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

**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 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], have started to build ontological reasoning modules on top of their existing software. 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* $\Sigma$. More specifically, $\Sigma$ 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 $\Sigma$. 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 QA over ontologies, is arousing increasing interest (Mugnier 2011). This family, generalizing 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.

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), *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 properties that are desired for ontology specification languages, none of them fully satisfy conditions $(1)$–$(5)$ above (see Section 8).

[*]Marco Manna's work was supported by the European Commission through the European Social Fund and by Calabria Region.


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

**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 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.[2] 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 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 CQs.

▶ We demonstrate that both *Parsimonious-Sets* and *Shy* preserve the same (data and combined) complexity of *Datalog* for atomic QA: 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 system for QA over *Shy* programs, which is profitably applicable for ontology-based QA. 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 systems for ontology-based QA. The results evidence that DLV$^\exists$ is the most effective system for QA 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*.

[2]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.

# 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 $\Delta_C$, $\Delta_N$ and $\Delta_V$, countably infinite domains of *terms* called *constants*, *nulls* and *variables*, respectively; by $\Delta$, the union of these three domains; by $t$, a generic *term*; by $\mathtt{c}$, $\mathtt{d}$ and $\mathtt{e}$, constants; by $\varphi$, a null; by $\mathtt{X}$ and $\mathtt{Y}$, variables; by $\mathbf{X}$ and $\mathbf{Y}$, sets of variables; by $\Pi$ 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 $\Pi$ and terms from $T$; if $\mathbf{a}$ is an atom, then $\mathsf{pred}(\mathbf{a})$ denotes the predicate symbol of $\mathbf{a}$; if $\varsigma$ is any formal structure containing atoms, then $\mathsf{terms}(\varsigma)$ (resp., $\mathsf{dom}(\varsigma)$) denotes all the terms from $\Delta$ (resp., $\Delta_C \cup \Delta_N$) occurring in the atoms of $\varsigma$.

**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 $\mu'$ is a restriction of $\mu$, then $\mu$ is called an *extension* of $\mu'$, 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 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 $\sigma$ from $\Delta$ 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:

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

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 for every person $X$ having both a father (some other person $Y$) and *john* as child:

```
∃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 $\sigma$ 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 $\sigma$ 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 $\sigma$ 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, $\mathsf{ans}_P(q) = \{\sigma_\emptyset\}$ only if $q$ 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 QA.

**Procedure 1** $\textsc{chase}(P)$

**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 $\sigma$ 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** $C$

**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 $\sigma$ for $q$, does $\sigma$ 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 $\sigma$ for $r$ w.r.t. $C$ is a substitution $\sigma$ on $\mathbf{X}$ s.t. $\sigma(\mathsf{body}(r)) \subseteq C$. Next, given a firing substitution $\sigma$ for $r$ w.r.t. $C$, the *fire* of $r$ on $C$ due to $\sigma$ infers $\hat{\sigma}(\mathsf{head}(r))$, where $\hat{\sigma}$ is an extension of $\sigma$ 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) $\mathsf{chase}(P)$.

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)* $\text{QA}_{[Datalog^\exists]}$ *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 as well as some of its properties. Due to space restrictions, some proofs have been sketched. Complete proofs can be found in the full version of this paper (Leone et al. 2011).

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

**Example 3.2.** Let $P$ be the "father-person" *Datalog*$^\exists$ program defined in the introduction, and augmented by the fact `person('john')`. Figure 1 compares $\mathsf{chase}(P)$ with $\mathsf{pChase}(P)$. Since, by definition, it holds that $\{$`person('john')`, `father('john',`$\varphi_1$`)`$\}$ $\Vdash$ `person(`$\varphi_1$`)`, then PARSIM-CHASE$(P)$ discards `person(`$\varphi_1$`)` and ends. □

| $\mathsf{chase}(P)$ | $\mathsf{pChase}(P)$ |
|---|---|
| `person('john')` | `person('john')` |
| $\downarrow$ | $\downarrow$ |
| `father('john',`$\varphi_1$`)` | `father('john',`$\varphi_1$`)` |
| $\downarrow$ | |
| `person(`$\varphi_1$`)` | |
| $\downarrow$ | |
| `father(`$\varphi_1$`,`$\varphi_2$`)` | |
| $\downarrow$ | |

Figure 1: $\mathsf{chase}(P)$ vs $\mathsf{pChase}(P)$ w.r.t. Example 3.2

Note that, differently from $\mathsf{chase}(P)$, $\mathsf{pChase}(P)$ 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 3.3.** A *Datalog*$^\exists$ program $P$ is *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 3.4.** *Algorithm* PARSIM-CHASE *over parsimonious programs is sound and complete w.r.t. atomic QA.*

*Proof (Sketch).* Let $P \in$ *Parsimonious-Sets*. Since, for any set of atoms $S$, $S \not\Vdash \mathbf{a}$ entails $S \not\models \mathbf{a}$, then, by Definition 3.1, $\mathsf{pChase}(P) \subseteq \mathsf{chase}(P)$ holds, ensuring soundness. For completeness, let $q$ be an atomic query and $\sigma \in \mathsf{ans}_P(q)$, if there is a substitution $\sigma'$ that maps $\sigma(q)$ to $\mathsf{chase}(P)$, then, by Definition 3.3, there is a homomorphism $h$ s.t. the substitution $h \circ \sigma'$ maps $\sigma(q)$ also to $\mathsf{pChase}(P)$. □

**Algorithm 2** ORACLE-QA$(P, q)$

**Input**: *Datalog*$^\exists$ program $P$ $\wedge$ Boolean atomic query $q$
**Output**: `true` $\vee$ `false`

```
if (IS-PARSIMONIOUS(P))
    return (pChase(P) ⊨ q)
else
    k := firstAwakeningLevel(P)
    P' := P ∪ (chase^k(P) − chase^{k−1}(P))
    return ORACLE-QA(P', q)
```

The following theorem claims that parsimony makes atomic QA decidable.

**Theorem 3.5.** *Atomic QA against Parsimonious-Sets programs is decidable.*

*Proof (Sketch).* Let $P \in$ *Datalog*$^\exists$. Proposition 3.4 ensures that atomic QA is sound and complete against $\mathsf{pChase}(P)$. Now, let $\alpha$ be the maximum arity over all predicate symbols in $P$, and $\Phi$ be a set of $\alpha$ nulls. PARSIM-CHASE termination is guaranteed by the existence of a one-to-one correspondence $\mu$ between $\mathsf{pChase}(P)$ and a subset of $\mathsf{base}(\mathsf{dom}(P) \cup \Phi)$ s.t. $\mathbf{a} \simeq \mu(\mathbf{a})$ for each $\mathbf{a} \in \mathsf{pChase}(P)$, entailing $|\mathsf{pChase}(P)| \leq |\mathsf{preds}(P)| \cdot (|\mathsf{dom}(P)| + \alpha)^\alpha$. □

We now show that recognizing parsimony is undecidable.

**Theorem 3.6.** *Checking whether a program is parsimonious is not decidable. In particular, it is* ***coRE****-complete.*

*Proof (Sketch).* Let $P \in$ *Datalog*$^\exists$. For membership, a CHASE run can semi-decide whether $P$ is not parsimonious. For hardness, we define Algorithm 2 that would solve $\text{QA}_{[Datalog^\exists]}$ (being **RE**-complete by Proposition 2.6) if the parsimony-check was decidable. In particular, IS-PARSIMONIOUS denotes the Boolean computable function deciding whether $P \in$ *Parsimonious-Sets*, while $\mathsf{firstAwakeningLevel}(P)$ the lowest level $k$ reached by the CHASE s.t. $\mathsf{pChase}(P) \Vdash \mathbf{a}$ 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, under these assumptions, Algorithm 2 would be sound and complete as well as it would always terminate. □

## 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 with a variable `X` at position $i$. This position is marked in $\mathbf{a}$ 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 `X` is either an $\exists$-variable, or a $\forall$-variable s.t. `X` occurs in $\mathsf{body}(r)$ in affected positions only. Otherwise, position $i$ is 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 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 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 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 X occurs at position $j$ in $\mathbf{b}$, if 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. $\mathsf{pred}(\mathsf{head}(r)) = \mathsf{pred}(\mathbf{a})$. $\square$

Note that $\mathsf{nullset}(i, \mathbf{a})$ may be empty. A representative null $\varphi$ *invades* a variable X that occurs at position $i$ in an atom $\mathbf{a}$ if $\varphi$ is contained in $\mathsf{nullset}(i, \mathbf{a})$. A variable X occurring in a conjunct $\mathbf{conj}$ is *attacked* in $\mathbf{conj}$ by a null $\varphi$ if each occurrence of X in $\mathbf{conj}$ is invaded by $\varphi$. A variable 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 X occurs in more than one body atom, then 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. $\square$

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 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 Y and 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 q does not belong to *Parsimonious-Sets*. $\square$

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 (Sketch).* Let $P \in$ *Shy* and $j$ be the level where PARSIM-CHASE stopped on $P$. If there is a level $k > j + 1$ with an atom $\mathbf{b}$ s.t. $\mathsf{pChase}(P) \not\Vdash \mathbf{b}$, then there must be a set $S \neq \emptyset$ of atoms from $\mathsf{chase}^{k-1}(P) - \mathsf{chase}^{k-2}(P)$ being essential for firing a rule $r$ on $\mathsf{chase}^{k-1}(P)$ to infer $\mathbf{b}$. Let us pick the smallest $k$. By Definition 3.3, for each $\mathbf{a} \in S$ there is a homomorphism $h$ s.t. $h(\mathbf{a}) \in \mathsf{pChase}(P)$. However, since $P$ is shy (see Definition 4.2), each $h(\mathbf{a})$ can be used, instead of $\mathbf{a}$, to infer an atom $\mathbf{b}' \preceq \mathbf{b}$ in $\mathsf{chase}^{k-1}(P)$. $\square$

**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 $\alpha$ 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. $\square$

### 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" $\varphi$ to constant in $\Delta_C$, 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 $\varphi$ 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 $\Delta_N$ to an extra constant in $\Delta_C$. 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 $k > 0$. $\square$

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. $\square$

We now prove that PARSIM-CHASE over *Shy* programs is complete w.r.t. CQ answering in finitely many resumptions.

**Lemma 4.9.** *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 (Sketch).* Let $P \in$ *Shy*, $q$ be a CQ, $\sigma_a \in \mathsf{ans}_P(q)$, $\sigma$ be a substitution proving that $P \models \sigma_a(q)$ holds, and $\mathbf{X}$ be only and all the $\exists$-variables of $q$ mapped by $\sigma$ to nulls. Then, there is a substitution $\sigma'$, proving that $P \models \sigma_a(q)$ holds, that maps at least one variable in $\mathbf{X}$ to a term occurring in $\mathsf{pChase}(P)$. Thus, in the worst case, to be sure that all the nulls involved by $\sigma'$ are generated, it is enough to compute $\mathsf{pChase}(P, n)$ where $n$ is the number of $\exists$-variables of $q$. Finally, $\mathsf{pChase}(P, n+1)$ contains the atoms for $\sigma'$. $\square$

**Theorem 4.10.** *Conjunctive QA over Shy is decidable.*

*Proof.* Soundness follows by Lemma 4.8, completeness by Lemma 4.9, while termination by combining Theorem 3.5 and Definition 4.7. $\square$

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 4.11.** 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 $q = \exists \mathtt{X}, \mathtt{Y}\ \mathtt{p(X,Y),u(X,Y)}$. Figure 2 shows that $q$ cannot be proved before two freezing.

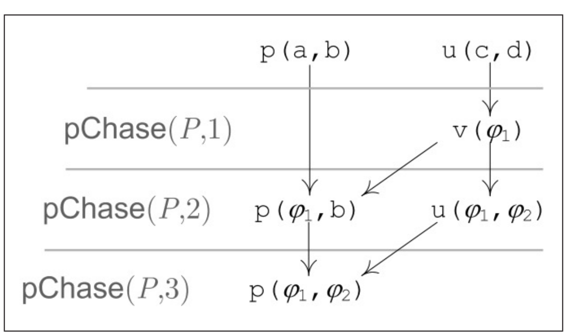


Figure 2: Snapshot of $\mathsf{pChase}(P, 3)$ w.r.t. Example 4.11

## 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. 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 (Sketch).* Let $P \in$ *Parsimonious-Sets*, $\alpha$ be the maximum arity over all predicate symbols in $P$ and $\beta$ be the maximum number of body atoms in $P$. From the bound identified in the proof of Theorem 3.5, PARSIM-CHASE performs no more than $|P - \mathsf{data}(P)| \cdot |\mathsf{preds}(P)|^{2 \cdot \beta} \cdot (|\mathsf{dom}(P)| + \alpha)^{2 \cdot \alpha \cdot \beta}$ operations. $\square$

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. $\square$

## 6 Implementation and Optimizations

We implemented a system for answering CQs 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 $P$ 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

$$\exists \mathtt{X}, \mathtt{Y}\ \mathtt{p(Z,X,W,Y)\ :\!-\ s(Z,W).}$$

is skolemized in

$$\mathtt{p}(\mathtt{Z}, t_1, \mathtt{W}, t_2)\ \mathtt{:\!-\ s(Z,W).}$$

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

**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, the facts belonging to predicates 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; Alviano et al. 2009), 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 skolemization, loading, and rewriting phases, DLV$^\exists$ 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, DLV$^\exists$ 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.

If the input query is atomic, then $\mathsf{pChase}(P)$ is sufficient to provide an answer (see Proposition 3.4); otherwise, the fixpoint computation should be resumed several times (see Lemma 4.9). 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 4.9. 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 $q$ 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$^\exists$.

**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** $|\mathbf{Y}_*|$

**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 due to various problems we had while trying to test them; as an example some of them offer no API and the only interaction is made possible by graphical, interactive, GUI making it impossible to accurately measure response times. In other cases there was no automatic tool for transforming test data in system's internal format. 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, CQ 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, coupled with the Univ-Bench ontology (Guo, Pan, and Heflin 2005). It refers to a university domain with a synthetic data generator. 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.

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 gener-

| | $Q_{all}$ | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $Q_5$ | $Q_6$ | $Q_7$ | $Q_8$ | $Q_9$ | $Q_{10}$ | $Q_{11}$ | $Q_{12}$ | $Q_{13}$ | $Q_{14}$ | # 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).

ate 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 ⊑ 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 (not reported due to space restrictions); 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 on-the-fly QA. Interestingly, even if DLV$^{\exists}$ is specifically designed for QA, 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 and Discussion

## 8.1 *Datalog*$^\exists$ 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 being 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).

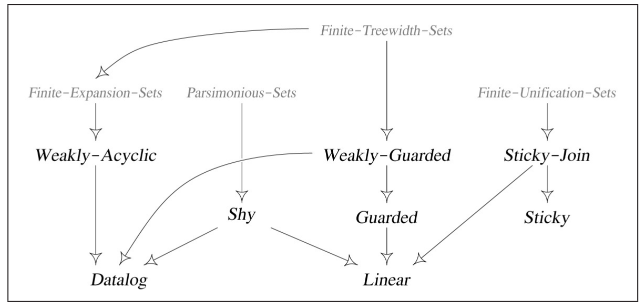


Figure 3: Taxonomy of representative *Datalog*$^\exists$ classes

Figure 3 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, and abstract classes are shown in grey.

**Theorem 8.1.** *For each pair $\mathcal{C}_1$ and $\mathcal{C}_2$ of classes represented in Figure 3, 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 *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[3] 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*. $\square$

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).
```

[3]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 $\sigma$ for a rule $r$ where $\mathbf{a} \in \sigma(\mathsf{body}(r))$.

| 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 $\texttt{QA}_{[\mathcal{C}]}$ problem

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).

Summarizing, *Shy* offers the best balance between expressivity and complexity. Conjunctive QA is efficiently computable in *Shy* (polynomial data-complexity) and, compared with other tractable *Datalog*$^\exists$ fragments, *Shy* is the only language supporting advanced properties like role-transitivity and concept-product (besides standard properties like role-hierarchy, role-inverse, concept-hierarchy). These properties are relevant in practice. More specifically, even though *Weakly-Guarded* encompasses and generalizes both *Datalog* and *Linear* as *Shy*, it has untractable data-complexity and no implementation. *Weakly-Acyclic* and *Guarded* are tractable (although they suffer of higher combined-complexity than *Shy*) but the former does not include *Linear* (even the basic "father-person" ontology cannot be represented), while the latter does include *Datalog* and does not support role-transitivity and concept-product. Moreover, no efficient implementation (such as the one proposed for *Shy*) of *Guarded* has been found so far since the natural termination condition needs a huge number of isomorphic atoms. *Sticky-Join* is suitable for an efficient implementation and captures some light-weight DL properties but, since it does not generalize *Datalog*, it cannot express important KR features like role-transitivity.

### 8.2 Ontology Reasoners

To the best of our knowledge, there is only one ongoing research work directly supporting $\exists$-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, e.g., transitivity and concept products.[4]

Since DLV$^\exists$ 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 FO semantics for unrestricted CQs; but the expressivity of their languages is limited to $\mathbf{AC}_0$ and excludes, e.g., 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 FO 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.[5] Similar to tableau-based systems, they perform full-materialization and implement expressive DLs, but they still miss to support the standard FO semantics for CQs (Glimm et al. 2008).

Summing up, it turns out that DLV$^\exists$ is the first system supporting the standard FO semantics for unrestricted CQs with $\exists$-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 QA.

## 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 classes and systems in Description Logics.

## References

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

Acciarri, A.; Calvanese, D.; De Giacomo, G.; Lembo, D.; Lenzerini, M.; Palmieri, M.; and Rosati, R. 2005. QUONTO: querying ontologies. In *Proc. of the 20th AAAI Conf. on AI*, volume 4, 1670–1671.

Alviano, M.; Faber, W.; Greco, G.; and Leone, N. 2009. Magic Sets for Disjunctive Datalog Programs. Technical Report 09/2009, Dept. of Math., Univ. of Calabria, Italy. See https://www.mat.unical.it/~faber/research/papers/TRMAT092009.pdf.

Baget, J.-F.; Leclère, M.; Mugnier, M.-L.; and Salvat, E. 2009. Extending Decidable Cases for Rules with Existential Variables. In *Proc. of the 21st IJCAI*, 677–682.

Baget, J.-F.; Leclère, M.; and Mugnier, M.-L. 2010. Walking the Decidability Line for Rules with Existential Variables. In *Proc. of the 12th KR Int. Conf.*, 466–476.

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

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

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 KR Int. Conf.*, 70–80. 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 *Proc. of the 28th PODS Symp.*, 77–86.

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 *Proc. of the 4th RR Int. Conf.*, volume 6333, 1–17.

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

Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. 2010. Enhancing ASP by Functions: Decidable Classes and Implementation Techniques. In *Proc. of the 24th AAAI Conf. on AI*, 1666–1670.

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. *Semant. 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 *Proc. of the 20th ICLP*, volume 3132, 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 Int. CIKM*. to appear.

Deutsch, A.; Nash, A.; and Remmel, J. 2008. The Chase Revisited. In *Proc. of the 27th PODS Symp.*, 149–158.

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

Glimm, B.; Horrocks, I.; Lutz, C.; and Sattler, U. 2008. Conjunctive query answering for the description logic SHIQ. *JAIR* 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 http://swat.cse.lehigh.edu/projects/lubm/.

Haarslev, V., and Möller, R. 2001. RACER System Description. In *Proc. of the 6th IJCAR*, 701–705.

Hustadt, U.; Motik, B.; and Sattler, U. 2004. Reducing SHIQ- Descrption Logic to Disjunctive Datalog Programs. In *Proc. of the 9th KR Int. Conf.*, 152–162.

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

Kloks, T. 1994. *Treewidth, Computations and Approximations*, volume 842 of *LNCS*. Springer.

Kollia, I.; Glimm, B.; and Horrocks, I. 2011. SPARQL Query Answering over OWL Ontologies. In *Proc. of the 24th DL Int. Workshop*, volume 6643 of *LNCS*. Springer. 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.

Leone, N.; Manna, M.; Terracina, G.; and Veltri, P. 2011. Efficiently Computable Datalog$^{\exists}$ Programs. Technical report, Dept. of Math., Univ. of Calabria, Italy. See www.mat.unical.it/kr2012/shy.pdf.

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

Marnette, B. 2009. Generalized schema-mappings: from termination to tractability. In *Proc. of the 28th PODS Symp.*, 13–22.

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. *JAIR* 36:165–228.

Mugnier, M.-L. 2011. Ontological query answering with existential rules. In *Proc. of the 5th RR Int. Conf.*, 2–23.

Rodriguez-Muro, M., and Calvanese, D. 2011a. Dependencies: Making Ontology Based Data Access Work in Practice. In *Proc. of the 5th AMW on Foundations of Data Management*, volume 477.

Rodriguez-Muro, M., and Calvanese, D. 2011b. Dependencies to Optimize Ontology Based Data Access. In *Description Logics*, volume 745. CEUR-WS.org.

Rosati, R., and Almatelli, A. 2010. Improving Query Answering over DL-Lite Ontologies. In *Proc. of the 12th KR Int. Conf.*, 290–300.

Rudolph, S.; Krötzsch, M.; and Hitzler, P. 2008. All Elephants are Bigger than All Mice. In *Proc. of the 21st DL Int. Workshop*, volume 353.

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 IJCAR*, volume 4130, 292–297.