Efficiently Computable *Datalog*$^{\exists}$ Programs

Technical report containing the full version of the paper submitted to KR 2012

Nicola Leone and **Marco Manna** and **Giorgio Terracina** and **Pierfrancesco Veltri**
Department of Mathematics, University of Calabria, Italy
{leone,manna,terracina,veltri}@mat.unical.it

Abstract

Datalog$^{\exists}$ is the natural extension of *Datalog*, allowing existentially quantified variables in rule heads. This language is highly expressive and enables easy and powerful knowledge-modeling, but the presence of existentially quantified variables makes reasoning over *Datalog*$^{\exists}$ undecidable, in the general case. The results in this paper enable powerful, yet decidable and efficient reasoning (query answering) on top of *Datalog*$^{\exists}$ programs.

On the theoretical side, we define the class of parsimonious *Datalog*$^{\exists}$ programs, and show that it allows of decidable and efficiently-computable reasoning. Unfortunately, we can demonstrate that recognizing parsimony is undecidable. However, we single out *Shy*, an easily recognizable fragment of parsimonious programs, that significantly extends both *Datalog* and *Linear-Datalog*$^{\exists}$, while preserving the same (data and combined) complexity of query answering over *Datalog*, although the addition of existential quantifiers.

On the practical side, we implement a bottom-up evaluation strategy for *Shy* programs inside the DLV system, enhancing the computation by a number of optimization techniques to result in DLV$^{\exists}$ – a powerful system for answering conjunctive queries over *Shy* programs, which is profitably applicable to ontology-based query answering. Moreover, we carry out an experimental analysis, comparing DLV$^{\exists}$ against a number of state-of-the-art systems for ontology-based query answering. The results confirm the effectiveness of DLV$^{\exists}$, which outperforms all other systems in the benchmark domain.

1 Introduction

Context and Motivation. In the field of data and knowledge management, ontology-based Query Answering (QA) is becoming more and more a challenging task (Calvanese et al. 2007; Calì, Gottlob, and Lukasiewicz 2009; Kollia, Glimm, and Horrocks 2011; Calì, Gottlob, and Pieris 2011). Actually, database technology providers – such as Oracle[1], Ontotext[2] and Ontoprise[3] – have started to build ontological reasoning modules on top of their existing software. Also, ontological reasoning is part of several research-based systems, such as QuOnto (Acciarri et al. 2005), FaCT++ (Tsarkov and Horrocks 2006), and Nyaya (De Virgilio et al. 2011). In this context, queries are not merely evaluated on an extensional relational database D, but against a logical theory combining the database D with an *ontological theory* Σ. More specifically, Σ describes rules and constraints for inferring intensional knowledge from the extensional data stored in D (Johnson and Klug 1984). Thus, for a conjunctive query (CQ) q, we do not actually check whether D entails q, but we would like to know whether $D \cup \Sigma$ does.

A key issue in ontology-based QA is the design of the language that is provided for specifying the ontological theory Σ. This language should balance expressiveness and complexity, and in particular it should possibly be: (1) intuitive and easy-to-understand; (2) QA-decidable (i.e., QA should be decidable in this language); (3) efficiently computable; (4) powerful enough in terms of expressiveness; and (5) suitable for an efficient implementation.

In this regard, Datalog$^{\pm}$, the family of *Datalog*-based languages proposed by Calì, Gottlob, and Lukasiewicz (2009) for tractable query answering over ontologies, is arousing increasing interest (Mugnier 2011). This family, that encompasses and generalizes well known ontology specification languages, is mainly based on *Datalog*$^{\exists}$, the natural extension of *Datalog* (Abiteboul, Hull, and Vianu 1995) that allows $\exists$-quantified variables in rule heads. For example, the following *Datalog*$^{\exists}$ rules

```
∃Y father(X,Y) :- person(X).
person(Y) :- father(X,Y).
```

state that if X is a person, then X must have a father Y, which has to be a person as well. However, more in general, the Datalog$^{\pm}$ family intends to collect all expressive extensions of *Datalog* which are based on *tuple-generating dependencies* (or TGDs, which are *Datalog*$^{\exists}$ rules with possibly multiple atoms in rule heads), *equality-generating dependencies* and *negative constraint*. In particular, the "plus" symbol refers to any possible combination of these extensions, while the "minus" one imposes at least decidability, since *Datalog*$^{\exists}$ alone is already undecidable.

A number of QA-decidable Datalog$^{\pm}$ languages have been defined in the literature. They rely on three main paradigms, called *weak-acyclicity* (Fagin et al. 2005),

[1]See: http://www.oracle.com/

[2]See: http://www.ontotext.com/

[3]See: http://www.ontoprise.de/

guardness (Calì, Gottlob, and Kifer 2008) and *stickiness* (Calì, Gottlob, and Pieris 2010a), depending on syntactic properties. But there are also QA-decidable "abstract" classes of *Datalog*$^\exists$ programs, called *Finite-Expansion-Sets*, *Finite-Treewidth-Sets* and *Finite-Unification-Sets*, depending on semantic properties that capture the three mentioned paradigms, respectively (Mugnier 2011). However, even if all known languages based on these properties enjoy the simplicity of *Datalog* and are endowed with a number of properties that are desired for ontology specification languages, none of them fully satisfy conditions (1)–(5) above (see Section 8).

Contribution. In this work, we single out a new class of *Datalog*$^\exists$ programs, called *Shy*, which enjoys a new semantic property called *parsimony* and results in a powerful and yet QA-decidable ontology specification language that combines positive aspects of different Datalog$^\pm$ languages. With respect to properties (1)–(5) above, the class of *Shy* programs behaves as follows: (1) it inherits the simplicity and naturalness of *Datalog*; (2) it is QA-decidable; (3) it is efficiently computable (tractable data complexity and limited combined-complexity); (4) it offers a good expressive power being a strict superset of *Datalog*; and (5) it is suitable for an efficient implementation. Specifically, *Shy* programs can be evaluated by parsimonious forward-chaining inference that allows of an efficient on-the-fly QA, as witnessed by our experimental results.[4] From a technical viewpoint, the contribution of the paper is the following.

► We propose a new semantic property called *parsimony*, and prove that on the abstract class of parsimonious *Datalog*$^\exists$ programs, called *Parsimonious-Sets*, (atomic) query answering is decidable and also efficiently computable.

► After showing that recognition of parsimony is undecidable (**coRE**-complete), we single out *Shy*, a subclass of *Parsimonious-Sets*, which guarantees both easy recognizability and efficient answering even to conjunctive queries.

► We demonstrate that both *Parsimonious-Sets* and *Shy* preserve the same (data and combined) complexity of *Datalog* for atomic query answering: the addition of existential quantifiers does not bring any computational overhead here.

► We implement a bottom-up evaluation strategy for *Shy* programs inside the DLV system, and enhance the computation by a number of optimization techniques, yielding DLV$^\exists$ – a powerful system for query answering over *Shy* programs, which is profitably applicable for ontology-based query answering. To the best of our knowledge, DLV$^\exists$ is the first system supporting the standard first-order semantics for unrestricted CQs with existential variables over ontologies with advanced properties (some of these beyond $\mathbf{AC}_0$), such as, role transitivity, role hierarchy, role inverse, and concept products (Glimm et al. 2008).

► We perform an experimental analysis, comparing DLV$^\exists$ against a number of state-of-the-art systems for ontology-based QA. The positive results attained through this analysis do give clear evidence that DLV$^\exists$ is definitely the most effective system for query answering in dynamic environments, where the ontology is subject to frequent changes, making pre-computations and static optimizations inapplicable.

► We analyze related work, providing a precise taxonomy of the QA-decidable *Datalog*$^\exists$ classes. It turns out that both *Parsimonious-Sets* and *Shy* strictly contain *Datalog* $\cup$ *Linear-Datalog*$^\exists$, while they are uncomparable to *Finite-Expansion-Sets*, *Finite-Treewidth-Sets*, and *Finite-Unification-Sets*.

Organization. The remaining of the paper is organized as follows. Section 2 formally fixes syntax and semantics of *Datalog*$^\exists$ programs, as well as some preliminaries and useful notation. Section 3 defines a new class of *Datalog*$^\exists$ programs introducing a novel semantic property: *parsimony*. Section 4 presents the *Shy* language and its main properties. Section 5 deals with complexity. Sections 6 and 7 describe the DLV$^\exists$ system and our experimental analysis. Finally, section 8 surveys notable related works and discusses our results.

2 The Framework

In this section, after some useful preliminaries, we introduce *Datalog*$^\exists$ programs and CQs. Next, we equip such structures with a formal semantics. Finally, we show the *chase*, a well-known procedure that allows of answering CQs (Maier, Mendelzon, and Sagiv 1979; Johnson and Klug 1984).

2.1 Preliminaries

The following notation will be used throughout the paper. We always denote by Δ_C, Δ_N and Δ_V, countably infinite domains of *terms* called *constants*, *nulls* and *variables*, respectively; by Δ, the union of these three domains; by t, a generic *term*; by $\mathtt{c}$, $\mathtt{d}$ and $\mathtt{e}$, constants; by φ, a null; by $\mathtt{X}$ and $\mathtt{Y}$, variables; by $\mathbf{X}$ and $\mathbf{Y}$, sets of variables; by Π an alphabet of *predicate symbols* each of which, say $\mathtt{p}$, has a fixed nonnegative arity, denoted by $\mathsf{arity}(\mathtt{p})$; by $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, *atoms* being expressions of the form $\mathtt{p}(t_1, \ldots, t_k)$, where $\mathtt{p}$ is a predicate symbol and $t_1, \ldots, t_k$ is a *tuple* of terms. Moreover, if the tuple of an atom consists of only constants and nulls, then this atom is called *ground*; if $T \subseteq \Delta_C \cup \Delta_N$, then $\mathsf{base}(T)$ denotes the set of all ground atoms that can be formed with predicate symbols in Π and terms from T; if $\mathbf{a}$ is an atom, then $\mathsf{pred}(\mathbf{a})$ denotes the predicate symbol of $\mathbf{a}$; if ς is any formal structure containing atoms, then $\mathsf{terms}(\varsigma)$ (resp., $\mathsf{dom}(\varsigma)$) denotes all the terms from Δ (resp., $\Delta_C \cup \Delta_N$) occurring in the atoms of ς.

Mappings. Given a *mapping* $\mu : S_1 \to S_2$, its *restriction to* a set S is the mapping $\mu|_S$ from $S_1 \cap S$ to S_2 s.t. $\mu|_S(s) = \mu(s)$ for each $s \in S_1 \cap S$. If μ' is a restriction of μ, then μ is called an *extension* of μ', also denoted by $\mu \supseteq \mu'$. Let $\mu_1 : S_1 \to S_2$ and $\mu_2 : S_2 \to S_3$ be two mappings. We denote by $\mu_2 \circ \mu_1 : S_1 \to S_3$ the *composite* mapping.

We call *homomorphism* any mapping $h : \Delta \to \Delta$ whose restriction $h|_{\Delta_C}$ is the identity mapping. In particular, h is an homomorphism from an atom $\mathbf{a} = \mathtt{p}(t_1, \ldots, t_k)$ to an

[4] Intuitively, parsimonious inference generates no isomorphic atoms (see Section 3); while on-the-fly QA does not need any preliminary materialization or compilation phase (see Section 7), and is very well suited for QA against frequently changing ontologies.

atom $\mathbf{b}$ if $\mathbf{b} = \mathtt{p}(h(t_1), \ldots, h(t_k))$. With a slight abuse of notation, $\mathbf{b}$ is denoted by $h(\mathbf{a})$. Similarly, h is a homomorphism from a set of atoms S_1 to another set of atoms S_2 if $h(\mathbf{a}) \in S_2$, for each $\mathbf{a} \in S_1$. Moreover, $h(S_1) = \{h(\mathbf{a}) : \mathbf{a} \in S_1\} \subseteq S_2$. In particular, if $S_1 = \emptyset$, then $h(S_1) = \emptyset$. In case the domain of h is the empty set, then h is called *empty homomorphism* and it is denoted by $h_\emptyset$. In particular, $h_\emptyset(\mathbf{a}) = \mathbf{a}$, for each atom $\mathbf{a}$.

An *isomorphism* between two atoms (or two sets of atoms) is a bijective homomorphism. Given two atoms $\mathbf{a}$ and $\mathbf{b}$, we say that: $\mathbf{a} \preceq \mathbf{b}$ iff there is a homomorphism from $\mathbf{b}$ to $\mathbf{a}$; $\mathbf{a} \simeq \mathbf{b}$ iff there is an isomorphism between $\mathbf{a}$ and $\mathbf{b}$; $\mathbf{a} \prec \mathbf{b}$ iff $\mathbf{a} \preceq \mathbf{b}$ holds but $\mathbf{a} \simeq \mathbf{b}$ does not.

A *substitution* is a homomorphism σ from Δ to $\Delta_C \cup \Delta_N$ whose restriction $\sigma|_{\Delta_C \cup \Delta_N}$ is the identity mapping. Also, $\sigma_\emptyset = h_\emptyset$ denotes the empty substitution.

2.2 Programs and Queries

A *Datalog*$^\exists$ *rule* r is a finite expression of the form:

$$\forall \mathbf{X} \exists \mathbf{Y} \ \ \mathbf{atom}_{[\mathbf{X}' \cup \mathbf{Y}]} \leftarrow \mathbf{conj}_{[\mathbf{X}]} \tag{1}$$

where (i) $\mathbf{X}$ and $\mathbf{Y}$ are disjoint sets of variables (next called $\forall$-variables and $\exists$-variables, respectively); (ii) $\mathbf{X}' \subseteq \mathbf{X}$; (iii) $\mathbf{atom}_{[\mathbf{X}' \cup \mathbf{Y}]}$ stands for an atom containing only and all the variables in $\mathbf{X}' \cup \mathbf{Y}$; and (iv) $\mathbf{conj}_{[\mathbf{X}]}$ stands for a *conjunct* (a conjunction of zero, one or more atoms) containing only and all the variables in $\mathbf{X}$. Constants are also allowed in r. In the following, $\mathsf{head}(r)$ denotes $\mathbf{atom}_{[\mathbf{X}' \cup \mathbf{Y}]}$, and $\mathsf{body}(r)$ the set of atoms in $\mathbf{conj}_{[\mathbf{X}]}$. Universal quantifiers are usually omitted to lighten the syntax, while existential quantifiers are omitted only if $\mathbf{Y}$ is empty. In the second case, r coincides with a standard *Datalog* rule. If $\mathsf{body}(r) = \emptyset$, then r is usually referred to as a *fact*. In particular, r is called *existential* or *ground* fact according to whether r contains some $\exists$-variable or not, respectively. A *Datalog*$^\exists$ program P is a finite set of *Datalog*$^\exists$ rules. We denote by $\mathsf{preds}(P) \subseteq \Pi$ the predicate symbols occurring in P, by $\mathsf{data}(P)$ all the atoms constituting the ground facts of P, and by $\mathsf{rules}(P)$ all the rules of P being not ground facts.

Example 2.1. The following expression is a *Datalog*$^\exists$ rule where $\mathbf{father}$ is the head and $\mathbf{person}$ the only body atom.

```
∃Y father(X,Y) :- person(X).
```
□

Given a *Datalog*$^\exists$ program P, a *conjunctive query* (CQ) q over P is a first-order (FO) expression of the form:

$$\exists \mathbf{Y} \ \mathbf{conj}_{[\mathbf{X} \cup \mathbf{Y}]} \tag{2}$$

where $\mathbf{X}$ are its free variables, and $\mathbf{conj}_{[\mathbf{X} \cup \mathbf{Y}]}$ is a conjunct containing only and all the variables in $\mathbf{X} \cup \mathbf{Y}$ and possibly some constants. To highlight the free variables, we write $q(\mathbf{X})$ instead of q. Query q is called *Boolean CQ* (BCQ) if $\mathbf{X} = \emptyset$. Moreover, q is called *atomic* if $\mathbf{conj}$ is an atom. Finally, $\mathsf{atoms}(q)$ denotes the set of atoms in $\mathbf{conj}$.

Example 2.2. The following expression is a CQ asking whether there exists a grandfather having `john` as nephew.

```
∃Y father('john',X),father(X,Y)
```
□

2.3 Query Answering and Universal Models

In the following, we equip *Datalog*$^\exists$ programs and queries with a formal semantics to result in a formal QA definition.

Given a set S of atoms and an atom $\mathbf{a}$, we say that $S \models \mathbf{a}$ (resp., $S \Vdash \mathbf{a}$) holds if there is a substitution σ s.t. $\sigma(\mathbf{a}) \in S$ (resp., a homomorphism h s.t. $h(\mathbf{a}) \in S$).

Let $P \in$ *Datalog*$^\exists$. A set $M \subseteq \mathsf{base}(\Delta_C \cup \Delta_N)$ is a *model* for P ($M \models P$, for short) if, for each $r \in P$ of the form (1), whenever there exists a substitution σ s.t. $\sigma(\mathsf{body}(r)) \subseteq M$, then $M \models \sigma|_{\mathbf{X}}(\mathsf{head}(r))$. (Note that, $\sigma|_{\mathbf{X}}(\mathsf{head}(r))$ contains only and all the $\exists$-variables $\mathbf{Y}$ of r.) The set of all the models of P are denoted by $\mathsf{mods}(P)$.

Let $M \in \mathsf{mods}(P)$. A BCQ q is *true* w.r.t. M ($M \models q$) if there is a substitution σ s.t. $\sigma(\mathsf{atoms}(q)) \subseteq M$. Analogously, the answer of a CQ $q(\mathbf{X})$ w.r.t. M is the set $\mathsf{ans}(q, M) = \{\sigma|_{\mathbf{X}} : \sigma \text{ is a substitution } \wedge\ M \models \sigma|_{\mathbf{X}}(q)\}$.

The answer of a CQ $q(\mathbf{X})$ w.r.t. a program P is the set $\mathsf{ans}_P(q) = \{\sigma : \sigma \in \mathsf{ans}(q, M)\ \forall M \in \mathsf{mods}(P)\}$. Note that, $\mathsf{ans}_P(q) = \{\sigma_\emptyset\}$ iff q is a BCQ. In this case, we say that q is *cautiously true* w.r.t. P or, equivalently, that q is *entailed* by P. This is denoted by $P \models q$, for short.

Let $\mathcal{C}$ be a class of *Datalog*$^\exists$ programs. The following definition formally fixes the computational problem studied in this paper, concerning query answering.

Definition 2.3. $\mathrm{QA}_{[\mathcal{C}]}$ is the following decision problem. Given a program P belonging to $\mathcal{C}$, an atomic query q, and a substitution σ for q, does σ belong to $\mathsf{ans}_P(q)$? □

In the following, a *Datalog*$^\exists$ class $\mathcal{C}$ is called QA-decidable if and only if problem $\mathrm{QA}_{[\mathcal{C}]}$ is decidable. Finally, before concluding this section, we mention that QA can be carried out by using a universal model. Actually, a model U for P is called *universal* if, for each $M \in \mathsf{mods}(P)$, there is a homomorphism h s.t. $h(U) \subseteq M$.

Proposition 2.4 (Fagin et al. 2005)**.** *Let U be a universal model for P. Then, (i) $P \models q$ iff $U \models q$, for each BCQ q; (ii) $\mathsf{ans}_P(q) \subseteq \mathsf{ans}(q, U)$ for each CQ q; and (iii) $\sigma \in \mathsf{ans}_P(q)$ iff both $\sigma \in \mathsf{ans}(q, U)$ and $\sigma : \Delta_V \to \Delta_C$.*

2.4 The Chase

As already mentioned, the chase is a well-known procedure for constructing a universal model for a *Datalog*$^\exists$ program. We are now ready to show how this procedure works, in one of its variants (although slightly revised).

First, we introduce the notion of *chase step*, which, intuitively, *fires* a rule r on a set C of atoms for inferring new knowledge. More precisely, given a rule r of the form (1) and a set C of atoms, a *firing* substitution σ for r w.r.t. C is a substitution σ on $\mathbf{X}$ s.t. $\sigma(\mathsf{body}(r)) \subseteq C$. Next, given a firing substitution σ for r w.r.t. C, the *fire* of r on C due to σ infers $\hat{\sigma}(\mathsf{head}(r))$, where $\hat{\sigma}$ is an extension of σ on $\mathbf{Y} \cup \mathbf{X}$ associating each $\exists$-variable in $\mathbf{Y}$ to a different null. Finally, Procedure 1 illustrates the overall *restricted chase procedure*. Importantly, we assume that different fires (on the same or different rules) always introduce different "fresh" nulls. The procedure consists of an exhaustive series of fires in a breadth-first (level-saturating) fashion, which leads as result to a (possibly infinite) $\mathsf{chase}(P)$.

Procedure 1 CHASE(P)

Input: *Datalog*$^\exists$ program P
Output: A Universal Model $\mathsf{chase}(P)$ for P
1. $C := \mathsf{data}(P)$
2. *NewAtoms* $:= \emptyset$
3. **for each** $r \in P$ **do**
4. **for each** firing substitution σ for r w.r.t. C **do**
5. **if** ($(C \cup \mathit{NewAtoms}) \not\models \sigma(\mathsf{head}(r))$)
6. $\mathsf{add}(\hat{\sigma}(\mathsf{head}(r)), \mathit{NewAtoms})$
7. **if** ($\mathit{NewAtoms} \neq \emptyset$)
8. $C := C \cup \mathit{NewAtoms}$
9. **go to** step 2
10. **return** C

The *level* of an atom in $\mathsf{chase}(P)$ is inductively defined as follows. Each atom in $\mathsf{data}(P)$ has level 0. The level of each atom constructed after the application of a restricted chase step is obtained from the highest level of the atoms in $\sigma(\mathsf{body}(r))$ plus one. For each $k \geq 0$, $\mathsf{chase}^k(P)$ denotes the subset of $\mathsf{chase}(P)$ containing only and all the atoms of level up to k. Actually, by Procedure 1, $\mathsf{chase}^k(P)$ is precisely the set of atoms which is inferred the k^{th}-time that the outer for-loop is ran.

Proposition 2.5. *(Fagin et al. 2005; Deutsch, Nash, and Remmel 2008) Given a Datalog*$^\exists$ *program* P, CHASE *constructs a universal model for* P.

Unfortunately, CHASE does not always terminates.

Proposition 2.6. *(Fagin et al. 2005; Deutsch, Nash, and Remmel 2008)* $\mathrm{QA}_{[\mathit{Datalog}^\exists]}$ *is undecidable even for atomic queries. In particular, it is* ***RE****-complete.*

3 A New QA-Decidable *Datalog*$^\exists$ Class

This section introduces a new class of *Datalog*$^\exists$ programs, called *Parsimonious-Sets*, as well as some of its properties.

Definition 3.1. For any *Datalog*$^\exists$ program P, *parsimonious chase* (PARSIM-CHASE(P) for short) is the procedure resulting by the replacement of operator $\not\models$ by $\nVdash$ in the condition of the if-instruction at step 5 in Procedure 1 CHASE(P). The output of PARSIM-CHASE(P) is denoted by $\mathsf{pChase}(P)$. □

Note that, differently from $\mathsf{chase}(P)$, here $\mathsf{pChase}(P)$ might not be a model any more. Based on Definition 3.1, we next define a new class of *Datalog*$^\exists$ programs depending on a novel semantic property, called *parsimony*.

Definition 3.2. A *Datalog*$^\exists$ program P is called *parsimonious* if $\mathsf{pChase}(P) \Vdash \mathbf{a}$, for each $\mathbf{a} \in \mathsf{chase}(P)$. *Parsimonious-Sets* next denotes the class of all parsimonious programs. □

We next show that atomic QA against a *Parsimonious-Sets* program can be carried out by the PARSIM-CHASE algorithm.

Proposition 3.3. *Algorithm* PARSIM-CHASE *over parsimonious programs is sound and complete w.r.t. atomic QA.*

Proof. Soundness follows, by Definition 3.1, since $\mathsf{pChase}(P) \subseteq \mathsf{chase}(P)$ holds. In fact, since each substitution is a homomorphism, then, given a set of atoms S and an atom $\mathbf{a}$, $S \models \mathbf{a}$ always entails $S \Vdash \mathbf{a}$. Conversely, $S \nVdash \mathbf{a}$ always entails $S \not\models \mathbf{a}$. Finally, $\mathsf{ans}(q, \mathsf{pChase}(P)) \subseteq \mathsf{ans}_P(q)$, for each CQ q.

For completeness, let P be a parsimonious program and q be an atomic query. To prove that $\mathsf{ans}_P(q) \subseteq \mathsf{ans}(q, \mathsf{pChase}(P))$ we observe that whenever $\sigma \in \mathsf{ans}_P(q)$, then $\mathsf{chase}(P) \models \sigma(q)$, namely there is a substitution σ' such that $\sigma'(\sigma(q)) \in \mathsf{chase}(P)$. But, by Definition 3.2, $\mathsf{pChase}(P) \Vdash \sigma'(\sigma(q))$, namely there is a homomorphism h such that $h(\sigma'(\sigma(q))) \in \mathsf{pChase}(P)$. Now, since each substitution is a homomorphism and since composition of homomorphisms is a homomorphism, we call h' the homomorphism $h \circ \sigma'$. Thus, $h'(\sigma(q)) \in \mathsf{pChase}(P)$. But, since $h' = h \circ \sigma'$ is actually a substitution, then $\mathsf{pChase}(P) \models \sigma(q)$, namely $\sigma \in \mathsf{ans}(q, \mathsf{pChase}(P))$. □

Now, before proving one of the main results of this section concerning decidability of atomic query answering against parsimonious programs, we show that the cardinality of $\mathsf{pChase}(P)$ is finite as well as the number of levels reached by PARSIM-CHASE.

Lemma 3.4. *Let* P *be a Datalog*$^\exists$ *program,* α *be the maximum arity over all predicate symbols in* P*, and* Φ *be a set of* α *nulls. Then, there is a one-to-one correspondence* μ *between* $\mathit{pChase}(P)$ *and a subset of* $\mathsf{base}(\mathsf{dom}(P) \cup \Phi)$ *such that* $\mathbf{a} \simeq \mu(\mathbf{a})$*, for each* $\mathbf{a} \in \mathit{pChase}(P)$.

Proof. First we observe that each atom in $\mathsf{pChase}(P)$, say $\mathbf{a}$, has at most α different nulls. Thus, after replacing the nulls of $\mathbf{a}$ with different nulls from Φ we obtain an isomorphic atom belonging to $\mathsf{base}(\mathsf{dom}(P) \cup \Phi)$. Now assume that two atoms $\mathbf{a}_1 \neq \mathbf{a}_2$ in $\mathsf{pChase}(P)$ had one common isomorph $\mathbf{b} \in \mathsf{base}(\mathsf{dom}(P) \cup \Phi)$, namely $\mathbf{a}_1 \simeq \mathbf{b}$ and $\mathbf{a}_2 \simeq \mathbf{b}$. This would clearly entail that $\mathbf{a}_1 \simeq \mathbf{a}_2$. But this is not possible since $\mathsf{data}(P)$ contains no pair of isomorphic atoms, and because PARSIM-CHASE (due to the introduction of operator $\Vdash$) does not allow any addition to $\mathsf{pChase}(P)$ of an isomorphic atom. Consequently, μ can be built by associating to each atom in $\mathsf{pChase}(P)$ one of its isomorphic atoms in $\mathsf{base}(\mathsf{dom}(P) \cup \Phi)$. □

Corollary 3.5. *Let* P *be a Datalog*$^\exists$ *program, and* α *be the maximum arity over all predicate symbols in* P*. Then,* $|\mathit{pChase}(P)| \leq |\mathit{preds}(P)| \cdot (|\mathit{dom}(P)| + \alpha)^\alpha$.

Proof. This upperbound directly follows from Lemma 3.4 by considering the cardinality of $\mathsf{base}(\mathsf{dom}(P) \cup \Phi)$, where Φ is a set of α nulls. □

The following theorem claims that parsimony makes atomic query answering decidable.

Theorem 3.6. *Atomic query answering against Parsimonious-Sets programs is decidable.*

Proof. Proposition 3.3 ensures that atomic QA is sound and complete against $\mathsf{pChase}(P)$. Corollary 3.5 ensures that the cardinality of $\mathsf{pChase}(P)$ is finite, entailing that both PARSIM-CHASE stops after computing no more that

Algorithm 2 ORACLE-QA(P, q)

Input: *Datalog*$^\exists$ program $P \wedge$ Boolean atomic query q
Output: `true` $\vee$ `false`
1. **if** (IS-PARSIMONIOUS(P))
2. **return** ($\mathsf{pChase}(P) \models q$)
3. **else**
4. $k := \mathsf{firstAwakeningLevel}(P)$
5. $P' := P \cup (\mathsf{chase}^k(P) - \mathsf{chase}^{k-1}(P))$
6. **return** ORACLE-QA(P', q)

$|\mathsf{pChase}(P)|$ levels, and the number of firing substitutions considered at step 4 of the algorithm is always finite. □

We now show that recognizing parsimony is undecidable.

Theorem 3.7. *Checking whether a program is parsimonious is not decidable. In particular, it is* ***coRE****-complete.*

Proof. For the membership, given a *Datalog*$^\exists$ program P, we show that one can semi-decide whether P is not parsimonious. In fact, in such a case, there must exist by definition a level k such that, for each atom $\mathbf{a} \in \mathsf{chase}^k(P)$, $\mathsf{chase}^{k-1}(P) \Vdash \mathbf{a}$ but there is an atom $\mathbf{a}' \in \mathsf{chase}^{k+1}(P)$ such that $\mathsf{chase}^k(P) \not\Vdash \mathbf{a}'$. Thus, if a program is not parsimonious, then we can discover that by running the CHASE.

For the hardness part, we use Algorithm 2, called ORACLE-QA, that would solve the $\text{QA}_{[\textit{Datalog}^\exists]}$ problem (which, by Proposition 2.6, is **RE**-complete) if the problem of checking whether a program is parsimonious was decidable. In particular, given a *Datalog*$^\exists$ program P, we denote by IS-PARSIMONIOUS the Boolean terminating function deciding whether P is parsimonious or not; and by $\mathsf{firstAwakeningLevel}(P)$ the lowest level k reached by the CHASE such that $\mathsf{pChase}(P) \Vdash \mathbf{a}$ for each $\mathbf{a} \in \mathsf{chase}^{k-1}(P)$, and $\mathsf{pChase}(P) \not\Vdash \mathbf{a}$ for at least one $\mathbf{a} \in \mathsf{chase}^k(P)$. Finally, it is enough to show that the algorithm: (i) is sound, since P' only contains atoms from $\mathsf{chase}(P)$; (ii) is complete, since P' evolves to a parsimonious program after each execution of instruction 5 adding to P' at least one atom $\mathbf{a}$ such that $\mathsf{chase}^{k-1}(P) \not\Vdash \mathbf{a}$; (iii) terminates, since the cardinality of $\mathsf{pChase}(P)$ is finite (where P denotes the initial program), entailing that at most $|\mathsf{pChase}(P)|$ recursive calls can be activated. □

4 Recognizable Parsimonious Programs

We next define a novel syntactic *Datalog*$^\exists$ class: *Shy*. Later, we prove that this class enjoys the parsimony property.

4.1 *Shy*: Definition and Main Properties

Calì, Gottlob, and Kifer (2008) introduced the notion of "affected position" to know whether an atom with a null at a given position might belong to the output of the CHASE. Specifically, let $\mathbf{a}$ be an atom of arity k with a variable $\mathtt{X}$ occurring at position $i \in [1..k]$. Position i of $\mathbf{a}$ is marked as *affected* w.r.t. P if there is a rule $r \in P$ s.t. $\mathsf{pred}(\mathsf{head}(r)) = \mathsf{pred}(\mathbf{a})$ and $\mathtt{X}$ is either an $\exists$-variable, or a $\forall$-variable s.t. $\mathtt{X}$ occurs in the body of r in affected positions only. Otherwise, position i is definitely marked as *unaffected*. However, this procedure might mark as affected some position hosting a variable that can never be mapped to nulls.

To better detect whether a program admits a firing substitution that maps a $\forall$-variable into a null, we introduce the notion of *null-set* of a position in an atom. More precisely, $\varphi^r_{\mathtt{X}}$ denotes the "representative" null that can be introduced by the $\exists$-variable $\mathtt{X}$ occurring in rule r. (If $(r, \mathtt{X}) \neq (r', \mathtt{X}')$, then $\varphi^r_{\mathtt{X}} \neq \varphi^{r'}_{\mathtt{X}'}$.)

Definition 4.1. Let P be a *Datalog*$^\exists$ program, $\mathbf{a}$ be an atom, and $\mathtt{X}$ a variable occurring in $\mathbf{a}$ at position i. The *null-set* of position i in $\mathbf{a}$ w.r.t. P, denoted by $\mathsf{nullset}(i, \mathbf{a})$, is inductively defined as follows. If $\mathbf{a}$ is the head atom of some rule $r \in P$, then $\mathsf{nullset}(i, \mathbf{a})$ is: (1) either the set $\{\varphi^r_{\mathtt{X}}\}$, if $\mathtt{X}$ is $\exists$-quantified in r; or (2) the intersection of every $\mathsf{nullset}(j, \mathbf{b})$ s.t. $\mathbf{b} \in \mathsf{body}(r)$ and $\mathtt{X}$ occurs at position j in $\mathbf{b}$, if $\mathtt{X}$ is $\forall$-quantified in r. If $\mathbf{a}$ is not a head atom, then $\mathsf{nullset}(i, \mathbf{a})$ is the union of $\mathsf{nullset}(i, \mathsf{head}(r))$ for each $r \in P$ s.t. $\mathsf{pred}(\mathsf{head}(r)) = \mathsf{pred}(\mathbf{a})$. □

Note that $\mathsf{nullset}(i, \mathbf{a})$ may be empty. A representative null φ *invades* a variable $\mathtt{X}$ that occurs at position i in an atom $\mathbf{a}$ if φ is contained in $\mathsf{nullset}(i, \mathbf{a})$. A variable $\mathtt{X}$ occurring in a conjunct $\mathbf{conj}$ is *attacked* in $\mathbf{conj}$ by a null φ if each occurrence of $\mathtt{X}$ in $\mathbf{conj}$ is invaded by φ. A variable $\mathtt{X}$ is *protected* in $\mathbf{conj}$ if it is attacked by no null. Clearly, each attacked variable is affected but the converse is not true.

We are now ready to define the new *Datalog*$^\exists$ class.

Definition 4.2. A rule r of a *Datalog*$^\exists$ program P is called *shy* w.r.t. P if the following conditions are both satisfied:

1. If a variable $\mathtt{X}$ occurs in more than one body atom, then $\mathtt{X}$ is protected in $\mathsf{body}(r)$;
2. If two distinct $\forall$-variables are not protected in $\mathsf{body}(r)$ but occur both in $\mathsf{head}(r)$ and in two different body atoms, then they are not attacked by the same null.

Finally, *Shy* denotes the class of all *Datalog*$^\exists$ programs containing only shy rules. □

After noticing that a program is *Shy* regardless its ground facts, we give an example of program being not *Shy*.

Example 4.3. Let P be the following *Datalog*$^\exists$ program:

```
r1 :  ∃Y u(X,Y) :- q(X).
r2 :  v(X,Y,Z) :- u(X,Y), p(X,Z).
r3 :  p(X,Y) :- v(X,Y,Z).
r4 :  u(Y,X) :- u(X,Y).
```

Let $\mathbf{a}_1, \ldots, \mathbf{a}_9$ be the atoms of P in left-to-right/top-to-bottom order. First, $\mathsf{nullset}(2, \mathbf{a}_1) = \{\varphi^{r_1}_{\mathtt{Y}}\}$. Next, this singleton is propagated (head-to-body) to $\mathsf{nullset}(2, \mathbf{a}_4)$ and $\mathsf{nullset}(2, \mathbf{a}_9)$. At this point, from $\mathbf{a}_9$ the singleton is propagated (body-to-head) to $\mathsf{nullset}(1, \mathbf{a}_8)$, and from $\mathbf{a}_4$ to $\mathsf{nullset}(2, \mathbf{a}_3)$, and so on, according to Definition 4.1. Finally, even if $\mathtt{X}$ is protected in r_2 since it is invaded only in $\mathbf{a}_4$, rule r_2, and therefore P, is not shy due to $\mathtt{Y}$ and $\mathtt{Z}$ that are attacked by $\varphi^{r_1}_{\mathtt{Y}}$ and occur in $\mathsf{head}(r_2)$. Moreover, it is easy to verify that P plus any fact for $\mathtt{q}$ does not belong to *Parsimonious-Sets*. □

Intuitively, the key idea behind this class is as follows. If a program is shy then, during a CHASE execution, nulls do

not meet each other to join but only to propagate. Moreover, a null is propagated, during a given fire, from a single atom only. Hence, the *shyness* property, which ensures parsimony.

Theorem 4.4. *Shy $\subset$ Parsimonious-Sets.*

Proof. Let P be a *Shy* program. Assume there exists a level k such that $\mathsf{pChase}(P) \Vdash \mathbf{a}$ for each $\mathbf{a} \in \mathsf{chase}^{k-1}(P)$, and $\mathsf{pChase}(P) \not\Vdash \mathbf{b}$ for at least one atom $\mathbf{b} \in \mathsf{chase}^{k}(P)$. Let $j < k-1$ be the level where PARSIM-CHASE has stopped. Since $\mathbf{b} \in \mathsf{chase}^{k}(P) - \mathsf{chase}^{k-1}(P)$, then there must be at least one atom in $\mathsf{chase}^{k-1}(P) - \mathsf{chase}^{k-2}(P)$ that is necessary for firing a rule r to $\mathsf{chase}^{k-1}(P)$ to infer $\mathbf{b}$. Let σ be the firing substitution for r w.r.t. $\mathsf{chase}^{k-1}(P)$ used for inferring $\mathbf{b}$, and $\mathbf{a}_1, \ldots, \mathbf{a}_n$ be the body atoms of r. Clearly, $\mathsf{pChase}(P) \Vdash \sigma(\mathbf{a}_i)$ for each $i \in [1..n]$. Now, since P is shy then, by Definition 3.2, σ may map a variable into a null only if such a variable does not appear in two different atoms, and two different variables appearing in the head cannot be mapped to the same null. This means that if we consider the n homomorphisms $h_1, \ldots, h_n$ such that $h_i(\sigma(\mathbf{a}_i)) \in \mathsf{pChase}(P)$ for each $i \in [1..n]$, then we can take the union h of their restrictions on the $\exists$-variables of r without generating any conflict. But this is not possible because $h \circ \sigma$ is also a firing substitution for r on $\mathsf{pChase}(P)$ entailing the existence of an homomorphism from $\sigma(\mathsf{head}(r))$ to $h(\sigma(\mathsf{head}(r)))$. Finally, this entails an homomorphism from $\mathbf{b}$ to the atom inferred by the extension of $h \circ \sigma$. □

Corollary 4.5. *Atomic QA over Shy is decidable.*

We now show that recognizing parsimony is decidable.

Theorem 4.6. *Checking whether a program P is shy is decidable. In particular, it is doable in polynomial-time.*

Proof. First, the occurrences of $\exists$-variables in P fix the number h of nulls appearing in the null-sets of P. Next, let k be the number of atoms occurring in P, and α be the maximum arity over all predicate symbols in P. It is enough to observe that P allows at most $k * \alpha$ null-sets each of which of cardinality no greater than h. Finally, the statement holds since the null-set-construction is monotone and stops as soon as a fixpoint has been reached. □

4.2 Conjunctive Queries over *Shy*

In this section we show that conjunctive QA against *Shy* programs is also decidable. To manage CQs, we next describe a technique called parsimonious-chase resumption, which is sound for any *Datalog*$^{\exists}$ program P, and also complete over *Shy*. Before proving formal results, we give a brief intuition of this approach. Assume that $\mathsf{pChase}(P)$ consists of the atoms $\mathtt{p}(\mathtt{c}, \varphi)$, $\mathtt{q}(\mathtt{d}, \mathtt{e})$, $\mathtt{r}(\mathtt{c}, \mathtt{e})$. It is definitely possible that $\mathsf{chase}(P)$ contains also $\mathtt{q}(\varphi, \mathtt{e})$, which, of course, cannot belong to $\mathsf{pChase}(P)$ due to $\mathtt{q}(\mathtt{d}, \mathtt{e})$. Now consider the CQ $q = \exists \mathtt{Y}\ \mathtt{p}(\mathtt{X}, \mathtt{Y}), \mathtt{q}(\mathtt{Y}, \mathtt{Z})$. Clearly, $\mathsf{pChase}(P)$ does not provide any answer to q even if P does. Let us both "promote" φ to constant in Δ_C, and "resume" the PARSIM-CHASE execution at step 3, in the same state in which it had stopped after returning the set C at step 10. But, now, since φ can be considered as a constant, then there is no homomorphism from $\mathtt{q}(\varphi, \mathtt{e})$ to $\mathtt{q}(\mathtt{d}, \mathtt{e})$. Thus, $\mathtt{q}(\varphi, \mathtt{e})$ may be now inferred by the algorithm and used to prove that $\mathsf{ans}_P(q)$ is nonempty.

We call *freeze* the act of promoting a null from Δ_N to an extra constant in Δ_C. Also, given a set S of atoms, we denote by $\lceil S \rfloor$ the set obtained from S after freezing all of its nulls. The following definition formalizes the notion of *parsimonious-chase resumption* after freezing actions.

Definition 4.7. Let $P \in$ *Datalog*$^{\exists}$. The set $\mathsf{pChase}(P, 0)$ denotes $\mathsf{data}(P)$, while the set $\mathsf{pChase}(P, k)$ denotes $\mathsf{pChase}(\mathsf{rules}(P) \cup \lceil \mathsf{pChase}(k-1) \rfloor)$, for each $k > 0$. □

Clearly, the sequence $\{\mathsf{pChase}(P, k)\}_{k \in \mathbb{N}}$ is monotonically increasing; the limit of this sequence is denoted by $\mathsf{pChase}(P, \infty)$. The next lemma states that the proposed resumption technique is always sound w.r.t. QA, and that its infinite application also ensures completeness.

Lemma 4.8. *$\mathsf{pChase}(P, \infty) = \mathsf{chase}(P)$ $\forall P \in$ Datalog$^{\exists}$.*

Proof. The statement holds since operator $\Vdash$ in PARSIM-CHASE behaves, on freezed nulls, as $\models$ in the CHASE. □

Before proving that the PARSIM-CHASE algorithm over *Shy* programs is complete w.r.t. CQ answering after a finite number of resumptions, we need to introduce some more notation. The *chase-graph* for a *Datalog*$^{\exists}$ program P is the directed acyclic graph $G_P = \langle \mathsf{chase}(P), A \rangle$ where $(\mathbf{a}, \mathbf{b}) \in A$ iff $\mathbf{b}$ has been inferred by the CHASE through a firing substitution σ for a rule r where $\mathbf{a} \in \sigma(\mathsf{body}(r))$. Moreover, for a given set $S \in \mathsf{chase}(P)$, G_P^S denotes the maximal subgraph of G_P where a node may have no ingoing arc only if it belongs to S.

Lemma 4.9. *Let P be a Shy program, q be a CQ, $\sigma_a \in \mathsf{ans}_P(q)$, σ be a substitution proving that $P \models \sigma_a(q)$ holds, and $\mathbf{Y}_N$ be only and all the $\exists$-variables of q mapped by σ to nulls. Then, there is a substitution σ', proving that $P \models \sigma_a(q)$ holds, that maps at least one variable in $\mathbf{Y}_N$ to a term occurring in $\mathsf{pChase}(P)$.*

Proof. Let Φ contain the nulls occurring in $\sigma(q)$, B contain the atoms in $\mathsf{chase}(P)$ where the nulls of Φ have been introduced for the first time, and $\mathbf{b}_1, \ldots, \mathbf{b}_n$ be the atoms of B listed in the same order they have been inferred by the CHASE. Moreover, let Φ_i denote, for each $i \in [1..n]$, the subset of Φ of only and all the nulls that have been introduced for the first time in $\mathbf{b}_i$, and $\mathbf{a}_1$ an atom form $\mathsf{pChase}(P)$ such that $\mathbf{a}_1 \preceq \mathbf{b}_1$. We build σ' in such a way that at least one variable in $\mathbf{Y}_N$ is mapped to some term occurring in $\mathbf{a}_1$. In particular, we build a set $A \subseteq \mathsf{chase}(P)$ and a homomorphism $h : \Phi \to \mathsf{terms}(A)$ such that $\mathbf{a}_1 \in A$ and, for each atom $\mathbf{b}$ in G_P^B containing at least a null from Φ, there is $h' \supseteq h$ such that $h'(\mathbf{b})$ belongs to G_P^A. Finally, $\sigma' = h \circ \sigma$.

We proceed by induction. More precisely, we construct A, G_P^A and h by progressively considering all the atoms of G_P^B in the same order they have been inferred by the CHASE. Initially, $A = \{\mathbf{a}_1\}$, G_P^A contains only node $\mathbf{a}_1$, and h maps each constant in $\sigma(q)$ to itself, and each null in Φ_1 occurring at position i in $\mathbf{b}_1$ to the i^{th} term of $\mathbf{a}_1$.

Base case: Let $\mathbf{b}$ be the first atom in G_P^B inferred by the CHASE, via a rule r, after $\mathbf{b}_1$. Let σ_r be the firing substitution for r used by the CHASE whose extension $\hat{\sigma_r}$ has produced $\mathbf{b}$. If $\sigma_r(\mathsf{body}(r))$ does not involve $\mathbf{b}_1$, then $\mathbf{b} = \mathbf{b}_2$ and we can choose any $\mathbf{a} \preceq \mathbf{b}$ to extend A. On the contrary, if $\sigma_r(\mathsf{body}(r))$ involves $\mathbf{b}_1$, since P is shy, then there is also a firing substitution σ'_r for r, where $\mathbf{a}_1 \in \sigma'_r(\mathsf{body}(r))$ and $\sigma_r(\mathsf{body}(r)) - \{\mathbf{b}_1\} = \sigma'_r(\mathsf{body}(r)) - \{\mathbf{a}_1\}$. (Note that also in this case, $\mathbf{b} = \mathbf{b}_2$ might hold.) Clearly, if σ_r can be extended to infer a new atom $\mathbf{b}$, then either σ'_r can be extended to infer a new atom $\mathbf{a}$ or there is already some $\mathbf{a}$ such that $\{\mathbf{a}\} \models \sigma'_r(\mathsf{head}(r))$. But since a null in $\mathbf{a}$ but not in $\mathbf{b}$ either comes from $\mathbf{a}_1$ or it is fresh, then $\mathbf{a} \preceq \mathbf{b}$. Finally, $\mathbf{a}$ is added to A, G_P^A is updated and, only in case $\mathbf{b} = \mathbf{b}_2$, h is updated according to $\mathbf{a}$.

Inductive hypothesis: After considering the first k atoms in G_P^B inferred by the CHASE, we assume that, for each such an atom $\mathbf{b}$ containing at least a null from Φ, there is $h' \supseteq h$ such that $h'(\mathbf{b})$ belongs to G_P^A.

Inductive step: Let $\mathbf{b}$ be the $(k+1)^{th}$ atom in G_P^B inferred by the CHASE, via a rule r. By using the same argument that was used in the Base case, we can extend A and G_P^A by an atom $\mathbf{a} \preceq \mathbf{b}$. Moreover, if $\mathbf{b} = \mathbf{b}_i$ for some $i \in [2..n]$, then h is updated according to $\mathbf{a}$. The only difference here is that $\mathbf{b}$ may require more than one atom among the first k already inferred. □

Lemma 4.10. *Let $P \in$ Shy and q be a CQ with n different $\exists$-variables. Then, $\mathsf{ans}_P(q) \subseteq \mathsf{ans}(q, \mathsf{pChase}(P, n+1))$.*

Proof. In Light of Lemma 4.9, in the worst case, to be sure that all the nulls involved by σ' are generated, we claim that it is enough to compute $\mathsf{pChase}(P, n)$ where n is the number of $\exists$-variables of q. With respect to Lemma 4.9, let $\mathtt{Y}$ be one of the variable in $\mathbf{Y}_N$ mapped by σ' to a term occurring in $\mathsf{pChase}(P)$. Assume that this term is a null say φ. After freezing φ, we could replace $\mathtt{Y}$ in q by φ to obtain q'. Clearly, $P \models \sigma_a(q)$ iff $P \models \sigma_a(q')$. However, The BCQ $\sigma_a(q')$ has an $\exists$-variable less than the BCQ $\sigma_a(q)$. Thus, we can use again the statement of Lemma 4.9 after replacing $\mathsf{pChase}(P)$ by $\mathsf{pChase}(P, 2)$ and q by q'. We can reiterate this process until the query has no $\exists$-variable, namely after $n-1$ resumptions producing $\mathsf{pChase}(P, n)$. Finally, by Definition 3.2, we are sure that $\mathsf{pChase}(P, n+1)$ contains all the atoms appearing in $\sigma_a(\sigma'(q))$. □

Theorem 4.11. *Conjunctive QA over Shy is decidable.*

Proof. Soundness follows by Lemma 4.8, completeness by Lemma 4.10, while termination by combining Theorem 3.6 and Definition 4.7. □

The following example, after defining a *Shy* program P, shows that P imposes the computation of $\mathsf{pChase}(P, 3)$ to prove (after two resumptions) that a BCQ q containing two atoms and two variables is entailed by P.

Example 4.12. Let P denote the following *Shy* program.

```
p(a,b). u(c,d). r1 : ∃Z v(Z) :- u(X,Y). r2 : ∃Y
u(X,Y) :- v(X). r3  : p(X,Z) :- v(X), p(Y,Z).
r4 : p(X,W) :- p(X,Y), u(Z,W).
```

Consider the BCQ $q = \exists \mathtt{X}, \mathtt{Y}\ \mathtt{p(X,Y)}, \mathtt{u(X,Y)}$. Figure 1 shows that q cannot be proved before two freezing.

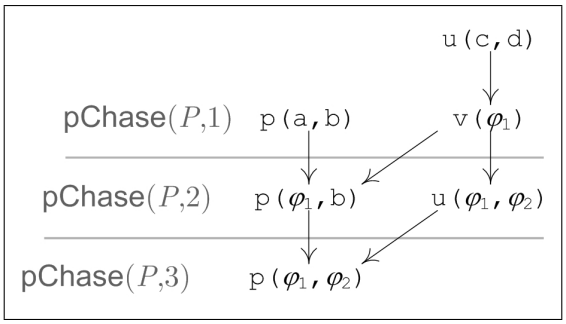

Figure 1: Snapshot of $\mathsf{pChase}(P, 3)$ w.r.t. Example 4.12

5 Computational Complexity

In this section we study the complexity of *Parsimonious-Sets* and *Shy* programs. Moreover, let $\mathcal{C}$ be one of these classes, we talk about *combined complexity* of $\mathrm{QA}_{[\mathcal{C}]}$ in general, and about *data complexity* of $\mathrm{QA}_{[\mathcal{C}]}$ under the assumption that $\mathsf{data}(P)$ are the only input while both q and $\mathsf{rules}(P)$ are considered fixed. The results obtained from our analysis have been then compared, in Section 8, with those already proved for some representative Datalog$^{\pm}$ languages. We start with upper bounds.

Theorem 5.1. $\mathrm{QA}_{[\textit{Parsimonious-Sets}]}$ *is in* ***P*** *(resp.,* ***EXP****) in data complexity (resp., combined complexity).*

Proof. Let P be a parsimonious program, α be the maximum arity over all predicate symbols in P, and β be the maximum number of body atoms over all rules in P. Since $|\mathsf{pChase}(P)| \leq |\mathsf{preds}(P)| \cdot (|\mathsf{dom}(P)| + \alpha)^{\alpha}$ by Corollary 3.5, then each rule admits at most $|\mathsf{pChase}(P)|^{\beta}$ different firing substitutions. Thus, all the firing substitutions are no more that $|P - \mathsf{data}(P)| \cdot |\mathsf{preds}(P)|^{\beta} \cdot (|\mathsf{dom}(P)| + \alpha)^{\alpha \cdot \beta}$. Moreover, for each firing substitution σ for a rule r, the algorithm has to check whether there is an homomorphism from $\hat{\sigma}(\mathsf{head}(r))$ to $\mathsf{pChase}(P)$. These checks are no more than $|P - \mathsf{data}(P)| \cdot |\mathsf{preds}(P)|^{2 \cdot \beta} \cdot (|\mathsf{dom}(P)| + \alpha)^{2 \cdot \alpha \cdot \beta}$. □

We now consider lower bounds, and thus completeness.

Theorem 5.2. *Both* $\mathrm{QA}_{[\textit{Shy}]}$ *and* $\mathrm{QA}_{[\textit{Parsimonious-Sets}]}$ *are* ***P****-complete (resp.,* ***EXP****-complete) in data complexity (resp., combined complexity).*

Proof. Since, by Theorem 4.4, a shy program is also parsimonious, then (i) upper-bounds of Theorem 5.1 hold for *Shy* programs as well; (ii) lower-bounds for $\mathrm{QA}_{[\textit{Datalog}]}$ (Dantsin et al. 2001) also hold both for *Shy* and *Parsimonious-Sets* programs, by Theorem 8.1. □

6 Implementation and Optimizations

We implemented a system for answering conjunctive queries over *Shy* programs (it actually works on any parsimonious program). The system, called DLV$^{\exists}$, efficiently integrates

the PARSIM-CHASE algorithm defined in Section 3 and the resumption technique introduced in Section 4.2, in the well known Answer Set Programming (ASP) system DLV (Leone et al. 2006). Following the DLV philosophy, it has been designed as an in-memory reasoning system.

To answer a CQ q against a *Shy* program P, DLV$^\exists$ carries out the following steps.

Skolemization. $\exists$-variables in rule heads are managed by skolemization. Given a head atom $\mathbf{a} = \mathtt{p}(t_1, \ldots, t_k)$, let us denote by $\mathsf{fpos}(\mathtt{Y}, \mathbf{a})$ the position of the first occurrence of variable $\mathtt{Y}$ in $\mathbf{a}$. The skolemized version of $\mathbf{a}$ is obtained by replacing in $\mathbf{a}$ each $\exists$-variable $\mathtt{Y}$ by $f^{\mathtt{p}}_{\mathsf{fpos}(\mathtt{Y},\mathbf{a})}(t'_1, \ldots, t'_k)$ where, for each $i \in [1..k]$, t'_i is either $\#_{\mathsf{fpos}(t_i,\mathbf{a})}$ or t_i according to whether t_i is an $\exists$-variable or not, respectively. Every rule in P with $\exists$-variables is skolemized in this way, and skolemized terms are interpreted as functional symbols (Calimeri et al. 2010) within DLV$^\exists$.

Example 6.1. The *Datalog*$^\exists$ rule

```
∃X,Y p(Z,X,W,Y) :- s(Z,W).
```

is skolemized in

```
p(Z,t1,W,t2) :- s(Z,W).
```

where $t_1 = f^{\mathtt{p}}_2(\mathtt{Z}, \#_2, \mathtt{W}, \#_4)$, $t_2 = f^{\mathtt{p}}_4(\mathtt{Z}, \#_2, \mathtt{W}, \#_4)$. □

Data Loading and Filtering. Since DLV$^\exists$ is an in-memory system, it needs to load input data in memory before the reasoning process can start. In order to optimize the execution, the system first singles out the set of predicates which are needed to answer the input query, by recursively traversing top-down (head-to-body) the rules in P, starting from the query predicates. This information is used to filter out, at loading time, all the facts belonging to predicates certainly irrelevant for answering the input query.

Program Optimization. Data filtering, carried out at the level of predicates, may still include some facts which are not needed for the query at hand. The DLV$^\exists$ computation is further optimized by "pushing-down" the bindings coming from possible query constants. To this end, the program is rewritten by a variant of the well-known magic-set optimization technique (Cumbo et al. 2004), that we adapted to *Datalog*$^\exists$ by avoiding to propagate bindings through "attacked" argument-positions (since $\exists$-quantifiers generate "unknown" constants). The result is a program, being equivalent to P for the given query, that can be evaluated more efficiently. In the following, P denotes the program that has been rewritten by magic-sets.

pChase Computation and Optimized Resumption. After the skolemization, loading, and rewriting phases, the system computes $\mathsf{pChase}(P)$ as defined in Section 3. Since $\exists$-variables have been skolemized, the rules are safe and can be evaluated in the usual bottom-up way; but, according to $\mathsf{pChase}(P)$, the generation of homomorphic atoms should be avoided. To this end, each time a new head-atom $\mathbf{a}$ is derivable, the system verifies whether an homomorphic atom had been previously derived, where each skolem term is considered as a null for the sake of homomorphisms verification. In the negative case, $\mathbf{a}$ is derived; otherwise it is discarded.

Algorithm 3 RESUMPTION-LEVEL(q, P)

Input: A CQ $q = \exists\mathbf{Y}\ \mathbf{conj}_{[\mathbf{X}\cup\mathbf{Y}]}$ and a program P
Output: The number of needed resumptions for q and P.
1. $\mathbf{Y}_* := \mathbf{Y}$
2. **for each** $\mathtt{Y} \in \mathbf{Y}$ **do**
3. **if** $\mathtt{Y}$ is protected in q **OR** $\mathtt{Y}$ occurs in only one atom of q
4. $\mathsf{remove}(\mathtt{Y}, \mathbf{Y}_*)$
5. **return** $|\mathbf{Y}_*|$

If the input query is atomic, then $\mathsf{pChase}(P)$ is sufficient to provide an answer (see Proposition 3.3); otherwise, the fixpoint computation should be resumed several times (see Lemma 4.10). In this case, every null (skolem term) derived in previous reiterations is *freezed* (see Section 4.2) and considered as a standard constant; in our implementation, this is implemented by attaching a "level" to each skolem term, representing the fixpoint reiteration where it has been derived. This is important because homomorphism verification must consider as nulls only skolem terms produced in the current resumption-phase; while previously introduced skolem terms must be interpreted as constants. The number k of times that the fixpoint must be reiterated has been stated in Lemma 4.10. In our implementation, this number is further reduced by Algorithm 3 considering the structure of the query w.r.t. P.

Query Answering. After the fixpoint is resumed k times, the answers to query q are given by $\mathsf{ans}(q, \mathsf{pChase}(P, k+1))$.

7 Experiments

In this section we report on some experiments we carried out to evaluate the efficiency and the effectiveness of DLV$^\exists$.

Benchmark Focus. The focus of our tests is on rapidly changing and evolving ontologies (rules or data). In fact, in many contexts data frequently vary, even within hours, and there is the need to always provide the most updated answers to user queries. One of these contexts is e-commerce; another example is the university context, where data on exams, courses schedule and assignments may vary on a frequent basis. Benchmark framework from university domain and obtained results are discussed next.

Compared Systems. As it will be pointed out in Section 8, ontology reasoners mainly rely on three categories of inference, namely: tableau, forward-chaining, and query-rewriting. Systems belonging to the latter category are still research prototypes and a comparison with them was not possible. We compared DLV$^\exists$ with the following systems, being representatives of the first two categories.

► Pellet (Sirin et al. 2007) is an OWL 2 reasoner which implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, conjunctive query answering).

► OWLIM-SE (Bishop et al. 2011) is a commercial product which supports the full set of valid inferences using RDFS semantics; it's reasoning is based on forward-chaining. This system is oriented to massive volumes of data and, as such, based on persistent storage manipulation and reasoning.
► OWLIM-Lite (Bishop et al. 2011), sharing the same inference mechanisms and semantics with OWLIM-SE, is another product of the OWLIM family designed for medium data volumes; reasoning and query evaluation are performed in main memory.

Data Sets. We concentrated on a well known benchmark suite for testing reasoners over ontologies, namely LUBM (Guo, Pan, and Heflin 2005).

The Lehigh University Benchmark (LUBM) has been specifically developed to facilitate the evaluation of Semantic Web reasoners in a standard and systematic way. In fact, the benchmark is intended to evaluate the performance of those reasoners with respect to extensional queries over large data sets that commit to a single realistic ontology. It consists of a university domain ontology with customizable and repeatable synthetic data. The LUBM ontology schema and its data generation tool are quite complex and their description is out of the scope of this paper.

We used the Univ-Bench ontology that describes (among others) universities, departments, students, professors and relationships among them; we considered the entire set of rules in Univ-Bench, except for equivalences with restrictions on roles, which cannot be expressed in *Shy* in some cases; these have been transformed in subsumptions. Data generation is carried out by the Univ-Bench data generator tool (UBA) whose main generation parameter is the number of universities to consider. The interested reader can find all information in (Guo, Pan, and Heflin 2005).

In order to perform scalability tests, we generated a number of increasing data sets named: `lubm-10`, `lubm-30`, and `lubm-50`, where right-hand sides of these acronyms indicate the number of universities used as parameter to generate the data. The number of statements (both individuals and assertions) stored in the data sets vary from about 1M for `lubm-10` to about 7M for `lubm-50`

LUBM incorporates a set of 14 queries aimed at testing different capabilities of the systems. A detailed description of rules and queries is provided at `http://www.mat.unical.it/kr2012`.

Data preparation. LUBM is provided as owl files. Each owl class is associated with a unary predicate in *Datalog*$^\exists$; each individual of a class is represented by a *Datalog*$^\exists$ fact on the corresponding predicate. Each role is translated in a binary *Datalog*$^\exists$ predicate with the same name. Finally, assertions are translated in suitable *Shy* rules. The following example shows some translations where the DL has been used for clarity.

Example 7.1. The assertions

`AdministrativeStaff` $\sqsubseteq$ `Employee`
`subOrgOf`$^+$

are translated in the following rules:

```
Employee(X) :- AdministrativeStaff(X).
subOrgOf(X,Z) :-subOrgOf(X,Y),subOrgOf(Y,Z).
```

where `subOrgOf` stands for `subOrganizationOf`. □

The complete list of correspondences between DL, OWL, and *Datalog*$^\exists$ rules and queries is provided at `http://www.mat.unical.it/kr2012`.

Results and Discussion. Tests have been carried out on an Intel Xeon X3430, 2.4 GHz, with 4 Gb Ram, running Linux Operating System; for each query, we allowed a maximum running time of 7200 seconds (two hours).

Table 1 reports the times taken by the tested systems to answer the 14 LUBM queries. Since, as previously pointed out, we are interested in evaluating a rapidly changing scenario, each entry of the table reports the *total* time taken to answer the respective query by a system (including also loading and reasoning). In addition, the first column (labeled Q_{all}) shows the time taken by the systems to compute all atomic consequences of the program; this roughly corresponds to loading and inference time for Pellet, OWLIM-Lite, and OWLIM-SE and to parsing and first fixpoint computation for DLV$^\exists$.

The results in Table 1 show that DLV$^\exists$ clearly outperforms the other systems as an on-the-fly reasoner. In fact, the overall running times for DLV$^\exists$ are significantly lower than the corresponding times for the other systems. Pellet shows, overall, the worst performances. In fact, it has not been able to complete any query against `lubm-30` and `lubm-50`, and is also slower than competitors for the smallest data sets.

For both OWLIM-Lite and OWLIM-SE, most of the total time is taken for loading/inference (Q_{all}), as the reconstruction of the answers from the materialized inferences is a trivial task, often taking less than one second. However, as previously stated, this behavior is unsuited for reasoning on frequently changing ontologies, where previous inferences and materialization cannot be re-used, and loading must be repeated or time-consuming updates must be performed. As expected, loading/inference times (Q_{all}) for OWLIM-SE are higher than for OWLIM-Lite, but OWLIM-SE is faster than OWLIM-Lite in the reconstruction of the answers from the materialized inferences (this time is basically obtainable by subtracting Q_{all}). Because of this inefficiency in answers-reconstruction OWLIM-Lite has not been able to answer some queries in the time-limit that we set for the experiments (two hours); these queries involve many classes and roles.

We carried out some tests also on ontology updates; just to show an example, deleting 10% of `lubm-50` individuals imposed OWLIM-SE 152 seconds of update activities, which is sensibly higher than the highest query time needed by DLV$^\exists$ (42 seconds for Q_9) on the same data set. OWLIM-Lite was even worse on updates, since it required 133 seconds for the deletion of just one individual.

It is worth pointing out that DLV$^\exists$ is the only of the tested systems for which the times needed for answering single queries ($Q_1 \ldots Q_{14}$) are significantly smaller than those required for materializing all atomic consequences (Q_{all}). This result highlights the effectiveness of the query-oriented optimizations implemented in DLV$^\exists$ (magic sets and filtering, in particular), and confirms the suitability of the system for

	Q_{all}	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	# solved	Geom. Avg time
lubm-10																	
DLV$^\exists$	17	5	4	2	4	6	1	6	4	8	5	<1	1	6	2	14	2.87
Pellet	27	82	84	84	82	80	88	81	89	95	82	82	89	82	84	14	84.48
OWLIM-Lite	33	33	–	33	33	33	33	4909	70	–	33	33	33	33	33	12	53.31
OWLIM-SE	105	105	105	105	105	105	105	105	106	106	105	105	105	105	105	14	105.14
lubm-30																	
DLV$^\exists$	55	16	13	7	14	21	3	21	12	25	18	<1	5	23	8	14	9.70
Pellet	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	–
OWLIM-Lite	106	107	–	107	106	107	106	–	528	–	107	106	106	107	106	11	123.18
OWLIM-SE	323	323	328	323	323	323	323	323	323	326	323	323	323	323	323	14	323.57
lubm-50																	
DLV$^\exists$	93	27	23	12	23	35	6	34	22	42	31	<1	9	33	14	14	16.67
Pellet	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	–
OWLIM-Lite	187	188	–	190	187	189	188	–	1272	–	189	187	187	189	187	11	223.79
OWLIM-SE	536	536	547	536	536	536	537	536	536	542	536	536	536	536	537	14	537.35

Table 1: Running times for LUBM queries (sec).

on-the-fly query answering. Interestingly, even if DLV$^\exists$ is specifically designed for query answering, it outperformed the competitors also for the computation of *all* atomic consequences (query Q_{all}). Indeed, on each of the three ontologies, DLV$^\exists$ took, respectively, about 17% and 51% of the time taken by OWLIM-SE and OWLIM-Lite.

8 Related Work

8.1 Datalog$^\pm$ Languages

We overview the most relevant QA-decidable subclasses of *Datalog*$^\exists$ defined in the literature. Then, we provide their precise taxonomy and the complexity of QA in each class, highlighting the differences to *Parsimonious-Sets* and *Shy*.

The best-known QA-decidable subclass of *Datalog*$^\exists$ is clearly *Datalog*, the largest $\exists$-free *Datalog*$^\exists$ class (Abiteboul, Hull, and Vianu 1995) which, notably, admits a unique and yet finite (universal) model enabling efficient QA.

Three abstract QA-decidable classes have been singled out, namely, *Finite-Expansion-Sets*, *Finite-Treewidth-Sets*, and *Finite-Unification-Sets* (Baget et al. 2009; Baget, Leclère, and Mugnier 2010). Intuitively, the semantic properties behind these classes rely on a "forward-chaining inference that halts in finite time", a "forward-chaining inference that generates a tree-shaped structure", and a "backward-chaining inference that halts in finite time", respectively.

Syntactic subclasses of *Finite-Treewidth-Sets*, of increasing complexity and expressivity, have been defined by Calì, Gottlob, and Kifer (2008). They are: (i) *Linear-Datalog*$^\exists$ where at most one body atom is allowed in each rule; (ii) *Guarded-Datalog*$^\exists$ where each rule needs at least one body atom that covers all $\forall$-variables; and (iii) *Weakly-Guarded-Datalog*$^\exists$ extending *Guarded* by allowing unaffected "unguarded" variables (see Section 4.1 for the meaning of unaffected). The first one generalizes the well known *Inclusion-Dependencies* class (Johnson and Klug 1984; Abiteboul, Hull, and Vianu 1995), with no computational overhead; while only the last one is a superset of *Datalog*, but at the price of a drastic increase in complexity. In general, to be complete w.r.t. QA, the CHASE ran on a program belonging to one of the latter two classes requires the generation of a very high number of isomorphic atoms, so that no (efficient) implementation has been realized yet.

More recently, another class of *Datalog*$^\exists$, called *Sticky*, has been defined by Calì, Gottlob, and Pieris (2010a). Such a class enjoys very good complexity, encompasses *Inclusion-Dependencies*, but since it is FO-rewritable, it has limited expressive power and, clearly, does not include *Datalog*. Intuitively, if a program is sticky, then all the atoms that are inferred (by the CHASE) starting from a given join contain the term of this join. Several generalizations of stickiness have been defined by Calì, Gottlob, and Pieris (2010b). For example, the *Sticky-Join* class preserves the sticky-complexity by also including *Linear-Datalog*$^\exists$. Both *Sticky* and *Sticky-Join* are subclasses of *Finite-Unification-Sets*.

Finally, in the context of data exchange, where a finite universal model is required, *Weakly-Acyclic-Datalog*$^\exists$, a subclass of *Finite-Expansion-Sets*, has been introduced (Fagin et al. 2005). Intuitively, a program is weakly-acyclic if the presence of a null occurring in an inferred atom at a given position does not trigger the inference of an infinite number of atoms (with the same predicate symbol) containing several nulls in the same position. This class both includes and has much higher complexity than *Datalog*, but misses to capture even *Inclusion-Dependencies*. A number of extensions, techniques and criteria for checking chase termination have been recently proposed in this context (Deutsch, Nash, and Remmel 2008; Marnette 2009; Meier, Schmidt, and Lausen 2009; Greco, Spezzano, and Trubitsyna 2011).

Figure 2 provides a precise taxonomy of the considered classes; while Table 2 summarizes the complexity of $\text{QA}_{[\mathcal{C}]}$, by varying $\mathcal{C}$ among the syntactic classes. In both diagrams, only *Datalog* is intended to be $\exists$-free; while *Datalog*$^\exists$ is the only undecidable language in the figure.

Theorem 8.1. *For each pair $\mathcal{C}_1$ and $\mathcal{C}_2$ of classes represented in Figure 2, the following hold: (i) there is a direct path from $\mathcal{C}_1$ to $\mathcal{C}_2$ iff $\mathcal{C}_1 \supset \mathcal{C}_2$; (ii) $\mathcal{C}_1$ and $\mathcal{C}_2$ are not linked by any directed path iff they are uncomparable.*

Proof. Relationships among known classes are pointed out by Mugnier (2011). *Shy* $\subset$ *Parsimonious-Sets* holds by Theorem 4.4. *Shy* $\supset$ *Datalog* $\cup$ *Linear* holds since *Datalog* programs only admit protected positions, while *Linear* ones only bodies with one atom. However, since there are both *Weakly-Acyclic* and *Sticky* programs being not *Parsimonious-Sets*, then both *Shy* and

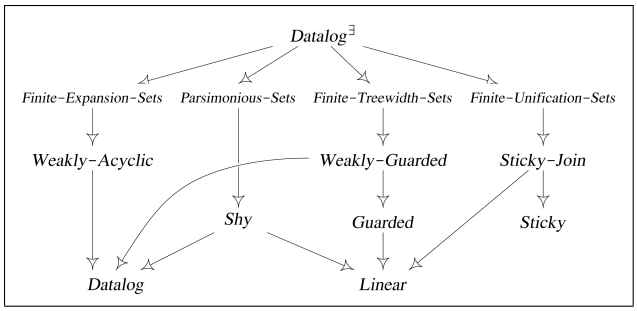

Figure 2: Taxonomy of representative Datalog$^{\pm}$ languages

Class $\mathcal{C}$	*Data Complexity*	*Combined Complexity*
Weakly-Guarded	**EXP**-complete	**2EXP**-complete
Guarded *Weakly-Acyclic*	**P**-complete	**2EXP**-complete
Datalog, *Shy* (*Parsimonious-Sets*)	**P**-complete	**EXP**-complete
Sticky, *Sticky-Join*	in $\mathbf{AC}_0$	**EXP**-complete
Linear	in $\mathbf{AC}_0$	**PSPACE**-complete

Table 2: Complexity of the $\mathtt{QA}_{[\mathcal{C}]}$ problem

Parsimonious-Sets are uncomparable to *Finite-Expansion-Sets*, *Weakly-Acyclic*, *Finite-Unification-Sets*, *Sticky-Join* and *Sticky*. Now, to prove that *Shy* $\not\subseteq$ *Finite-Treewidth-Sets* we use the shy program

```
set1(a,a).    ∃V′ set1(V,V′) :- set1(X,V).
set2(b,b).    ∃V′ set2(V,V′) :- set2(X,V).
graphK(V1,V2) :- set1(V1,X), set2(V2,Y).
```

whose chase-graph G_P has no finite treewidth (Calì, Gottlob, and Kifer 2008) since it contains a complete bipartite graph $K_{n,n}$ of $2n$ vertices – the treewidth of which is n (Kloks 1994) – where n is not finite. Finally, since there are *Guarded* programs that are not *Parsimonious-Sets*, then both *Shy* and *Parsimonious-Sets* are uncomparable to *Finite-Treewidth-Sets*, *Weakly-Guarded* and *Guarded*. □

We care to notice that the proof of Theorem 8.1 uses the so called *concept product* to generate a complete and infinite bipartite graph. A natural and common example is

```
biggerThan(X,Y) :- elephant(X), mouse(Y).
```

that is expressible in *Shy* if `elephant` and `mouse` are disjoint concepts. However, such a concept cannot be expressed in *Finite-Treewidth-Sets* and can be only simulated by a very expressive ontology language for which no tight worst-case complexity is known (Rudolph, Krötzsch, and Hitzler 2008).

8.2 Ontology Reasoners

To the best of our knowledge, there is only one ongoing research work directly supporting ∃-quantifiers in *Datalog*, namely Nyaya (De Virgilio et al. 2011). This system, based on an SQL-rewriting, allows a strict subclass of *Shy* called *Linear-Datalog*$^{\exists}$, which does not include, for instance, transitivity and concept products.[5]

Since the system we developed enables ontology reasoning, existing ontology reasoners are also related. They can be classified in three groups: *query-rewriting*, *tableau* and *forward-chaining*.

The systems QuOnto (Acciarri et al. 2005), Presto (Rosati and Almatelli 2010), Quest (Rodriguez-Muro and Calvanese 2011a), Mastro (Calvanese et al. 2011) and OBDA (Rodriguez-Muro and Calvanese 2011b) belong to the query-rewriting category. They rewrite axioms and queries to SQL, and use RDBMSs for answers computation. Such systems support standard first-order semantics for unrestricted CQs; but the expressivity of their languages is limited to $\mathbf{AC}_0$ and excludes, for instance, transitivity property or concept products.

The systems FaCT++ (Tsarkov and Horrocks 2006), RacerPro (Haarslev and Möller 2001), Pellet (Sirin et al. 2007) and HermiT (Motik, Shearer, and Horrocks 2009) are based on tableau calculi. They materialize all inferences at loading-time, implement very expressive description logics, but they do not support the standard first-order semantics for CQs (Glimm et al. 2008). Actually, the Pellet system enables first-order CQs but only in the acyclic case.

OWLIM (Bishop et al. 2011) and KAON2 (Hustadt, Motik, and Sattler 2004) are based on forward-chaining.[6] Similar to tableau-based systems, they perform full-materialization and implement expressive DLs, but they still miss to support the standard first-order semantics for CQs (Glimm et al. 2008).

Summing up, it turns out that DLV$^{\exists}$ is the first system supporting the standard first-order semantics for unrestricted CQs with ∃-variables over ontologies with advanced properties (some of these beyond $\mathbf{AC}_0$), such as, role transitivity, role hierarchy, role inverse, and concept products. The experiments confirm the efficiency of DLV$^{\exists}$, which constitutes a powerful system for a fully-declarative ontology-based query answering.

9 Acknowledgments

The authors want to thank: (i) Georg Gottlob, Giorgio Orsi, and Andreas Pieris for useful discussions on the problem; (ii) Mario Alviano for his support in the adaptation of the magic-set technique; and (iii) Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner for some tips about Description Logic classes and systems.

References

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases: The Logical Level*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Acciarri, A.; Calvanese, D.; De Giacomo, G.; Lembo, D.; Lenzerini, M.; Palmieri, M.; and Rosati, R. 2005.

[5] We could not compare DLV$^{\exists}$ with Nyaya since, as a research prototype, Nyaya provides no API for data loading and querying.

[6] Actually, KAON2 first translates the ontology to a disjunctive *Datalog* program, on which forward inference is then performed.

QUONTO: querying ontologies. In *Proc. of the 20th national conference on Artificial intelligence*, volume 4, 1670–1671. AAAI Press.

Baget, J.-F.; Leclère, M.; Mugnier, M.-L.; and Salvat, E. 2009. Extending Decidable Cases for Rules with Existential Variables. In Boutilier, C., ed., *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI '09, 677–682.

Baget, J.-F.; Leclère, M.; and Mugnier, M.-L. 2010. Walking the Decidability Line for Rules with Existential Variables. In Lin, F.; Sattler, U.; and Truszczynski, M., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference*, KR '10. AAAI Press.

Bishop, B.; Kiryakov, A.; Ognyanoff, D.; Peikov, I.; Tashev, Z.; and Velkov, R. 2011. OWLIM: A family of scalable semantic repositories. *Semant. web* 2:33–42.

Calì, A.; Gottlob, G.; and Kifer, M. 2008. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. In *Proc. of the 11th International Conference on Principles of Knowledge Representation and Reasoning*, 70–80. AAAI Press. Revised version: `http://dbai.tuwien.ac.at/staff/gottlob/CGK.pdf`.

Calì, A.; Gottlob, G.; and Lukasiewicz, T. 2009. A general datalog-based framework for tractable query answering over ontologies. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '09, 77–86. New York, NY, USA: ACM.

Calì, A.; Gottlob, G.; and Pieris, A. 2010a. Advanced Processing for Ontological Queries. *PVLDB* 3(1):554–565.

Calì, A.; Gottlob, G.; and Pieris, A. 2010b. Query Answering under Non-guarded Rules in Datalog$^{\pm}$. In Hitzler, P., and Lukasiewicz, T., eds., *Proceedings of the 4th International Conference on Web Reasoning and Rule Systems*, volume 6333 of *Lecture Notes in Computer Science*, 1–17. Springer.

Calì, A.; Gottlob, G.; and Pieris, A. 2011. New Expressive Languages for Ontological Query Answering. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence*, 1541–1546.

Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. 2010. Enhancing ASP by Functions: Decidable Classes and Implementation Techniques. In Fox, M., and Poole, D., eds., *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI '10. AAAI Press.

Calvanese, D.; 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. Reason.* 39:385–429.

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.

Cumbo, C.; Faber, W.; Greco, G.; and Leone, N. 2004. Enhancing the magic-set method for disjunctive datalog programs. In *Proceedings of the the 20th International Conference on Logic Programming - ICLP '04*, volume 3132 of *LNCS*, 371–385.

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

De Virgilio, R.; Orsi, G.; Tanca, L.; and Torlone, R. 2011. Semantic Data Markets: A Flexible Environment for Knowledge Management. In *Proc. of the 20th ACM international Conference on Information and Knowledge Management*, CIKM '11. New York, NY, USA: ACM. to appear.

Deutsch, A.; Nash, A.; and Remmel, J. 2008. The chase revisiteddeutschnashremmelpods2008. In *Proc. of the 27th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.*, PODS '08, 149–158. New York, NY, USA: ACM.

Fagin, R.; Kolaitis, P. G.; Miller, R. J.; and Popa, L. 2005. Data exchange: semantics and query answering. *Theoretical Computer Science* 336(1):89–124.

Glimm, B.; Horrocks, I.; Lutz, C.; and Sattler, U. 2008. Conjunctive query answering for the description logic SHIQ. *J. Artif. Int. Res.* 31(1):157–204.

Greco, S.; Spezzano, F.; and Trubitsyna, I. 2011. Stratification criteria and rewriting techniques for checking chase termination. *PVLDB* 4(11):1158–1168.

Guo, Y.; Pan, Z.; and Heflin, J. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semant.* 3:158–182. See `URL:http://swat.cse.lehigh.edu/projects/lubm/`.

Haarslev, V., and Möller, R. 2001. Racer system description. In Goré, R.; Leitsch, A.; and Nipkow, T., eds., *International Joint Conference on Automated Reasoning, IJCAR'2001*, 701–705. Siena, Italy: Springer-Verlag.

Hustadt, U.; Motik, B.; and Sattler, U. 2004. Reducing SHIQ- Descrption Logic to Disjunctive Datalog Programs. In *Proc. of the 9th International Conference on Knowledge Representation and Reasoning*, KR '04, 152–162.

Johnson, D., and Klug, A. 1984. Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences* 28(1):167–189.

Kloks, T. 1994. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer.

Kollia, I.; Glimm, B.; and Horrocks, I. 2011. Sparql query answering over owl ontologies. In *Proceedings of the 24th International Workshop on Description Logics*, volume 6643 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 382–396.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL* 7(3):499–562.

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

Marnette, B. 2009. Generalized schema-mappings: from termination to tractability. In *Proceedings of the twenty-eighth*

ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS '09, 13–22. New York, NY, USA: ACM.

Meier, M.; Schmidt, M.; and Lausen, G. 2009. On Chase Termination Beyond Stratification. *PVLDB* 2(1):970–981.

Motik, B.; Shearer, R.; and Horrocks, I. 2009. Hypertableau Reasoning for Description Logics. *Journal of Artificial Intelligence Research* 36:165–228.

Mugnier, M.-L. 2011. Ontological query answering with existential rules. In *Proceedings of the 5th international conference on Web reasoning and rule systems*, RR'11, 2–23. Berlin, Heidelberg: Springer-Verlag.

Rodriguez-Muro, M., and Calvanese, D. 2011a. Dependencies: Making ontology based data access work in practice. In *Proc. of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management*, volume 477.

Rodriguez-Muro, M., and Calvanese, D. 2011b. Dependencies to optimize ontology based data access. In Rosati, R.; Rudolph, S.; and Zakharyaschev, M., eds., *Description Logics*, volume 745 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Rosati, R., and Almatelli, A. 2010. Improving Query Answering over DL-Lite Ontologies. In *Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR 2010)*, KR '10, 290–300. Toronto, Ontario, Canada: AAAI Press.

Rudolph, S.; Krötzsch, M.; and Hitzler, P. 2008. All elephants are bigger than all mice. In *Proceedings of the 21st International Workshop on Description Logics*, volume 353 of *DL '08*. CEUR-WS.org.

Sirin, E.; Parsia, B.; Grau, B. C.; Kalyanpur, A.; and Katz, Y. 2007. Pellet: A practical OWL-DL reasoner. *Web Semant.* 5(2):51–53.

Tsarkov, D., and Horrocks, I. 2006. FaCT++ Description Logic Reasoner: System Description. In *Proc. of the 3rd Int. Joint Conf. on Automated Reasoning*, volume 4130 of *IJCAR '06*, 292–297.