k tuples that are used to yield t_i.

In the example, e.g., t_5 is obtained by applying σ_4 to t_2 and t_4. The encoding of our running example can be found in Figure 2 (c).

We use a new relational symbol T of arity $a + k + 4$ not present in the schema of D for the representation of the tuples from S. Thus, r_{tuples} is just:

$$T(1, r_1, f_1, x_{11}, \dots, x_{1a}, s_1, c_{11}, \dots, c_{1k}), \dots,$$
$$T(N, r_N, f_N, x_{N1}, \dots, x_{Na}, s_N, c_{N1}, \dots, c_{Nk}).$$

The sub-program P_{tuples} is intended to "fill" T with suitable tuples. The intention is that T contains all tuples that could be used in a chase sequence in principle. At this point, there are no restrictions regarding the chase rules. To this end, P_{tuples} uses two kinds of rules, one for tuples from D and one for tuples yielded by the chase. For each relation symbol R_j of D, P_{tuples} has a rule

[4] We use the names of the parameters as variable names in r_{goal} as well.

(a) Σ :

σ_1: $R_1(X,Y) \to \exists Z\ R_4(X,Y,Z)$
σ_2: $R_2(Y,Z) \to \exists X\ R_4(X,Y,Z)$
σ_3: $R_3(X,Z) \to \exists Y\ R_4(X,Y,Z)$
σ_4: $R_4(X_1,Y_1,Z_1), R_4(X_2,Y_2,Z_2) \to R_5(X_1,Z_2)$

(b) $q : R_5(X,Y), R_3(Y,X)$

(c) D :

R_1	
a	b
c	d

R_2	
e	g

R_3	
g	a
g	h

Figure 1: Simple example with (a) a set Σ of tgds, (b) a query q and (c) a database D.

(a)

- t_1: $R_1(a,b)$
- t_2: $R_4(a,b,\bot_2)$
- t_3: $R_2(e,g)$
- t_4: $R_4(\bot_4,e,g)$
- t_5: $R_5(a,g)$
- t_6: $R_3(g,a)$

(b)

- t_1: $R_1(a,b,a)$
- t_2: $R_4(a,b,\bot_2)$
- t_3: $R_2(e,g,e)$
- t_4: $R_4(\bot_4,e,g)$
- t_5: $R_5(a,g,a)$
- t_6: $R_3(g,a,g)$

(c)

i	r_i	f_i	x_{i1}	x_{i2}	x_{i3}	s_i	c_{i1}	c_{i2}
1	1	0	a	b	a	0	0	0
2	4	1	a	b	2	1	1	1
3	2	0	e	g	e	0	0	0
4	4	1	4	e	g	2	3	3
5	5	1	a	g	a	4	2	4
6	3	0	g	a	g	0	0	0

Figure 2: (a) Example chase sequence, (b) its extension and (c) its encoding. t_2 is obtained by applying σ_1 to t_1. Likewise t_4 and t_5 are obtained by applying σ_2 to t_3 and σ_4 to t_2 and t_4, respectively.

(a) Σ :

σ_1: $R_1(X,Y,X), R_1(X,Y,X) \to \exists Z\ R_4(X,Y,Z)$
σ_2: $R_2(Y,Z,Y), R_2(Y,Z,Y) \to \exists X\ R_4(X,Y,Z)$
σ_3: $R_3(X,Z,X), R_3(X,Z,X) \to \exists Y\ R_4(X,Y,Z)$
σ_4: $R_4(X_1,Y_1,Z_1), R_4(X_2,Y_2,Z_2) \to R_5(X_1,Z_2,X_1)$

(b)

$q : R_5(X,Y,U), R_3(Y,X,V)$

(c) D :

R_1		
a	b	a
c	d	c

R_2		
e	g	e

R_3		
g	a	g
g	h	g

Figure 3: Modified example with (a) a set Σ of tgds, (b) a query q and (c) a database D.

$$T(Z,j,0,X1,\dots,Xa,0,0,\dots,0) :- R_j(X_1,\dots,X_a), \text{Num}(Z).$$

which adds all tuples from R_j to T and makes them accessible for every possible position (Z) in S.

The following rule adds tuples that can possibly be obtained by chase steps.

$$T(Z,Y,1,X_1,\dots,X_a,V,U_1,\dots,U_k) :- \text{Num}(Z), \text{Num}(Y), \text{DNum}(X_1),\dots,\text{DNum}(X_a), \text{Num}(V), \text{Num}(U_1),\dots,\text{Num}(U_k), 1 \le Y \le m, 1 \le V \le \ell, U_1 < Z,\dots,U_k < Z \quad (1)$$

Here, the first two inequalities make sure that only allowed relation and tgd numbers are used, the latter inequalities guarantee that to yield a tuple by a chase rule only tuples with smaller numbers can be used.[5] The rule uses one further predicate DNum that has not yet been defined. Its purpose is to contain all possible values, that is: $\text{dom}(D) \cup \text{Num}$. It is (easily) defined by further rules of P_{tuples}. Note that this leaves the values for the X_j unconstrained, hence they can carry either domain values or numerical values.

[5] As the latter constraints are independent from the concrete tgds, we decided to put them here. They could as well be tested in r_{chase}.

P_{chase} and r_{chase}. Next, we describe the part of r_{goal} that checks that S constitutes an actual chase sequence and the rules of P that specify the corresponding auxiliary relations.

The following kinds of conditions have to be checked to ensure that the tuples "guessed" by r_{tuples} constitute a chase sequence.

(1) For every i, the relation R_{r_i} of a tuple t_i has to match the head of its rule σ_{s_i}.

- In the example, e.g., r_4 has to be 4 as the head of σ_2 is an R_4-atom.

(2) Likewise, for each i and j the relation number of tuple $t_{c_{ij}}$ has to be the relation number of the j-th atom of σ_{s_i}.

- In the example, e.g., r_2 must be 4, as $c_{5,1} = 2$ and the first atom of $\sigma_{s_5} = \sigma_4$ is an R_4-atom.

(3) If the head of σ_{s_i} contains an existentially quantified variable, the new null value is represented by the numerical value i.

- This is illustrated by t_4 in the example: the first position of the head of rule 2 has an existentially quantified variable and thus $x_{4,1} = 4$.

(4) If a variable occurs at two different positions in σ_{s_i} then the corresponding positions in the tuples used to produce t_i carry the same value.

(5) If a variable in the body of σ_{s_i} also occurs in the head of σ_{s_i} then the values of the corresponding positions in the body tuple and in t_i are equal.

- Z_2 occurs in position 3 of the second atom of the body of σ_4 and in position 2 of its head. Therefore, $x_{4,3}$ and $x_{5,2}$ have to coincide (where the 4 is determined by $c_{5,2}$.

Note that all these conditions depend on the given tgds. Indeed, every tgd from Σ contributes conditions of each of the five forms. For the sake of simplicity of presentation, we explain the effect of a tgd through the following example tgd that contains all relevant features that might arise in a tgd. The generalization to arbitrary tgds is straightforward but tedious to spell out in full detail. Let us thus assume that σ_1 is the tgd[6]

$$R_2(X,Y), R_3(Y,Z) \rightarrow \exists V\ R_4(X,V).$$

Condition (1) states that if a tuple t_i is obtained by applying σ_1 it should be a tuple from R_4. In terms of variables this means, that for every i it should hold: if $s_i = 1$ then $r_i = 4$.

This is the first occasion where we need some way to express a disjunction in r_{goal} (namely: $s_i \neq 1 \vee r_i = 4$). We can meet this challenge with the help of an additional predicate to be specified in P_{chase}. More precisely, we let P_{chase} specify a 4-ary predicate IfThen(X_1, X_2, U_1, U_2) that is intended to contain all tuples fulfilling the condition: if $X_1 = X_2$ then $U_1 = U_2$. IfThen can be specified by the following two rules. IfThen$(X, X, U, U) : -$DNum(X), DNum(U).

$$\begin{aligned}\text{IfThen}(X_1, X_2, U_1, U_2) :- \\ \text{DNum}(X_1), \text{DNum}(X_2), \text{DNum}(X_1), \\ \text{DNum}(X_2), \text{DNum}(U_1), \text{DNum}(U_2), \text{Neq}(X_1, X_2)\end{aligned}$$

Thus, condition (1) can be guaranteed with respect to tgd σ_1 for all tuples t_i by adding all atoms of the form IfThen$(s_i, 1, r_i, 4)$ to r_{chase}.

Condition (2) is slightly more complicated. For our example tgd σ_1 it says that if a tuple t_i is obtained using σ_1 then the first tuple used for the chase step should be an R_2-tuple. In terms of variables this can be stated as: if $s_i = 1$ and $c_{i1} = j$ then $r_j = 2$ (and likewise for the second atom of σ_1. To express this IF-statement we use a 6-ary auxiliary predicate IfThen2$(X_1, X_2, Y_1, Y_2, U_1, U_2)$ expressing that if $X_1 = X_2$ and $Y_1 = Y_2$ then $U_1 = U_2$. It can be specified in P_{chase} by the three rules shown in Figure 4.

For every pair of numbers $i, j \leq N$, r_{goal} then has atoms IfThen2$(s_i, 1, c_{i1}, j, r_j, 2)$ and IfThen2$(s_i, 1, c_{i2}, j, r_j, 2)$.

In a similar fashion

- condition (3) yields one atom IfThen$(s_i, 1, x_{i2}, i)$, for every i;
- condition (4) yields one atom IfThen3$(s_i, 1, c_{i1}, j_1, c_{i2}, j_2, x_{j_1 2}, x_{j_2 1})$, for every $i, j_1, j_2 \leq N$, where IfThen3 is the 8-ary predicate for IfThen-statements with three conjuncts that can be defined analogously as IfThen2;
- condition (5) yields one atom IfThen2$(s_i, 1, c_{i1}, j, x_{j1}, x_{i1})$ for every $i, j \leq N$.

Altogether, r_{chase} has $\mathcal{O}(N^3 \ell k)$ atoms that together guarantee that the variables of r_{tuples} encode an actual chase sequence.

P_{query} and r_{query}. Finally, we explain how it can be checked that there is a homomorphism from q to S. We explain the issue through the little example query $R_3(x, y) \wedge R_4(y, z)$. To evaluate this query, r_{query} makes use of two additional variables q_1 and q_2, one for each atom of q. The intention is that these variables bind to the numbers of the tuples that the atoms are mapped to. We have to make sure two kinds of conditions. First, the tuples need to have the right relation symbol and second, they have to obey value equalities induced by the variables of q that occur more than once.

The first kind of conditions is checked by adding atoms IfThen$(q_1, i, r_i, 3)$ and IfThen$(q_2, i, r_i, 4)$ to r_{query}, for every $i \leq N$. The second kind of conditions can be checked by atoms IfThen2$(q_1, i, q_2, j, x_{i2}, x_{j1})$, for every $i, j \leq N$.

As we do not need any further auxiliary predicates, P_{query} is empty (but we kept it for symmetry reasons).

This completes the description of P. Note that P is non-recursive, and has polynomial size in the size of q and Σ. In order to finish the proof of part (a) of Theorem 3.1, we next explain how to reduce the arity of P.

This final step of the construction is based on two ideas.

First, by using Boolean variables and some new ternary relations, we can replace the 6-ary relation IfThen2 (and likewise the 4-ary relation IfThen). More precisely, we replace every atom IfThen2$(X_1, X_2, Y_1, Y_2, U_1, U_2)$ by a conjunction of the form

$$\begin{aligned}\text{IfEq}(X_1, X_2, B_1), \text{IfEq}(Y_1, Y_2, B_2), \text{IfEq}(U_1, U_2, B_3), \\ \text{NotB}(B_1, B_1'), \text{NotB}(B_2, B_2'), \text{OrB}(B_1, ', B_2', B_4), \\ \text{OrB}(B_3, B_4, B_5), \text{TrueB}(B_5).\end{aligned}$$

Here, NotB, OrB, are predicates that mimic Boolean gates, e.g., OrB(B_3, B_4, B_5) holds if B_5 is the Boolean Or of B_3 and B_4, in particular all values have to be from $\{0, 1\}$. TrueB(B_5) only holds if $B_5 = 1$. The predicate IfEq(X_1, X_2, B_1) holds if $B_1 = 1$ and $X_1 = X_2$ or if $B_1 = 0$ and $X_1 \neq X_2$. The relations IfEq, NotB, OrB, TrueB can easily be defined in P_{chase}.

The second idea is that T need not be materialized. We only materialize a relation T' of arity $a + 1$ which is intended to represent all database tuples. More precisely, $T'(j, X_1, \ldots, X_a)$ shall hold if $(X_1, \ldots, X_a)$ represents a tuple from relation R_j or if $j = 0$. Clearly, T' can be defined in P_{tuples}.

Every tuple $T(j, r_j, f_j, x_{j1}, \ldots, x_{ja}, s_j, c_{j1}, \ldots, c_{jk})$ in r_{tuples} is then replaced by a conjunction of atoms with the same semantics. The conjunct $T'(r_j', x_{j1}, \ldots, x_{ja})$ tests whether $(x_{j1}, \ldots, x_{ja})$ is in $R_{r_j'}$. Further atoms ensure that $r_j = r_j'$ if $f_j = 0$. Finally, it is ensured that, if $f_j = 1$ the

[6]This example tgd is not related to our running example as that does not have a single tgd with all features.

$$\text{IfThen2}(X, X, Y, Y, U, U) :- \text{DNum}(X), \text{DNum}(Y), \text{DNum}(U).$$

$$\text{IfThen2}(X_1, X_2, Y_1, Y_2, U_1, U_2) :- \text{DNum}(X_1), \text{DNum}(X_2), \text{DNum}(Y_1), \text{DNum}(Y_2), \text{DNum}(U_1), \text{DNum}(U_2), \text{Neq}(X_1, X_2).$$

$$\text{IfThen2}(X_1, X_2, Y_1, Y_2, U_1, U_2) :- \text{DNum}(X_1), \text{DNum}(X_2), \text{DNum}(Y_1), \text{DNum}(Y_2), \text{DNum}(U_1), \text{DNum}(U_2), \text{Neq}(Y_1, Y_2).$$

Figure 4: Three rules defining $\text{IfThen2}(X_1, X_2, Y_1, Y_2, U_1, U_2)$.

values are restricted as by the right-hand side of rule 1.

In order to prove part (b), we must get rid of the numeric domain (except for 0 and 1). This is actually very easy. We just replace each numeric value by a logarithmic number of bits (coded by our 0 and 1 domain elements), and extend the predicate arities accordingly. As a matter of fact, this requires an increase of arity by a factor of $\log N = \mathcal{O}(\log |q|)$. It is well-known that a successor predicate and a vectorized $<$ predicate for such bit-vectors can be expressed by a polynomially-sized nonrecursive Datalog program, see (Dantsin et al. 2001). The rest is completely analogous to the above proof. This concludes the proof sketch for Theorem 3.1.

We would like to conclude this section with some remarks:

Remark 1. Note that the evaluation complexity of the Datalog program obtained for case (b) is not significantly higher than the evaluation complexity of the program P constructed for case (a). For example, in the most relevant case of bounded arities, both programs can be evaluated in NPTIME combined complexity over a database D. In fact, it is well-known that the combined complexity of a Datalog program of bounded arity is in NPTIME (see (Dantsin et al. 2001)). But it is easy to see that if we expand the signature of such a program (and of the underlying database) by a logarithmic number of Boolean-valued argument positions (attributes), nothing changes, because the possible values for such vectorized arguments are still of polynomial size. It is just a matter of coding. In a similar way, the data complexity in both cases (a) and (b) is the same (PTIME).

Remark 2. It is easy to generalize this result to the setting where q is actually a union of conjunctive queries (UCQ).

Remark 3. The method easily generalizes to translate non-Boolean queries, i.e., queries with output, to polynomially-sized nonrecursive Datalog programs with output. We are here only interested in *certain answers* consisting of tuples of values from the original domain $dom(D)$ (see (Fagin et al. 2005)). Assume that the head of q is an atom $R(X_1, \ldots, X_k)$ where R is the output relation symbol, and the X_i are variables also occurring in the body of q. We then obtain a nonrecursive Datalog translation by acting as in the above proof, except for the following modifications. Make $R(X_1, \ldots, X_k)$ the head of rule r_{goal}, and add for $1 \leq i \leq k$ an atom $adom(X_i)$ to r_{query}, where $adom$ is an auxiliary predicate such that $adom(u)$ is iff u is in the *active non-numeric domain* of the database, that is, iff $u \in dom(D)$ and u effectively occurs in the database. It is easy to see that the auxiliary predicate $adom$ itself can be achieved via a nonrecursive Datalog program from D. Clearly, by construction of (the so modified) program P, the output of P are then precisely the certain answers of the query q.

Remark 4. While nonrecursive Datalog is strictly less expressive than first-order logic (FO), nonrecursive Datalog is, in general, a more succinct formalism. This means that certain FO-properties can be represented by exponentially smaller nonrecursive datalog programs than they can by classical FO formulas. The reason is that, unlike FO, nonrecursive Datalog allows one to define new relations that can be shared by various rule bodies (this is often referred to as *predicate sharing*). In FO, such shared definitions are not possible. Thus, it is, in general, not possible to translate a nonrecursive Datalog program in polynomial time into an equivalent FO formula over the same database D. However, it is not hard to see that the specific nonrecursive Datalog program P of the above proof actually *can* be transformed in polynomial time into an equivalent first-order query over D. In fact, for numerical databases, the generated program P is of bounded depth. This means that the "tree" of predicate definition dependencies is of bounded depth. We can thus replace multiple occurrences of the same predicate by multiple occurrences of a genuine FO-formula defining the same predicate. This needs to be done Inductively, but due to the bounded depth,it only yields a polynomial blow up. For non-numerical databases, in addition, we need to define a vectorized $<$-predicate on Boolean vectors of logarithmic length. This can indeed easily be done via a polynomially sized FO formula. In summary, under the premises of the above theorem, a polynomial translation of (Σ, q) into a plain FO formula (and thus into a plain SQL query rather than a SQL DL program with view definitions)of polynomial size is possible, and can easily be ontained from our nonrecursive Datalog program P. We believe, however, that, from an efficiency point of view, the translation into the nonrecursive program P is actually smarter: Why would one want to re-compute several times the same shared intensional relation?

4 Further Results Derived From the Main Theorem

We wish to mention some interesting consequences of the Main Theorem that follow easily from the above result after combining it with various other known results.

4.1 Linear TGDs

A linear tgd (Calì, Gottlob, and Lukasiewicz 2009) is one that has a single atom in its rule body. The class of linear tgds is a fundamental one in the Datalog$^{\pm}$ family. This class contains the class of *inclusion dependencies*. It was already shown in (Johnson and Klug 1984) for inclusion dependencies that classes of linear tgds of bounded (predicate) arities enjoy the PWP. That proof carries over to linear tgds, and we thus can state:

Lemma 4.1. *Classes of linear tgds of bounded arity enjoy the PWP.*

By Theorem 3.1, we then conclude:

Theorem 4.2. *Conjunctive queries under linear tgds of bounded arity are polynomially rewritable as nonrecursive Datalog programs in the same fashion as for the Main Theorem. So are sets of inclusion dependencies of bounded arity.*

4.2 DL-Lite and OWL 2 QL

A pioneering and highly significant contribution towards tractable ontological reasoning was the introduction of the *DL-Lite* family of description logics (DLs) by Calvanese et al. (Calvanese et al. 2007; Poggi et al. 2008). *DL-Lite* was further studied and developed in (Artale et al. 2009).

A DL-lite theory (or TBox) $\Sigma = (\Sigma^-, \Sigma^+)$ consists of a set of negative constraints Σ^- such as key and disjointness constraints, and of a set Σ^+ of positive constraints that resemble tgds. As shown in (Calvanese et al. 2007), the negative constraints Σ^- can be compiled into a polymomially sized first-order formula (actually a union of conjunctive queries) of the same arity as Σ^- such that for each database and BCQ q, $(D, \Sigma) \models q$ iff $D \not\models \Sigma^-$ and $(D, \Sigma^+) \models q$. In the full version of (Calì, Gottlob, and Lukasiewicz 2009) it was shown that for the main DL-Lite variants defined in (Calvanese et al. 2007), each Σ^+ can be immediately translated into an equivalent set of linear tgds of arity 2. By virtue of this, and the above we obtain the following theorem.

Theorem 4.3. *Let q be a CQ and let $\Sigma = (\Sigma^-, \Sigma^+)$ be a DL-Lite theory expressed in one of the following DL-Lite variants: DL-Lite$_{\mathcal{F},\sqcap}$, DL-Lite$_{\mathcal{R},\sqcap}$, DL-Lite$^+_{\mathcal{A},\sqcap}$, DLR-Lite$_{\mathcal{F},\sqcap}$, DLR-Lite$_{\mathcal{R},\sqcap}$, or DLR-Lite$^+_{\mathcal{A},\sqcap}$. Then Σ^+ can be rewritten into a nonrecursive Datalog program P such that for each database D, $(D, \Sigma^+) \models q$ iff $D \models P$. Regarding the arities of P, the same bounds as in the Main Theorem hold.*

The OWL-based query language OWL 2 QL, which we already mentioned in the introduction, is essentially a syntactic variant of *DL-Lite*$_{\mathcal{R}}$. Note that the former is used in a context where the unique name assumption (UNA) is not made, while the UNA is made by the DL-Lite logics, and, in particular, *DL-Lite*$_{\mathcal{R}}$. However, conjunctive query answering with or without adopting the UNA is equivalent. Therefore, Theorem 4.3 also extends to the case where Σ is an OWL 2 QL theory.

4.3 Sticky and Sticky Join TGDs

Sticky tgds (Calì, Gottlob, and Pieris 2010b) and sticky-join tgds (Calì, Gottlob, and Pieris 2010b) are special classes of tgds that generalize linear tgds but allow for a limited form of join (including as special case the cartesian product). They allow one to express natural ontological relationships not expressible in DLs such as OWL. We do not define these classes here, and refer the reader to (Calì, Gottlob, and Pieris 2011). By Theorem 3.24 of (Calì, Gottlob, and Pieris 2011), which will also be discussed in more detail in the upcoming journal version of the present paper, both classes enjoy the Polynomial Witness Property. By Theorem 3.1, we thus obtain the following result:

Theorem 4.4. *Conjunctive queries under sticky tgds and sticky-join tgds over a fixed signature $\mathcal{R}$ are rewritable into polynomially sized nonrecursive Datalog programs of arity bounded as in Theorem 3.1.*

5 Related Work on Query Rewriting

Our work fits into the framework of *ontology based data access (OBDA)* (Calvanese et al. 2007; Pérez-Urbina, Horrocks, and Motik 2009; Calvanese et al. 2011; Kikot, Kontchakov, and Zakharyaschev 2011; Gottlob, Orsi, and Pieris 2011; Orsi and Pieris 2011; Calì et al. 2010). In this framework, a classical relational database D, often stored in a relational DBMS, is enhanced by an ontological theory (or TBox) Σ. The database D is then queried "modulo Σ", that is, queries are evaluated over (the models of) $D \cup \Sigma$ rather than over D alone. The idea is, however, to exploit the underlying DBMS as much as possible, by *rewriting* both the given query q and the theory Σ into a query q' over D. After such a rewriting, the full power of existing DBMS, including sophisticated classical query optimization can be exploited for answering q'.

Several techniques for query-rewriting have been developed. We will briefly discuss some of these below. Let us note upfront that these previously existing techniques of query rewriting do not make any assumption about the existence of special constants 0 and 1 in the vocabulary of D, let alone about the availability of a numeric domain *Num* with an associated linear order $<$. However, these assumptions are not at all problematic, as all large-scale DBMS we know of, in particular, SQL-based systems, provide the type *integer* with appropriate arithmetic operations and comparison predicates for free. Our assumptions are thus perfectly fulfilled by existing DBMS.

An early algorithm, introduced in (Calvanese et al. 2007) and implemented in the QuOnto system[7], reformulates the given query into a union of CQs (UCQs) by means of a backward-chaining resolution procedure. The size of the computed rewriting increases exponentially w.r.t. the number of atoms in the given query. This is mainly due to the fact that unifications are derived in a "blind" way from every unifiable pair of atoms, even if the generated rule is superfluous. An alternative resolution-based rewriting technique was proposed by Peréz-Urbina et al. (Pérez-Urbina,

[7] http://www.dis.uniroma1.it/ quonto/

Motik, and Horrocks 2009), implemented in the Requiem system[8], that produces a UCQs as a rewriting which is, in general, smaller (but still exponential in the number of atoms of the query) than the one computed by QuOnto. This is achieved by avoiding many useless unifications, and thus the generation of redundant rules due to such unifications. This algorithm works also for more expressive non-first-order rewritable DLs. In this case, the computed rewriting is a (recursive) Datalog query. Following a more general approach, Calì et al. (Calì, Gottlob, and Pieris 2010a) proposed a backward-chaining rewriting algorithm for the first-order rewritable Datalog$^{\pm}$ languages mentioned above. However, this algorithm is inspired by the original QuOnto algorithm, and inherits all its drawbacks. In (Gottlob, Orsi, and Pieris 2011), a rewriting technique for linear Datalog$^{\pm}$ into unions of conjunctive queries is proposed. This algorithm is an improved version of the one already presented in (Calì, Gottlob, and Pieris 2010a), where further superfluous unifications are avoided, and where, in addition, redundant atoms in the body of a rule, that are logically implied (w.r.t. the ontological theory) by other atoms in the same rule, are eliminated. This elimination of body-atoms implies the avoidance of the construction of redundant rules during the rewriting process. However, the size of the rewriting is still exponential in the number of query atoms.

Of more interest to the present work are rewritings into nonrecursive Datalog. In (Kontchakov et al. 2010; 2011) a polynomial-size rewriting into nonrecursive Datalog is given for the description logics DL-Lite$^{\mathcal{F}}_{horn}$ and DL-Lite$_{horn}$. For DL-Lite$^{\mathcal{N}}_{horn}$, a DL with counting, a polynomial rewriting involving aggregate functions is proposed. It is, moreover, shown in (the full version of) (Kontchakov et al. 2010) that for the description logic DL-Lite$_{\mathcal{F}}$ a polynomial-size pure first-order query rewriting is possible. Note that neither of these logics allows for role inclusion, while our approach covers description logics with role inclusion axioms. Other results in (Kontchakov et al. 2010; 2011) are about *combined rewritings* where both the query and the database D have to be rewritten. A recent very interesting paper discussing polynomial size rewritings is (Kikot, Kontchakov, and Zakharyaschev 2011). Among other results, (Kikot, Kontchakov, and Zakharyaschev 2011) provides complexity-theoretic arguments indicating that without the use of special constants (e.g, 0 and 1, or the numerical domain), a polynomial rewriting such as ours may not be possible. Rosati et al. (Rosati and Almatelli 2010) recently proposed a very sophisticated rewriting technique into nonrecursive Datalog, implemented in the Presto system. This algorithm produces a non-recursive Datalog program as a rewriting, instead of a UCQs. This allows the "hiding" of the exponential blow-up inside the rules instead of generating explicitly the disjunctive normal form. The size of the final rewriting is, however, exponential in the number of non-eliminable existential join variables of the given query; such variables are a subset of the join variables of the query, and are typically less than the number of atoms in the query. Thus, the size of the rewriting is exponential in the query size in the worst case. Relevant further optimizations of this method are given in (Orsi and Pieris 2011). A very recent paper on query-rewritings in the OWL 2 QL context gives interesting and practically relevant sufficient conditions on queries and ontologies that guarantee small first-order rewritings (Kikot, Kontchakov, and Zakharyaschev 2012).

[8] http://www.comlab.ox.ac.uk/projects/requiem/home.html

6 Conclusion and Further Research

It was shown in this paper that for highly relevant description logics, ontological queries over a database extended by an ontological theory can be translated in polynomial time to classical queries over the original database.

As stated in the introduction, our results are so far of theoretical nature, and we do not claim as of now that they will lead to better practical algorithms. However, we do hope that this technique may soon lead to more efficient, and actually practical algorithms for query evaluation in the given context. We think that this is possible by applying further optimization techniques to the resulting non-recursive Datalog program. In fact, we deem it rewarding to identify appropriate further optimization techniques for decomposing the large rule body of the r_{goal} rule in the proof of Theorem 3.1, and to process this rule more efficiently. Our hope is that by applying smart splitting methods and/or decomposition techniques based on a structural (graph theoretic) analysis of this rule, and by defining a smart instantiation strategy based on an analysis of sideways information passing, as well as on classical join optimization techniques, it will be possible to come up with a practically efficient query rewriting.

To test such rewritings, they could be implemented on top of the existing system Nyaya (Virgilio et al. 2011), the current reference implementation system for the Datalog$^{\pm}$ language family. Nyaya already provides an efficient implementation[9] of reasoning procedures for several Datalog$^{\pm}$ languages (see e.g. (Gottlob, Orsi, and Pieris 2011)), and provides scalable persistent storage by exploiting the native referential partitioning techniques of Oracle 11g R2.

Acknowledgment G. Gottlob's work was funded by the EPSRC Grant EP/H051511/1 ExODA: Integrating Description Logics and Database Technologies for Expressive Ontology-Based Data Access. We thank the anonymous referees of the shorter DL'2011 version of this paper, as well as Roman Kontchakov, Carsten Lutz, and Michael Zakharyaschev for useful comments on an earlier version of this paper.

[9] A demo Nyaya is currently available online at http://www.nyaya.eu.

References

Acciarri, A.; Calvanese, D.; Giacomo, G. D.; Lembo, D.; Lenzerini, M.; Palmieri, M.; and Rosati, R. 2005. QuOnto: Querying ontologies. In *Proc. of AAAI*, 1670–1671.

Artale, A.; Calvanese, D.; Kontchakov, R.; and Zakharyaschev, M. 2009. The dl-lite family and relations. *J. Artif. Intell. Res. (JAIR)* 36:1–69.

Beeri, C., and Vardi, M. Y. 1981. The implication problem for data dependencies. In *Proc. of ICALP*, 73–85.

Calì, A.; Gottlob, G.; Lukasiewicz, T.; Marnette, B.; and Pieris, A. 2010. Datalog+/-: A family of logical knowledge representation and query languages for new applications. In *LICS*, 228–242. IEEE Computer Society.

Calì, A.; Gottlob, G.; and Kifer, M. 2008. Taming the infinite chase: Query answering under expressive relational constraints. In *Proc. of KR*, 70–80.

Calì, A.; Gottlob, G.; and Lukasiewicz, T. 2009. A general datalog-based framework for tractable query answering over ontologies. In *Proc. of PODS*, 77–86. Full version to appear in *Journal of Web Semantics*; currently available at http://dl.dropbox.com/u/472264/jws.pdf.

Calì, A.; Gottlob, G.; and Pieris, A. 2010a. Query rewriting under non-guarded rules. In *Proc. AMW*.

Calì, A.; Gottlob, G.; and Pieris, A. 2010b. Advanced processing for ontological queries. *PVLDB* 3(1):554–565.

Calì, A.; Gottlob, G.; and Pieris, A. 2010c. Query answering under non-guarded rules in datalog+/-. In *Proc. of RR*, 175–190.

Calì, A.; Gottlob, G.; and Pieris, A. 2011. Towards more expressive ontology languages: The query answering problem. Technical report, University of Oxford, Department of Computer Science. Submitted for publication - currently available at http://dl.dropbox.com/u/3185659/CGP.pdf.

Calvanese, D.; De Giacomo, G.; Lembo, D.; Lenzerini, M.; and Rosati, R. 2007. Tractable reasoning and efficient query answering in description logics: The DL-lite family. *J. Autom. Reasoning* 39(3):385–429.

Calvanese, D.; Giacomo, G. D.; Lembo, D.; Lenzerini, M.; Poggi, A.; Rodriguez-Muro, M.; Rosati, R.; Ruzzi, M.; and Savo, D. F. 2011. The mastro system for ontology-based data access. *Semantic Web* 2(1):43–53.

Chong, E. I.; Das, S.; Eadon, G.; and Srinivasan, J. 2005. An efficient sql-based rdf querying scheme. In Böhm, K.; Jensen, C. S.; Haas, L. M.; Kersten, M. L.; Larson, P.-Å.; and Ooi, B. C., eds., *VLDB*, 1216–1227. ACM.

Dantsin, E.; Eiter, T.; Georg, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.* 33(3):374–425.

Deutsch, A.; Nash, A.; and Remmel, J. B. 2008. The chase revisisted. In *Proc. of PODS*, 149–158.

Fagin, R.; Kolaitis, P. G.; Miller, R. J.; and Popa, L. 2005. Data exchange: Semantics and query answering. *Theor. Comput. Sci.* 336(1):89–124.

Gottlob, G., and Schwentick, T. 2011. Rewriting ontological queries into small nonrecursive datalog programs. In Rosati, R.; Rudolph, S.; and Zakharyaschev, M., eds., *Description Logics*, volume 745 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Gottlob, G.; Orsi, G.; and Pieris, A. 2011. Ontological queries: Rewriting and optimization. In *Proc. of ICDE*.

Grau, B. C.; Horrocks, I.; Motik, B.; Parsia, B.; Patel-Schneider, P.; and Sattler, U. 2008. Owl 2: The next step for owl. *Web Semantics: Science, Services and Agents on the World Wide Web* 6(4):309 – 322. ¡ce:title¿Semantic Web Challenge 2006/2007¡/ce:title¿.

Johnson, D. S., and Klug, A. C. 1984. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comput. Syst. Sci.* 28(1):167–189.

Kikot, S.; Kontchakov, R.; and Zakharyaschev, M. 2011. On (In)Tractability of OBDA with OWL2QL. In *Proc. DL 2011*.

Kikot, S.; Kontchakov, R.; and Zakharyaschev, M. 2012. Conjunctive Query Answering with OWL 2 QL. In *Proc. KR 2012, to appear*.

Kontchakov, R.; Lutz, C.; Toman, D.; Wolter, F.; and Zakharyaschev, M. 2010. The combined approach to query answering in dl-lite. In Lin, F.; Sattler, U.; and Truszczynski, M., eds., *KR*. AAAI Press.

Kontchakov, R.; Lutz, C.; Toman, D.; Wolter, F.; and Zakharyaschev, M. 2011. The combined approach to ontology-based data access. In *IJCAI*.

Maier, D.; Mendelzon, A. O.; and Sagiv, Y. 1979. Testing implications of data dependencies. *ACM Trans. Database Syst.* 4(4):455–469.

Orsi, G., and Pieris, A. 2011. Optimizing query answering under ontological constraints. *PVLDB* 4(11):1004–1015.

Pérez-Urbina, H.; Horrocks, I.; and Motik, B. 2009. Efficient query answering for owl 2. In Bernstein, A.; Karger, D. R.; Heath, T.; Feigenbaum, L.; Maynard, D.; Motta, E.; and Thirunarayan, K., eds., *International Semantic Web Conference*, volume 5823 of *Lecture Notes in Computer Science*, 489–504. Springer.

Pérez-Urbina, H.; Motik, B.; and Horrocks, I. 2009. Tractable query answering and rewriting under description logic constraints. *Journal of Applied Logic* 8(2):151–232.

Poggi, A.; Lembo, D.; Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Rosati, R. 2008. Linking data to ontologies. *J. Data Semantics* 10:133–173.

Rosati, R., and Almatelli, A. 2010. Improving query answering over DL-Lite ontologies. In *Proc. KR*.

Virgilio, R. D.; Orsi, G.; Tanca, L.; and Torlone, R. 2011. Semantic data markets: a flexible environment for knowledge management. In Macdonald, C.; Ounis, I.; and Ruthven, I., eds., *CIKM*, 1559–1564. ACM.