Complete Query Answering Over Horn Ontologies Using a Triple Store

Yujiao Zhou, Yavor Nenov, Bernardo Cuenca Grau, and Ian Horrocks

Department of Computer Science, University of Oxford, UK

Abstract. In our previous work, we showed how a scalable OWL 2 RL reasoner can be used to compute both lower and upper bound query answers over very large datasets and arbitrary OWL 2 ontologies. However, when these bounds do not coincide, there still remain a number of possible answer tuples whose status is not determined. In this paper, we show how in the case of Horn ontologies one can exploit the lower and upper bounds computed by the RL reasoner to efficiently identify a subset of the data and ontology that is large enough to resolve the status of these tuples, yet small enough so that the status can be computed using a fully-fledged OWL 2 reasoner. The resulting hybrid approach has enabled us to compute exact answers to queries over datasets and ontologies where previously only approximate query answering was possible.

1 Introduction

An increasing number of applications rely on RDF and SPARQL for storing and querying semistructured data. The functionality of many such applications is enhanced by an OWL 2 ontology, which is used to *(i)* unambiguously specify the meaning of data in the application, *(ii)* provide the vocabulary and background knowledge needed for users to formulate accurate queries, and *(iii)* enrich query answers with information not explicitly represented in the dataset.

However, the appealing benefits of using an OWL 2 ontology come at the cost of scalability, since answering queries over OWL 2 ontologies is of very high computational complexity. Despite intensive efforts at optimisation, fully-fledged OWL 2 reasoners, such as HermiT [16], Pellet [23] and Racer [9], still fall far short of meeting the scalability demands of applications that require efficient management of large-scale RDF datasets.

To achieve more favourable scalability, a common approach is to delegate reasoning and query answering tasks to a rule-based RDF triple store. State-of-the-art triple stores such as OWLim [3], Oracle's RDF Semantic Graph [26] and RDFox[1] provide robust and scalable query answering support for ontologies in the OWL 2 RL profile [17] and datasets containing millions, or even billions, of triples. However, such triple stores are intrinsically limited in their reasoning capabilities, as they ignore (parts of) axioms in the application's ontology that

[1] `http://www.cs.ox.ac.uk/isg/tools/RDFox/`

aren't captured by OWL 2 RL. As a result, they are incomplete: for some combinations of ontology, query and data, they will fail to return all query answers.

In this paper, we propose a novel approach to query answering that addresses the scalability challenge for ontology languages beyond OWL 2 RL without giving up completeness of query answers. The key idea is to employ a hybrid technique that combines an OWL 2 RL reasoner based on a highly scalable RDF triple store (RL reasoner for short) with a fully-fledged OWL 2 reasoner (OWL reasoner for short) such that most of the computational workload can be delegated to the RL reasoner, and the OWL reasoner is used only as necessary to ensure completeness. The difficulty in realising this approach is to efficiently determine when and where fully-fledged reasoning is needed.

Our hybrid query answering technique builds on previous work [27], where we showed how an RL reasoner can be exploited to efficiently compute lower and upper bound query answers over very large datasets and arbitrary OWL 2 ontologies. When the two bounds coincide (which was often the case in the experiments reported in [27]), the query has been fully answered. When the two bounds do not coincide, however, there may still remain a significant number of possible answer tuples whose status is undetermined. In theory, the status of these tuples can be determined using an OWL reasoner, but for large-scale datasets, even checking single tuples is often infeasible in practice.

The main contribution of this paper is a technique for identifying a (typically small) subset of the dataset and ontology that is sufficient for determining the status of a possible answer tuple. The basic idea is that, starting from a query Q and a possible answer tuple $\vec{a}$, we use backward chaining with axioms from the upper-bound ontology to identify those axioms and data triples from the input ontology and dataset that might contribute to a proof that $\vec{a}$ is an answer to Q. An OWL reasoner is then used to check if the identified axioms and data triples entail that $\vec{a}$ is an answer to Q. Currently, our technique is only known to be applicable to Horn ontologies (i.e., ontologies that can be translated to first-order Horn clauses). However, many OWL 2 ontologies are Horn [6], as are all the profiles. Moreover, we conjecture that the approach can be extended to arbitrary OWL 2 ontologies; verifying this conjecture is left for future work.

Our new technique also addresses an important limitation with the approach presented in [27]: if the upper-bound ontology and dataset is unsatisfiable, then it is necessary to check the satisfiability of the input ontology and dataset, but this was impractical with large datasets. Now, we can simply use our new technique to compute the answer to the query `owl:Nothing`(x), with the ontology and dataset being satisfiable iff the answer is empty.

We have developed a reasoner that integrates RDFox and the HermiT OWL reasoner. A preliminary evaluation has shown very promising results. For instance, we can compute in reasonable time the exact answers to a range of queries over the LUBM(40) ontology and dataset—results that are far beyond the capabilities of any other OWL reasoner known to us. Our technique appears to be very effective in identifying small relevant subsets of the data and ontology: for many queries, only 2% of the data and just a few axioms from the ontology

were necessary to determine the status of all unverified answer tuples. Moreover, the effectiveness of our technique is not restricted to LUBM: we have obtained encouraging results with the Fly Anatomy ontology—a biomedical ontology containing more than 7,000 classes and 140,000 axioms—and its associated dataset.

2 Preliminaries

We adopt standard notions from first-order logic with equality, such as variables, constants, terms, atoms, formulas, sentences, substitutions, entailment (written $\models$), and (un)satisfiability. The equality atom between terms t and t' is denoted as $t \approx t'$; we use the abbreviation $t \not\approx t'$ for $\neg(t \approx t')$ (an inequality atom). The falsum atom is denoted as $\bot$ (equivalent to `owl:Nothing`), whereas the dual universal truth atom is denoted as $\top$ (equivalent to `owl:Thing`).

OWL 2 Ontologies. We assume familiarity with the normative specifications of OWL 2 and OWL 2 RL. We deviate slightly from the normative documents only in that we make an explicit distinction between schema-level and data-level axioms. We use *ontology* and *dataset* to refer to a set of schema-level and a set of data-level axioms, respectively. W.l.o.g. we assume that data assertions are given as *facts* (ground atoms), each of which corresponds to a single RDF triple. Consider as our running example the following ontology $\mathcal{O}^{ex}$ and dataset $\mathcal{D}^{ex}$.

$$\begin{aligned}
\mathcal{O}^{ex} = \{&\mathit{SubClassOf}(\mathsf{Animal}\ \mathit{SomeValuesFrom}(\mathsf{eats}\ \mathsf{Thing})), && \text{(T1)}\\
&\mathit{SubClassOf}(\mathsf{Herbivore}\ \mathit{AllValuesFrom}(\mathsf{eats}\ \mathsf{Plant})), && \text{(T2)}\\
&\mathit{DisjointClasses}(\mathsf{Herbivore}\ \mathsf{Carnivore}), && \text{(T3)}\\
&\mathit{SubClassOf}(\mathsf{Carnivore}\ \mathit{MinCardinality}(2\ \mathsf{hasParent}\ \mathsf{Thing}))) \} && \text{(T4)}\\
\mathcal{D}^{ex} = \{&\mathit{ClassAssertion}(\mathsf{Animal}\ \mathit{lion}), && \text{(A1)}\\
&\mathit{ClassAssertion}(\mathsf{Animal}\ \mathit{rabbit}), && \text{(A2)}\\
&\mathit{ClassAssertion}(\mathsf{Herbivore}\ \mathit{rabbit}), && \text{(A3)}\\
&\mathit{ClassAssertion}(\mathsf{Herbivore}\ \mathit{sheep}), && \text{(A4)}\\
&\mathit{PropertyAssertion}(\mathsf{eats}\ \mathit{sheep}\ \mathit{grass}), && \text{(A5)}\\
&\mathit{ClassAssertion}(\mathsf{Carnivore}\ \mathit{wolf}) \} && \text{(A6)}
\end{aligned}$$

Queries. A *conjunctive query* (CQ) is a first-order formula in the form of $Q(\vec{x}) = \exists \vec{y}(\varphi(\vec{x}, \vec{y}))$ with Q a distinguished query predicate and $\varphi(\vec{x}, \vec{y})$ a conjunction of atoms without inequalities. The variables in $\vec{x}$ are *distinguished*. The following CQ with a distinguished variables x asks for all individuals that eat plants:

$$Q^{ex}(x) := \exists y(\mathsf{eats}(x, y) \wedge \mathsf{Plant}(y)).$$

A tuple of constants $\vec{a}$ is a *certain answer* to $Q(\vec{x})$ w.r.t. a set of first-order sentences $\mathcal{F}$ and a set of facts $\mathcal{D}$ if $\mathcal{F} \cup \mathcal{D} \models Q(\vec{a})$. The set of all certain answers to $Q(\vec{x})$ w.r.t. $\mathcal{F}$ and $\mathcal{D}$ is denoted as $\mathsf{cert}(Q, \mathcal{F}, \mathcal{D})$. For example, the individuals *sheep* and *rabbit* are certain answers to Q^{ex} w.r.t. $\mathcal{O}^{ex}$ and $\mathcal{D}^{ex}$. We omit the

distinguished variables of $Q(\vec{x})$ and write just Q for brevity. *SPARQL conjunctive queries* are CQs with only distinguished variables.[2]

Rule languages. Rule languages are widely-used knowledge representation formalisms that have strong connections with different fragments of OWL 2 [4]. Specifically, OWL 2 RL is strongly connected to *datalog*, whereas Horn ontologies are related to datalog$^{\pm}$—an extension of datalog with existential quantifiers allowed in rule heads. For instance, our example ontology $\mathcal{O}^{ex}$ is equivalent to the following datalog$^{\pm}$ rules in which all free variables are universally quantified.

$$\exists y(\mathsf{eats}(x, y)) \leftarrow \mathsf{Animal}(x) \tag{P1}$$

$$\mathsf{Plant}(y) \leftarrow \mathsf{eats}(x, y) \wedge \mathsf{Herbivore}(x) \tag{P2}$$

$$\bot \leftarrow \mathsf{Carnivore}(x) \wedge \mathsf{Herbivore}(x) \tag{P3}$$

$$\exists y_1 \exists y_2(\mathsf{hasParent}(x, y_1) \wedge \mathsf{hasParent}(x, y_2) \wedge y_1 \not\approx y_2) \leftarrow \mathsf{Carnivore}(x) \tag{P4}$$

Formally, a datalog$^{\pm}$ rule is a first-order sentence of the following form [5]:

$$\forall \vec{x}(\exists \vec{y}(C_1 \wedge \cdots \wedge C_m) \leftarrow B_1 \wedge \cdots \wedge B_n), \tag{1}$$

where each B_j is an atom with variables in $\vec{x}$ that is neither $\bot$ nor an inequality atom, and either *(i)* $m = 1$ and $C_1 = \bot$, or *(ii)* $m \geq 1$ and, for each $1 \leq i \leq m$, C_i is an atom different from $\bot$ with free variables in $\vec{x} \cup \vec{y}$. A *datalog* rule is a rule of the form (1) with no existentially quantified variables.[3] A *datalog (resp. datalog$^{\pm}$) program* is a set of datalog (resp. datalog$^{\pm}$) rules.

Horn ontologies (i.e., ontologies that can be normalised as a set of first-order Horn clauses) can also be represented by datalog$^{\pm}$ programs. Furthermore, each OWL 2 RL ontology can be represented by a datalog program. Axioms T2 and T3 in our running example are in OWL 2 RL and can be represented by the datalog rules P2 and P3, respectively; in contrast, T1 is outside OWL 2 RL and, as such, is only expressible by a datalog$^{\pm}$ rule (in our case P1).

Datalog rules allow for easy and efficient computation of the dataset $\mathcal{D}_\Sigma$ consisting of all facts entailed by a datalog program Σ and a dataset $\mathcal{D}$. The set $\mathcal{D}_\Sigma$ is called the *materialisation* of Σ w.r.t. $\mathcal{D}$. The set of certain answers $\mathsf{cert}(Q, \Sigma, \mathcal{D})$ for an arbitrary query Q coincides with $\mathsf{cert}(Q, \emptyset, \mathcal{D}_\Sigma)$. Consider, for example, the set Σ_L comprising the datalog rules P2 and P3. The materialisation of Σ_L w.r.t. $\mathcal{D}^{ex}$ extends $\mathcal{D}^{ex}$ with the single fact $\mathit{ClassAssertion}(\mathsf{Plant}\ \mathit{grass})$; clearly, *sheep* is an answer to the query Q^{ex} w.r.t. Σ_L and $\mathcal{D}^{ex}$, but *rabbit* is not.

3 Our Approach in a Nutshell

In this paper, we propose a hybrid reasoning technique that combines an RL reasoner based on a highly scalable RDF triple store with a fully-fledged OWL

[2] In general, a SPARQL query can include non-distinguished variables; however, the semantics of SPARQL means that this is equivalent to treating all variables as distinguished and then applying a suitable projection.

[3] Our definition of datalog is slightly non-standard as it allows conjunction in rule heads; such rules can be equivalently split into multiple rules with atomic heads.

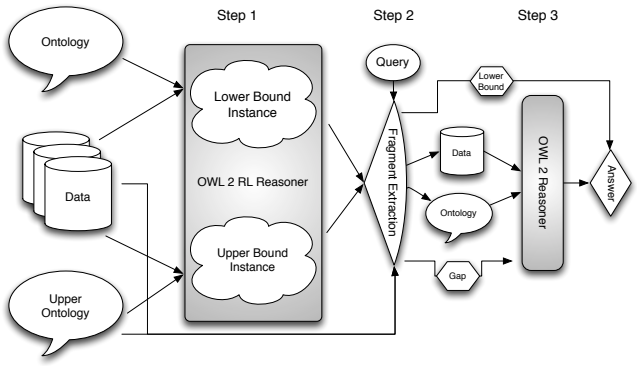

Fig. 1. Overview

reasoner. The key feature of our query answering technique is that it tries to delegate most of the computational workload to the RL reasoner, thus minimising the use of the less scalable OWL reasoner. Given a Horn OWL 2 ontology $\mathcal{O}$, a dataset $\mathcal{D}$, and a CQ Q, we compute the certain answers $\mathsf{cert}(Q, \mathcal{O}, \mathcal{D})$ in three steps, which we summarise next and schematically depict in Figure 1.

Step 1: Lower and upper bound query answers. Our first step is to compute two OWL 2 RL ontologies $\mathcal{O}_L$ (the *lower bound ontology*) and $\mathcal{O}_U$ (the *upper bound ontology*) satisfying the following property: $\mathsf{cert}(Q, \mathcal{O}_L, \mathcal{D}) \subseteq \mathsf{cert}(Q, \mathcal{O}, \mathcal{D}) \subseteq \mathsf{cert}(Q, \mathcal{O}_U, \mathcal{D})$. Since both $\mathcal{O}_L$ and $\mathcal{O}_U$ are OWL 2 RL ontologies, we can then use an RL reasoner to compute the *lower bound* $\mathsf{cert}(Q, \mathcal{O}_L, \mathcal{D})$ and the *upper bound* $\mathsf{cert}(Q, \mathcal{O}_U, \mathcal{D})$. If the set $G = \mathsf{cert}(Q, \mathcal{O}_U, \mathcal{D}) \backslash \mathsf{cert}(Q, \mathcal{O}_L, \mathcal{D})$ of tuples in the "gap" between lower and upper bound is empty, then the set of certain answers $\mathsf{cert}(Q, \mathcal{O}, \mathcal{D})$ coincides with both lower and upper bounds, in which case we don't need to resort to the OWL reasoner. This step exploits the techniques in our previous work [27], which we briefly recapitulate in Section 4.

Step 2: Computing ontology and dataset fragments. In the second step we exploit the lower and upper bound ontologies and query answers to identify (small) fragments $\mathcal{O}_f$ of $\mathcal{O}$ and $\mathcal{D}_f$ of $\mathcal{D}$ satisfying: $\mathcal{O} \cup \mathcal{D} \models Q(\vec{a})$ iff $\mathcal{O}_f \cup \mathcal{D}_f \models Q(\vec{a})$ for each $\vec{a} \in G$. Thus, $\mathcal{O}_f$ and $\mathcal{D}_f$ are sufficient for determining whether each tuple in G is indeed a certain answer to Q. The fragments $\mathcal{O}_f$ and $\mathcal{D}_f$ depend on both the input query Q and the tuples in G. This novel technique is the main contribution of our paper, and it is described in Section 5.

Step 3: Calling the OWL reasoner. In the final step we resort to the OWL reasoner to verify whether $\mathcal{O}_f \cup \mathcal{D}_f \models Q(\vec{a})$ for each tuple $\vec{a} \in G$. We return as certain answers the union of the lower bound and the verified tuples in G: $\mathsf{cert}(Q, \mathcal{O}, \mathcal{D}) = \mathsf{cert}(Q, \mathcal{O}_L, \mathcal{D}) \cup \{\vec{a} \in G \mid \mathcal{O}_f \cup \mathcal{D}_f \models Q(\vec{a})\}$.

4 Lower and Upper Bound Query Answers

In our previous work [27] we showed how an RL reasoner can be used to efficiently compute upper and lower bound query answers over arbitrary OWL 2 ontologies.

In this section, we recapitulate the techniques proposed there. Our description will be rather informal, and we refer the interested reader to [27] for details.

Lower bound answers. RL reasoners are flexible enough to process arbitrary ontologies on a "best efforts" basis; that is, the reasoner ignores (parts of) the axioms that are outside OWL 2 RL, thus effectively reasoning with a lower bound ontology $\mathcal{O}_L$. RL reasoners are guaranteed to be sound (i.e., $\mathcal{O} \models \mathcal{O}_L$), and hence all the tuples they compute are indeed certain answers; we can therefore compute lower bound answers simply by running the RL reasoner as a "black box" on the input Q, $\mathcal{O}$, and $\mathcal{D}$. For instance, when given our example ontology $\mathcal{O}^{ex}$, dataset $\mathcal{D}^{ex}$, and query Q^{ex}, a typical RL reasoner will reduce $\mathcal{O}^{ex}$ to the OWL 2 RL ontology $\mathcal{O}_L^{ex} = \{T2, T3\}$, and will compute $\mathsf{cert}(Q^{ex}, \mathcal{O}_L^{ex}, \mathcal{D}^{ex}) = \{sheep\}$.

Upper bound answers. We transform $\mathcal{O}$ into an OWL 2 RL ontology $\mathcal{O}_U$ such that $\mathcal{O}_U \models \mathcal{O}$. First, $\mathcal{O}$ is normalised into a datalog$^{\pm}$ program $\Sigma^{\pm}$ using a variant of the structural transformation of first-order logic (see [16,27]). For instance, our example ontology $\mathcal{O}^{ex}$ can be normalised into the datalog$^{\pm}$ program consisting of rules P1-P4. The crucial second step is the transformation of the resulting datalog$^{\pm}$ program into a (stronger) datalog program Σ_U satisfying $\Sigma_U \models \mathcal{O}$; roughly speaking, Σ_U is obtained by Skolemising all existential quantifiers into fresh constants. For example, the datalog$^{\pm}$ rules P1 and P4 get transformed into the rules D1, D4 and D5 to give the following datalog program:

$$\begin{aligned}
\Sigma_U^{ex} = \{&\mathsf{eats}(x, c) \leftarrow \mathsf{Animal}(x), && \text{(D1)}\\
&\mathsf{Plant}(y) \leftarrow \mathsf{eats}(x, y) \wedge \mathsf{Herbivore}(x), && \text{(D2)}\\
&\bot \leftarrow \mathsf{Carnivore}(x) \wedge \mathsf{Herbivore}(x), && \text{(D3)}\\
&\mathsf{hasParent}(x, c_1) \wedge \mathsf{hasParent}(x, c_2) \leftarrow \mathsf{Carnivore}(x), && \text{(D4)}\\
&\bot \leftarrow c_1 \approx c_2\}. && \text{(D5)}
\end{aligned}$$

Finally, the datalog program Σ_U is transformed into the upper bound OWL 2 RL ontology $\mathcal{O}_U$, where $\mathcal{O}_U \models \Sigma_U$; roughly speaking, each rule in Σ_U is transformed into an OWL 2 RL axiom by "rolling up" the rule's body and head into class descriptions, while possibly introducing fresh predicates in order to satisfy the syntactic restrictions of OWL 2 RL. For instance, the datalog rules D1–D5 in our running example are transformed into the following OWL 2 RL axioms:

$$\begin{aligned}
\mathcal{O}_U^{ex} = \{&\mathit{SubClassOf}(\mathsf{Animal}\ \mathit{HasValue}(\mathsf{eats}\ c)), && \text{(R1)}\\
&\mathit{SubClassOf}(\mathsf{Herbivore}\ \mathit{AllValuesFrom}(\mathsf{eats}\ \mathsf{Plant})), && \text{(R2)}\\
&\mathit{DisjointClasses}(\mathsf{Herbivore}\ \mathsf{Carnivore}), && \text{(R3)}\\
&\mathit{SubClassOf}(\mathsf{Carnivore}\ \mathit{HasValue}(\mathsf{hasParent}\ c_1)), && \text{(R4.1)}\\
&\mathit{SubClassOf}(\mathsf{Carnivore}\ \mathit{HasValue}(\mathsf{hasParent}\ c_2)), && \text{(R4.2)}\\
&\mathit{DifferentFrom}(c_1\ c_2)\}. && \text{(R4.3)}
\end{aligned}$$

As a result, we obtain that $\mathcal{O}_U \models \mathcal{O}$ and hence $\mathsf{cert}(Q, \mathcal{O}, \mathcal{D}) \subseteq \mathsf{cert}(Q, \mathcal{O}_U, \mathcal{D})$. Clearly, the transformation of $\mathcal{O}$ into the upper bound ontology $\mathcal{O}_U$ will in general introduce consequences that are not entailed by the original ontology $\mathcal{O}$. To

see this, consider again our running example. The axioms R1, R2, A1 and A2 entail $\mathit{ObjectPropertyAssertion}(\mathsf{eats}\ \mathit{rabbit}\ c)$, $\mathit{ObjectPropertyAssertion}(\mathsf{eats}\ \mathit{lion}\ c)$ and $\mathit{ClassAssertion}(\mathsf{Plant}\ c)$. We thus get that (in addition to *sheep*) *rabbit* and *lion* are also answers to Q^{ex}, i.e. $\mathsf{cert}(Q^{ex}, \mathcal{O}_U^{ex}, \mathcal{D}^{ex}) = \{\mathit{sheep}, \mathit{rabbit}, \mathit{lion}\}$. However, we have that $\mathit{lion} \notin \mathsf{cert}(Q^{ex}, \mathcal{O}^{ex}, \mathcal{D}^{ex})$.

The final transformation from Σ_U into $\mathcal{O}_U$ is only required if the RL reasoner to be used only accepts OWL 2 RL ontologies; our RL reasoner RDFox can handle datalog rules natively, and this transformation can be dispensed with.

Dealing with unsatisfiability. An important limitation with the approach presented in [27] is that, given an ontology $\mathcal{O}$ and a dataset $\mathcal{D}$, if $\mathcal{O}_L \cup \mathcal{D}$ is satisfiable (i.e., $\mathsf{cert}(\bot(x), \mathcal{O}_L, \mathcal{D}) = \emptyset$) but $\mathcal{O}_U \cup \mathcal{D}$ is unsatisfiable (i.e., $\mathsf{cert}(\bot(x), \mathcal{O}_U, \mathcal{D}) \neq \emptyset$), we must check if $\mathcal{O} \cup \mathcal{D}$ is satisfiable; if $\mathcal{O} \cup \mathcal{D}$ is satisfiable, we can still use the above procedure to compute upper bound answers, but if $\mathcal{O} \cup \mathcal{D}$ is *not* satisfiable, then everything is entailed and the (upper bound) answer to any query is trivially the set of all tuples of the appropriate arity that can be formed from individuals in $\mathcal{D}$. The difficulty is that, when $\mathcal{D}$ is large, it may be impractical to check the satisfiability of $\mathcal{O} \cup \mathcal{D}$ using an OWL reasoner.

We can now address this issue by using our new hybrid query answering technique: if $\mathsf{cert}(\bot(x), \mathcal{O}_L, \mathcal{D}) = \emptyset$, but $\mathsf{cert}(\bot(x), \mathcal{O}_U, \mathcal{D}) \neq \emptyset$, then in Step 2 we will compute fragments $\mathcal{O}_f$ and $\mathcal{D}_f$ for $\bot(x)$, and in Step 3 we will use these fragments with an OWL reasoner to compute $\mathsf{cert}(\bot(x), \mathcal{O}, \mathcal{D})$. Clearly, $\mathcal{O} \cup \mathcal{D}$ is satisfiable iff $\mathsf{cert}(\bot(x), \mathcal{O}, \mathcal{D}) = \emptyset$.

5 Computing Ontology and Dataset Fragments

Given an input ontology $\mathcal{O}$, a dataset $\mathcal{D}$ and a set of possible answer tuples G, our goal is to compute an ontology $\mathcal{O}_f \subseteq \mathcal{O}$ and a dataset $\mathcal{D}_f \subseteq \mathcal{D}$ such that

- $\mathcal{O}_f \cup \mathcal{D}_f \models Q(\vec{a})$ iff $\mathcal{O} \cup \mathcal{D} \models Q(\vec{a})$ for each tuple $\vec{a} \in G$; and
- $\mathcal{O}_f \cup \mathcal{D}_f$ is as small as possible.

5.1 Overview

In a nutshell, our technique for computing $\mathcal{O}_f$ and $\mathcal{D}_f$ works as follows.

1. We consider the upper bound datalog rules Σ_U and, for each $\vec{a} \in G$, we compute all (minimal) proofs of $Q(\vec{a})$ in $\Sigma_U \cup \mathcal{D}$. Specifically, we consider "backwards chaining" proofs based on SLD-resolution.
2. We define $\mathcal{D}_f$ (resp. Σ_f) as the set of facts in $\mathcal{D}$ (resp. rules in Σ_U) that have been used in some SLD-resolution proof for some $\vec{a} \in G$.
3. Finally, we "trace back" the rules in $\Sigma_f \subseteq \Sigma_U$ to the OWL 2 axioms $\mathcal{O}_f \subseteq \mathcal{O}$ from which they were derived.

To illustrate this process, let us consider our example ontology $\mathcal{O}^{ex}$, data set $\mathcal{D}^{ex}$ and query Q^{ex}. In this case, we have $\{\mathit{sheep}\}$ as the lower bound answer and $\{\mathit{sheep}, \mathit{rabbit}, \mathit{lion}\}$ as the upper bound answer; our goal is thus to determine

$S_0 := \mathsf{eats}(rabbit, y) \wedge \mathsf{Plant}(y)$

$S_1 := \mathsf{Plant}(c) \wedge \mathsf{Animal}(rabbit)$ via $D1$

$S_2 := \mathsf{Animal}(rabbit) \wedge \mathsf{eats}(x, c) \wedge \mathsf{Herbivore}(x)$ via $D2$

$S_3 := \mathsf{eats}(x, c) \wedge \mathsf{Herbivore}(x)$ via $A2$

$S_4 := \mathsf{Herbivore}(x) \wedge \mathsf{Animal}(x)$ via $D1$

$S_5 := \mathsf{Animal}(rabbit)$ via $A3$

$S_6 := \top$ via $A2$

Fig. 2. A proof of $Q^{ex}(rabbit)$ in $\Sigma_U^{ex} \cup \mathcal{D}^{ex}$.

whether *rabbit* and *lion* are indeed certain answers. To this end, we consider the upper bound datalog program Σ_U^{ex}, and inspect all the "backwards chaining" proofs for $Q^{ex}(rabbit)$ and $Q^{ex}(lion)$ in $\Sigma_U^{ex} \cup \mathcal{D}^{ex}$.

An example of such proof for $Q^{ex}(rabbit)$ is given in Figure 2. Starting from the goal $Q^{ex}(rabbit) = \mathsf{eats}(rabbit, y) \wedge \mathsf{Plant(y)}$, we can use rule D1 and the unifier $\{x \mapsto rabbit, y \mapsto c\}$ to obtain the subgoal S_1, which, together with D1, entails S_0. Then, we can use rule D2 and the unifier $\{y \mapsto c\}$ to obtain from S_1 the new subgoal S_2. The first conjunct in S_2 can be eliminated using the fact A2 in $\mathcal{D}^{ex}$ to produce S_3. From S_3 we can use again rule D1 to produce S_4. The first conjunct in S_4 can be eliminated using fact A3 in $\mathcal{D}^{ex}$ and finally we can obtain the empty goal by subsequently using fact A2 to eliminate the remaining atom. We have now shown that $\{D1, D2, A2, A3\} \models Q^{ex}(rabbit)$; therefore, facts A2 and A3 must be included in $\mathcal{D}_f$, and axioms T1 and T2 from $\mathcal{O}^{ex}$, from which rules D1 and D2 were (respectively) derived, must be included in $\mathcal{O}_f$.

To identify all the axioms and facts in $\mathcal{O} \cup \mathcal{D}$ that are relevant to $Q^{ex}(rabbit)$ and $Q^{ex}(lion)$, we need to consider all their possible backwards chaining proofs. By doing so, we can show that only axioms T1 and T2, and facts A1, A2 and A3 are (possibly) relevant to determining the status of *rabbit* and *lion*.

5.2 Technical Approach

We start by formalising backwards chaining proofs based on SLD-resolution.

Definition 1. *A* goal *is a is a conjunction of function-free atoms* $A_1 \wedge \ldots \wedge A_m$. *The* SLD-resolution rule *takes as premises a goal and a datalog rule, and it produces a new goal as follows*

$$\frac{A_1 \wedge \ldots \wedge A_m, \qquad C_1 \wedge \ldots \wedge C_q \leftarrow B_1 \wedge \ldots \wedge B_p}{A_2\theta \wedge \ldots \wedge A_m\theta \wedge B_1\theta \wedge \ldots, B_p\theta}$$

where θ *is the most general unifier of* A_1 *and* C_j *for some* $1 \leq j \leq q$. *The new goal, together with the rule entail the original goal.*

Let G_0 *be a goal and* Γ *be a datalog program. An* SLD-proof *of* G_0 *in* Γ *is a sequence of goals* $G_0 \overset{r_1,\theta_1}{\rightsquigarrow} G_1 \rightsquigarrow \ldots \rightsquigarrow G_{n-1} \overset{r_n,\theta_n}{\rightsquigarrow} G_n$, *where* $G_n = \top$ *and each*

G_{i+1} is obtained from G_i and rule $r_{i+1} \in \Gamma$ by means of a single SLD-resolution with substitution θ_{i+1}.

Finally, we say that a rule r is relevant *for G_0 in Γ if there exists an SLD-proof of G_0 in Γ involving r.*

SLD-resolution is sound and complete for datalog: for each datalog program Γ, conjunctive query $Q(\vec{x}) = \exists \vec{y}(\varphi(\vec{x}, \vec{y}))$ and tuple of constants $\vec{a}$ we have $\Gamma \models Q(\vec{a})$ iff there exists an SLD-proof of the goal $\varphi(\vec{a}, \vec{y})$ in Γ.

We are now ready to define the relevant fragments $\mathcal{O}_f \subseteq \mathcal{O}$ and $\mathcal{D}_f \subseteq \mathcal{D}$ that we can use to verify the answers in G using an OWL reasoner.

Definition 2. *Let Q, $\mathcal{O}$ and $\mathcal{D}$ be the input CQ, Horn ontology and dataset respectively, let Σ_U be the upper bound datalog program for $\mathcal{O}$, and let $\Xi(\cdot)$ be the function mapping each axiom in $\mathcal{O}$ into its corresponding set of rules in Σ_U. Finally, let G be the set of tuples between lower and upper bound. The (Q, G)-*relevant fragments *$\mathcal{O}_f$ of $\mathcal{O}$ and $\mathcal{D}_f$ of $\mathcal{D}$ are defined as follows:*

$$\mathcal{O}_f = \{\alpha \in \mathcal{O} \mid \exists \vec{a} \in G \text{ and } \exists r \in \Xi(\alpha) \text{ s.t. } r \text{ is relevant for } Q(\vec{a}) \text{ in } \Sigma_U \cup \mathcal{D}\},$$
$$\mathcal{D}_f = \{\alpha \in \mathcal{D} \mid \exists \vec{a} \in G \text{ s.t. } \alpha \text{ is relevant for } Q(\vec{a}) \text{ in } \Sigma_U \cup \mathcal{D}\}.$$

The correctness of our approach is established by the following theorem.

Theorem 1. *Let $\mathcal{O}_f$ and $\mathcal{D}_f$ be the (Q, G)-relevant fragments of $\mathcal{O}$ and $\mathcal{D}$, respectively. Then, $\mathcal{O} \cup \mathcal{D} \models Q(\vec{a})$ iff $\mathcal{O}_f \cup \mathcal{D}_f \models Q(\vec{a})$ for each $\vec{a} \in G$.*

The proof of Theorem 1 is rather lengthy and can be found in Appendix A. The idea behind the proof is, however, quite straightforward. The 'if' direction follows directly from the fact that $\mathcal{O}_f \cup \mathcal{D}_f \subseteq \mathcal{O} \cup \mathcal{D}$. For the 'only if' direction, assume that $\mathcal{O} \cup \mathcal{D} \models Q(\vec{a})$. From $\Sigma_U \models \mathcal{O}$ it follows that $\Sigma_U \cup \mathcal{D} \models Q(\vec{a})$. As a result, we can conclude from the completeness of SLD-resolution that there exists an SLD proof of $Q(\vec{a})$ in $\Sigma_U \cup \mathcal{D}$, and let R be the set of rules used in this proof. By construction, the axioms in $\mathcal{O}$ and facts in $\mathcal{D}$ that correspond to rules in R are contained in $\mathcal{O}_f$ and $\mathcal{D}_f$ respectively, and it can be further shown that these axioms entail $Q(\vec{a})$.

5.3 An Optimised Backward Chaining Algorithm

Computing all SLD-proofs for each answer in the gap between lower and upper bounds can be infeasible in practice with a naive backward chaining algorithm. Indeed, our problem is more challenging than typical backward chaining reasoning in datalog, where computing just a single proof suffices to verify the goal.

In this section, we describe an optimised algorithm for computing the set $R_{\vec{a}}$ of all rules that appear in an SLD proof of $Q(\vec{a})$ in $\Sigma_U \cup \mathcal{D}$, where the facts in $\mathcal{D}$ are treated as rules with an empty body, i.e., of the form $A(\vec{a}) \leftarrow$. To ensure termination of backward chaining, we apply the well-known tabling technique [24,22]. To improve performance, we use an aggressive pruning technique that exploits the upper and lower bound ontologies to detect irrelevant branches in

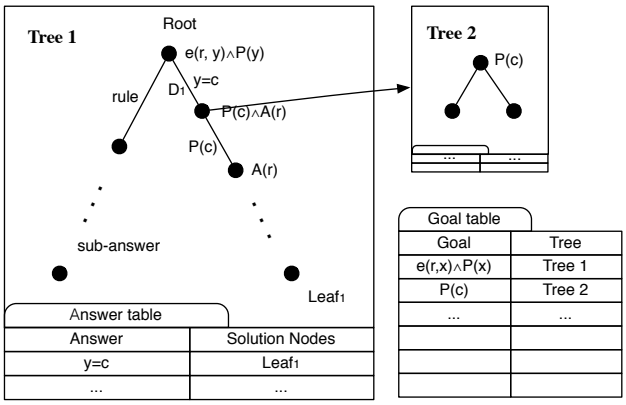

Fig. 3. Data Structure

the backward chaining tree. In the remainder of this section, we describe the specifics of our implementation of backward chaining.

Backward chaining with tabling. Our implementation of backward chaining with tabling is based on the techniques described in [22]. We deviate from [22] in that our algorithm only terminates once all SLD-proofs of the goal are computed, and also in that we keep track of all rules used in such proofs.

We first describe our data structures. To keep track of all SLD proofs of a goal $Q(\vec{a})$ and all rules that occur in them, we maintain a labelled tree t_A for each encountered subgoal A consisting of a single atom; additionally, we maintain a similar tree for the original goal $Q(\vec{a})$ (see Figure 3). The tree t_A encodes all proofs of A; each node of t_A is labelled with a goal and each edge of t_A is labelled with a pair of a substitution and a datalog rule. The labels of each edge and the nodes that it connects encode an SLD resolution step, and a branch in t_A encodes an SLD derivation. We associate with t_A an *answer table*, which will eventually map each grounding of the root goal that proves it to the list of all relevant leaf nodes in t_A for that grounding. We also maintain a global *goal table* mapping each relevant subgoal to its corresponding tree (c.f. Figure 3). Finally, we say that a node u with associated goal $A_1 \wedge \ldots \wedge A_n$ is *linked* to a tree t if the first atom A_1 in the goal corresponds to the root of t (e.g. node D_1 in Figure 3 is linked to Tree 2); we use the goal table to check whether u is linked to t.

The backward chaining algorithm is initialised with a tree $t_{Q(\vec{a})}$ consisting of just a root node labelled with the goal $Q(\vec{a})$; then, we add an entry in the Goal table mapping $Q(\vec{a})$ to $t_{Q(\vec{a})}$ and we associate with $t_{Q(\vec{a})}$ an empty answer table. After this initialisation step, we evaluate the root node of $t_{Q(\vec{a})}$ using the recursive procedure described below, which takes as input a node u in a tree t.

Case 1. If u is labelled with $\top$, we have reached a proof of the goal $A(\vec{x})$ labelling the root of t. We take $A(\vec{a})$ to be the grounding of $A(\vec{x})$ using the composition of the substitutions on the path between the root of t and the node u. We add u to the list associated with $A(\vec{a})$ in the answer table of t. We then try to resolve $A(\vec{a})$ with the goal of every node that is linked to t.

If we can resolve $A(\vec{a})$ with the goal of such a node v using a substitution θ, we add a child node v' to v, we label v with the resolvent, and we label the edge between v and v' with the pair $\langle A(\vec{a}), \theta \rangle$. We then recursively evaluate the node v'.

Case 2. If u is the root node of t, we resolve its goal with all possible rules, we create a child node for each of the resolvents, label the new nodes and edges accordingly, and recursively evaluate each of the child nodes.

Case 3. Otherwise, let $A_1 \wedge \ldots \wedge A_m$ be the goal of u.

3.1. If A_1 has not been tabled yet, i.e. it is not in the goal table, we initialise a tree t' with root node v, label v with A_1, add an entry $\langle A_1, t' \rangle$ to the goal table linking u to t, and recursively process v;

3.2. otherwise, we retrieve the available answers for A_1 from its associated tree, resolve u with those answers, create a child node for each of the resolvents, and recursively compute each added node.

Pruning. To improve performance, we apply a pruning technique that exploits the lower and upper bound ontologies. Let v be a node with associated goal $Q_v := A_1 \wedge \ldots \wedge A_m$. Before recursively evaluating v, we proceed as follows.

- If $\mathsf{cert}(Q_v, \mathcal{O}_U, \mathcal{D}) = \emptyset$, we terminate the evaluation of the current node as this branch cannot lead to a proof. This is due to the fact that the rules used in the backward chaining algorithm are logically equivalent to $\mathcal{O}_U \cup \mathcal{D}$.
- If Q_v contains no variables, and $\mathsf{cert}(Q_v, \mathcal{O}_L, \mathcal{D}) \neq \emptyset$, we create a child node for v, label it with $\top$, and we label the new edge with the empty substitution and the set of atoms $A_1, \ldots, A_m$. We recursively evaluate the new node, after which we terminate the evaluation of v. We can do so because $\mathcal{O}_L \cup \mathcal{D} \models Q_v$, and we know that the current branch will lead to exactly one proof.
- Otherwise, if A_1 contains no variable and $\mathsf{cert}(A_1, \mathcal{O}_L, \mathcal{D}) \neq \emptyset$, we create a child node for v, we label it with the goal $A_2 \wedge \ldots \wedge A_m$, label the new edge with the empty substitution and the atom A_1, and recursively evaluate the new node, after which we terminate the evaluation of v. We can do so because $\mathcal{O}_L \cup \mathcal{D} \models A_1$, and because v has no other children.

Rule Extraction. The backward-chaining evaluation of the goal $Q(\vec{a})$ results in a forest of trees encoding all possible proofs of $Q(\vec{a})$ in $\Sigma_U \cup \mathcal{D}$. In addition to all proofs of $Q(\vec{a})$, the forest also contains many superfluous derivations that should be ignored. We now describe an algorithm that traverses the forest and extracts the set $R_{\vec{a}}$ of all rules that participate in proofs of $Q(\vec{a})$. The algorithm builds the set $R_{\vec{a}}$ by carrying out a bottom-up, breath-first search on the nodes in the forest whose goals appear in proofs of $Q(\vec{a})$. It proceeds as follows.

Step 1 Initialise a set N with all solution nodes in the answer table of $t_{Q(\vec{a})}$.

Step 2 While N is not empty, remove from N a node v with a goal $A_1 \wedge \cdots \wedge A_m$. If v has a parent, do the following:

2.1. Add the parent of v to N.

2.2. If v is a resolvent of its parent and a rule $r \in \Sigma_U \cup \mathcal{D}$, add r to $R_{\vec{a}}$.

2.3. If v is a resolvent of its parent and an answer A from a tree t, retrieve all solution nodes for A in t and add them to N.

Table 1. Statistics for datasets

Data	DL	Horn	Existential	Classes	Properties	Axioms	Individuals	Dataset
LUBM(n)	$\mathcal{SHI}$	Yes	8	43	32	93	$1.7 \times 10^4 n$	$10^5 n$
FLY	$\mathcal{SRI}$	Yes	8,396	7,533	24	144,407	1,606	6,308

Table 2. Results for LUBM(40)

Query	$\lvert V\rvert$	n	$\lvert G\rvert$	t_f	$\lvert\mathcal{O}_f\rvert$	$\lvert\mathcal{D}_f\rvert$	t_{check}	t_{total}
M_1	2	3	39	36.4	6	29041	H: 23.3	H: 60.7
M_2	3	4	1	37.1	6	29004	H: 4.0	H: 42.1
M_3	4	6	16	38.2	6	29054	H: 8.4	H: 47.6
M_4	2	3	30	36.0	6	29032	H: 23.3	H: 60.2
M_5	3	4	4	39.4	6	29010	H: 24.0	H: 64.3
M_6	4	6	29	2,845.8	10	87209	H: 483.0	H: 3339.4
M_7	3	5	15	38.0	6	29033	H: 10.3	H: 49.3
M_8	3	5	14	39.3	6	29038	H: 11.9	H: 52.2
M_9	3	4	10	328.9	12	86785	H: 556.2	H: 886.7
S	1	2	39	310.0	12	86802	H: 1,780.0 P: 16,592.1	H: 2,126.5 P: 16,870.0

6 Evaluation

We have developed a prototype reasoner to carry out a preliminary evaluation. Our prototype integrates the RL reasoner RDFox[4] and an OWL reasoner, which in our case can be either HermiT[16] or Pellet[23]. RDFox is used to compute lower and upper bound query answers (c.f. Step 1 in Section 3), as well as to assist with pruning during backward chaining (c.f. Section 5.3). The OWL reasoner is used with the fragments computed by the backward chaining algorithm (c.f. Step 3 in Section 3) to determine the status of any tuples in the gap.

RDFox is a in-memory triple store that supports OWL 2 RL and datalog reasoning, and uses shared memory parallel reasoning for increased efficiency and scalability. An important feature of RDFox is its rapid query response time—this is particularly relevant during backward chaining, where queries are used in a crucial pruning optimisation. HermiT and Pellet are well-established OWL reasoners that provide support for CQ answering. HermiT can answer tree-shaped CQs with a single answer variable; Pellet supports SPARQL CQs.

In our experiments we have used LUBM and the Fly Anatomy ontology as test sets; their key features are summarised in Table 1. All tests were performed on a 14 core 3.30GHz Intel Xeon E5-2643 with 125GB of RAM, and running Linux 2.6.32. All times are given in seconds.

Evaluation results for LUBM. LUBM is a well-known benchmark ontology that comes with a predefined dataset generator parameterised by the number of universities. We tested our reasoner on LUBM(40), which contains in its dataset

[4] `http://www.cs.ox.ac.uk/isg/tools/RDFox/`

over 4 million facts about 40 universities. We used the 14 standard LUBM queries as well as 78 synthetic queries generated using SyGENiA [12].

Using RDFox, we were able to compute lower and upper bound answers for all 92 queries in less than 20s (c.f. Step 1 in Section 3). As the focus of this paper is on checking the tuples in the gap between the two bounds of a given query, we concentrate our attention on those queries whose bounds do not coincide. Only 6 queries show a non-empty gap ($Q_3, Q_{45}, Q_{51}, Q_{64}, Q_{67}, Q_{69}$)—all of them SyGENiA generated—and, due to the relatively simple nature of the LUBM ontology, it is inevitable that these queries look very similar. Since the generated queries tend to produce unrealistically large answers, we added some additional terms to those queries in order to make them more specific and thus return smaller answers. The resulting 10 "non-trivial" queries, denoted by M_1–M_9 and S, are all tree-shaped CQs, with S being the only SPARQL CQ.[5] Performance on these queries is summarised in Table 2, where $|V|$, n and $|G|$ denote the number of variables, the number of triple patterns and the number of gap tuples for each query respectively; t_f, $|\mathcal{O}_f|$ and $|\mathcal{D}_f|$ denote the time needed to compute the relevant fragments $\mathcal{O}_f$ and $\mathcal{D}_f$ using backward chaining, and their respective sizes; t_{check} denotes the total time required for checking all the tuples in G using the OWL reasoner (c.f. Step 3 in Section 3); and t_{total} denotes the total time for answering the query. We have presented timings for Hermit (H) on all queries, and Pellet (P) on the single SPARQL CQ.

We can observe that backward chaining times were rather modest for all queries but M_6, for which the computed backward chaining tree had a very large branching factor. We can also observe that for all queries the dataset fragment $\mathcal{D}_f$ contains only about 2% of the facts in LUBM(40), while $\mathcal{O}_f$ contained just a few schema-level axioms. This significant reduction in size made it possible for HermiT to verify answer tuples in reasonable time; indeed, query answering for LUBM(40) is far beyond the capabilities of HermiT or Pellet, and we were unable to verify even a single answer tuple using either OWL reasoner over the original ontology and dataset. A standard optimisation applied by RL reasoners such as OWLim is to classify the original ontology first and add the entailed subsumption axioms in $\mathcal{O}_L$. Although this optimisation closes the gap between the lower and upper bounds for query S, allowing, for example, OWLim to compute all answers for S, it has no effect on queries M_1–M_9.

To test the scalability of our reasoner, we have also evaluated queries M_1, M_6, M_9 and S against the datasets LUBM(1)–LUBM(40) (results for queries M_2–M_5, M_7 and M_8 are similar to those for M_1). The results of the evaluation are summarised in Figure 3, which shows the timings and memory usage of our reasoner for the different datasets.

Evaluation results for FLY. Fly Anatomy is a realistic and complex ontology describing the anatomy of flies, which comes with a dataset containing more than $6,000$ facts. We have tested our system using 5 realistic queries provided by the domain experts who are developing the ontology; all are CQs with non-distinguished variables, so we were only able to use HermiT in the evaluation.

[5] All test queries are available at `http://tinyurl.com/ccmwvc6`.

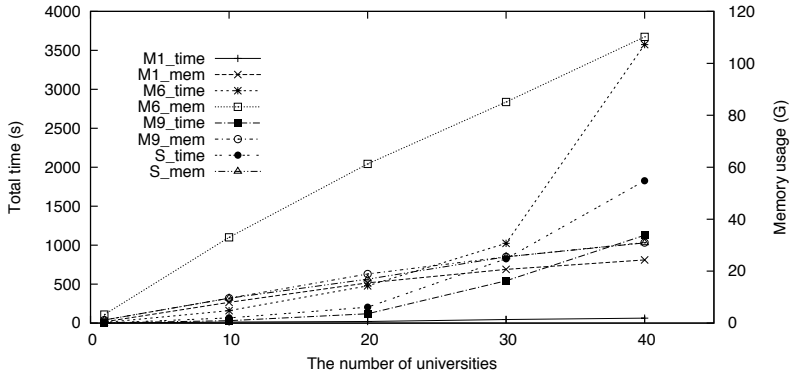

Fig. 4. Scalability test for LUBM(1)-LUBM(40)

Table 3. Results for FLY

Query	$\|V\|$	n	$\|G\|$	t_f	$\|\mathcal{O}_f\|$	$\|\mathcal{D}_f\|$	t_{check} (H)	t_{total}	t_{HermiT}
Q_1	2	3	803	108.9	224	4515	45.9	155.2	3,465.9
Q_2	3	5	342	97.7	224	4054	16.0	114.0	3,179.0
Q_3	1	1	28	91.0	217	3712	0.9	92.3	5,863.3
Q_4	2	3	25	94.3	233	3762	4.7	99.2	2,944.3
Q_5	2	2	518	100.3	222	3712	24.0	124.6	3,243.7

RDFox was able to compute lower and upper bound answers for all 5 queries in less than 15s, with the bounds being different in all cases. Our results are summarised in Table 3; columns 1–9 are the same as in Table 2, and column 10 gives the time taken for HermiT to answer the query—in contrast to the case of LUBM(40), HermiT is able to answer all these queries directly. In each case, $\mathcal{O}_f$ contained less than 0.2% of all the schema-level axioms; in contrast, $\mathcal{D}_f$ contained about 60% of the facts in the dataset. The reduction in number of schema-level axioms had a significant effect on performance: our reasoner took less than 200s per query, whereas HermiT required more than $3,000s$ per query.

7 Related Work

In recent years there has been a growing interest in the problem of query answering over ontologies and large-scale datasets. Some OWL 2 reasoners, such as HermiT, Pellet and RACER, support query answering, but despite intensive efforts at optimisation they can only deal with modestly-sized datasets [14,15,10].

The idea of using a rule-based engine for query answering over ontologies in the description logic $\mathcal{SHIQ}$ was proposed by Hustadt et al. [11] and implemented in the KAON2 system. The transformation of the ontology, however, results in a disjunctive datalog program which is exponential in the worst case. An

alternative approach based on tableaux reasoning and data summarisation was implemented in the reasoner SHER, which is complete for the description logic $\mathcal{SHIN}$, but (effectively) supports only SPARQL CQs [7].

Many specialised query answering techniques have been developed for ontologies in the QL and RL profiles of OWL 2. RL reasoners such as OWLim, Oracle's and RDFox are based on forward-chaining reasoning. QL reasoners such as QuOnto [1], Presto [21], and Quest [20] are based on query rewriting. These reasoners are, however, incomplete for ontologies outside the relevant profile. Query rewriting techniques have been extended to more expressive Horn Description Logics, and implemented in systems such as REQUIEM [19] and Clipper [8].

The idea of combining a profile-specific reasoner with a fully-fledged OWL 2 reasoner was proposed in [2], but only for ontology classification. The idea of transforming the ontology, data, and/or query to obtain upper bound query answers has also received some attention in the Semantic Web literature. In addition to our own previous work [27], approximations into OWL 2 QL [13,18] and into Datalog [25] have also been explored; however, all these techniques are worst case exponential, and the question of how to deal with cases where upper and lower bounds do not coincide was not considered.

8 Conclusion

In this paper we have described a hybrid approach for complete query answering over Horn OWL 2 ontologies. Our technique combines a scalable OWL 2 RL reasoner with a fully-fledged OWL 2 reasoner s.t. most of the computational workload is delegated to the RL reasoner, with the OWL 2 reasoner being used only as necessary to ensure completeness. We have implemented a prototype reasoner that integrates the RL reasoner RDFox and the OWL 2 reasoner HermiT. A preliminary evaluation of our prototype produced very promising results: we managed to compute in reasonable time the exact answers to a range of queries over LUBM(40)—results that are far beyond the capabilities of any other OWL 2 reasoner known to us. Our system also outperforms HermiT on the realistic Fly ontology by at least an order of magnitude. We are currently working on an extension to support query answering over arbitrary OWL 2 ontologies (and not just Horn ontologies), as well as on several promising optimisations.

References

1. Acciarri, A., Calvanese, D., Giacomo, G.D., Lembo, D., Lenzerini, M., Palmieri, M., Rosati, R.: Quonto: Querying ontologies. In: AAAI. pp. 1670–1671 (2005)
2. Armas Romero, A., Cuenca Grau, B., Horrocks, I.: MORe: Modular combination of owl reasoners for ontology classification. In: ISWC. pp. 1–16 (2012)
3. Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: OWLim: A family of scalable semantic repositories. Semantic Web J. 2(1), 33–42 (2011)
4. Bry, F., Eisinger, N., Eiter, T., Furche, T., Gottlob, G., Ley, C., Linse, B., Pichler, R., Wei, F.: Foundations of rule-based query answering. In: RR. pp. 1–153 (2007)

5. Cali, A., Gottlob, G., Lukasiewicz, T., Marnette, B., Pieris, A.: Datalog+/-: A family of logical knowledge representation and query languages for new applications. In: LICS (2010)
6. Cuenca Grau, B., Horrocks, I., Krötzsch, M., Kupke, C., Magka, D., Motik, B., Wang, Z.: Acyclicity conditions and their application to query answering in description logics. In: Proc. of the 13th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2012) (2012), `download/2012/CHKKMMW12a.pdf`
7. Dolby, J., Fokoue, A., Kalyanpur, A., Ma, L., Schonberg, E., Srinivas, K., Sun, X.: Scalable grounded conjunctive query evaluation over large and expressive knowledge bases. In: The Semantic Web-ISWC 2008, pp. 403–418. Springer (2008)
8. Eiter, T., Ortiz, M., Šimkus, M., Tran, T.K., Xiao, G.: Query rewriting for horn-shiq plus rules. In: AAAI (2012)
9. Haarslev, V., Möller, R.: RACER system description. J. of Automated Reasoning (JAR) pp. 701–705 (2001)
10. Haarslev, V., Hidde, K., Möller, R., Wessel, M.: The RacerPro knowledge representation and reasoning system. Semantic Web 3(3), 267–277 (2012)
11. Hustadt, U., Motik, B., Sattler, U.: Reasoning in Description Logics by a Reduction to Disjunctive Datalog. Journal of Automated Reasoning 39(3), 351–384 (2007)
12. Imprialou, M., Stoilos, G., Grau, B.: Benchmarking ontology-based query rewriting systems. In: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, AAAI (2012)
13. Kaplunova, A., Möller, R., Wandelt, S., Wessel, M.: Towards scalable instance retrieval over ontologies. Knowledge Science, Engineering and Management (2010)
14. Kollia, I., Glimm, B.: Cost based query ordering over OWL ontologies. In: ISWC. pp. 231–246 (2012)
15. Kollia, I., Glimm, B., Horrocks, I.: SPARQL query answering over OWL ontologies. In: ESWC. pp. 382–396 (2011)
16. Motik, B., Shearer, R., Horrocks, I.: Hypertableau reasoning for description logics. J. of Artificial Intelligence Research (JAIR) 36(1), 165–228 (2009)
17. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language Profiles (2nd Edition) (2012), W3C Recommendation.
18. Pan, J., Thomas, E., Zhao, Y.: Completeness guaranteed approximations for OWL-DL query answering. DL 477 (2009)
19. Pérez-Urbina, H., Horrocks, I., Motik, B.: Efficient query answering for OWL 2. In: ISWC. pp. 489–504 (2009)
20. Rodriguez-Muro, M., Calvanese, D.: High performance query answering over DL-Lite ontologies. In: KR (2012)
21. Rosati, R.: Prexto: Query rewriting under extensional constraints in DL-Lite. In: ESWC. pp. 360–374 (2012)
22. Sagonas, K., Swift, T.: An abstract machine for tabled execution of fixed-order stratified logic programs. ACM Transactions on Programming Languages and Systems (TOPLAS) 20(3), 586–634 (1998)
23. Sirin, E., Parsia, B., Cuenca Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. Web Semantics (JWS) 5(2), 51–53 (2007)
24. Tamaki, H., Sato, T.: OLD resolution with tabulation. In: ICLP. pp. 84–98 (1986)
25. Tserendorj, T., Rudolph, S., Krötzsch, M., Hitzler, P.: Approximate OWL-reasoning with screech. In: RR. pp. 165–180 (2008)
26. Wu, Z., Eadon, G., Das, S., Chong, E.I., Kolovski, V., Annamalai, M., Srinivasan, J.: Implementing an inference engine for RDFS/OWL constructs and user-defined rules in Oracle. In: ICDE. pp. 1239–1248 (2008)

27. Zhou, Y., Cuenca Grau, B., Horrocks, I., Wu, Z., Banerjee, J.: Making the Most of your Triple Store: Query Answering in OWL 2 Using an RL Reasoner. In: WWW (2013)

A Proof for Theorem 1

In this section, we proof the correctness of Theorem 1. Because the 'if' direction is trivial for monotonicity, we next prove the 'only if' direction.

Let $\mathcal{O}$ be an OWL 2 ontology, $\mathcal{D}$ be a dataset, Q be a conjunctive query and G be the gap between lower and upper bounds of Q. According to the Definition 2, $\mathcal{O}_f \cup \mathcal{D}_f$ is the union of the $(Q, \{\vec{a}\})$-relevant fragment, denoted as $\mathcal{O}_{\vec{a}} \cup \mathcal{D}_{\vec{a}}$,for each $\vec{a} \in G$. Due to the monotonicity, it suffices to prove that for each $\vec{a} \in G$,

$$\text{if } \mathcal{O} \cup \mathcal{D} \models Q(\vec{a}), \text{ then } \mathcal{O}_{\vec{a}} \cup \mathcal{D}_{\vec{a}} \models Q(\vec{a}).$$

A.1 Resolution Notations

Our definition of SLD-resolution rule is slightly deviated from the standard SLD-resolution by restricting ourselves on the most general unifiers. The next lamma guarantees that in terms of the rule is also complete for horn clauses because the completeness of standard SLD-resolution.

Lemma 1. *Let Λ be a set of datalog rules with functions, and let Q be a Boolean conjunctive query, if there is a standard SLD-refutation*

$$(H_0 = G_Q) \overset{r_1,\mu_1}{\rightsquigarrow} H_1 \overset{r_2,\mu_2}{\rightsquigarrow} \dots \overset{r_n,\mu_n}{\rightsquigarrow} (H_n = \square)$$

of G_Q in Λ with $r_k \in \Lambda$ and θ_k a substitution for each k, then there is an SLD-proof using the same sequence of rules $r_1, \dots, r_n$ of the form

$$(P_0 = G_Q) \overset{r_1,\theta_1}{\rightsquigarrow} P_1 \overset{r_2,\theta_2}{\rightsquigarrow} \dots \overset{r_n,\theta_n}{\rightsquigarrow} (P_n = \square)$$

Definition 3. *Let Γ be a set of datalog rules with functions, and let G_0 be a goal of the form $A_1 \wedge \dots \wedge A_m$. A* SLD-tree *of G_0 in Γ $T(G_0, \Gamma)$ is a labeled tree satisfying the following conditions:*

- *the root of $T(G_0, \Gamma)$ is labeled with $A_1 \wedge \dots \wedge A_m$;*
- *if G_{i+1} can be obtained by a single SLD-resolution $G_i \overset{r,\theta}{\rightsquigarrow} G_{i+1}$, then the node labeled by G_i has a child labeled by G_{i+1} by an edge labeled by r.*

If a branch of $T(G_0, \Gamma)$ finishes with $\perp$, the branch is an SLD-proof of G_0 in Γ. The refutation rules *of G_0 in Γ is $R(G_0, \Gamma)$ defined as all the edge labels in any SLD-refutation in $T(G_0, \Gamma)$.*

The SLD-tree of G_0 in Γ should contain all the SLD-proof of G_0 in Γ.

A.2 Over-approximation Notations

We next define some notations to formalize the over-approximation from datalog$^{\pm}$ rules to datalog rules.

Definition 4. *Let Λ be a first order logic signature, and let Λ' be a disjoint signature containing only constants such that if a function $f \in \Lambda$, then there exists a corresponding constant $c_f \in \Lambda'$. Let t be a term in signature Λ, $\chi^{\Lambda}(t)$ is defined as follows.*

- *if t is a constant or variable, then $\chi^{\Lambda}(t) = t$;*
- *if t is in the form of $f(t_1, \ldots, t_l)$ then $\chi^{\Lambda}(t) = c_f$.*

The Λ is omitted for brevity if the signature is clear from the context.

Let φ be a first order formula in signature Λ, $\chi(\varphi)$ is defined as the resulting formula by replacing each term t in φ by $\chi(t)$. Let θ be a substitution of the form $\{t_1/x_1, \ldots, t_n/x_n\}$ from signature Λ, $\chi(\theta)$ is $\{\chi(t_1)/x_1, \ldots, \chi(t_n)/x_n\}$.

Property 1. Let Λ be a first order logic signature, $\chi(t\theta) = \chi(t)\chi(\theta)$ for any term t and any substitution θ from Λ.

Definition 5. *A* datalog$^{\pm}$ rule r *is a function-free first order sentence of the following form with the commas standing for conjunctions and the outside universal quantifier $\forall \vec{x}$ omitted for brevity*

$$\exists \vec{y}\,(C_1 \wedge \ldots \wedge C_q) \leftarrow B_1 \wedge \ldots \wedge B_p \tag{2}$$

where $B_1, \ldots, B_p$ are non-$\bot$ atoms with free variables in $\vec{x}$, $C_1, \ldots, C_q$ are non-$\bot$ atoms with free variables in $\vec{x} \cup \vec{y}$.

Let Σ be a set of datalog$^{\pm}$ rules. For each rule $r \in \Sigma$ of the form (2) and each variable $y_i \in \vec{y}$, let f_r^i be a function symbol unique for r and y_i. Then $\mathsf{sk}(r)$, the skolemization of r, is the rule

$$C_1\theta \wedge \ldots \wedge C_q\theta \leftarrow B_1 \wedge \ldots \wedge B_p$$

where θ is a substitution that maps each variable $y_i \in \vec{y}$ to $f_r^i(\vec{x})$. And $\mathsf{sk}(\Sigma) = \{\mathsf{sk}(r) \mid r \in \Sigma\}$.

Let Σ be a set of datalog$^{\pm}$ rules, Λ be the signature of $\mathsf{sk}(\Sigma)$ and Λ' the corresponding signature as defined in Definition 4. Then $\Xi(r)$, the over-approximation of r defined in our previous paper [27], can be also defined as $\chi(\mathsf{sk}(r))$. And recall that $\Xi(\Sigma) = \{\Xi(r) \mid r \in \Sigma\}$.

It is easy to verify that $\Xi(\cdot)$ is a injection function from datalog$^{\pm}$ rules Σ to datalog rules, and thus, its inverse $\Xi^{-}(\cdot)$ on $\Xi(\Sigma)$ is well-defined. For a set of datalog rule $\Gamma \subseteq \Xi(\Sigma)$, $\Xi^{-}(\Gamma) = \{\Xi^{-}(s) \mid s \in \Gamma\}$.

Let Σ be a set of datalog$^{\pm}$ rules. According to Definition 5, $\mathsf{sk}(\Sigma)$ is a set of datalog rules with functions and $\Xi(\Sigma)$ is a set of datalog rules. For any rule $r \in \Sigma$ of the form (2),

$$\begin{aligned}
\mathsf{sk}(r): \quad & C_1\theta, \ldots, C_q\theta \leftarrow B_1, \ldots, B_p \\
& = \exists \vec{y}\,(C_1\theta, \ldots, C_q\theta) \leftarrow B_1, \ldots, B_p \\
& = r\theta \\
\Xi(r): \quad & \chi(\mathsf{sk}(r)) = \chi(r\theta) = r\chi(\theta)
\end{aligned}$$

A.3 Proof

Now we are ready to formally prove the theorem. Let $\vec{a}$ be an arbitrary tuple in G. The fragment $\mathcal{O}_{\vec{a}} \cup \mathcal{D}_{\vec{a}}$ is the set of OWL axioms that are equivalent to $\Xi^-(R(Q(\vec{a}), \Xi(\Sigma_{\mathcal{O}})))$, where $\Sigma_{\mathcal{O}}$ is the datalog$^{\pm}$ program that are equivalent to $\mathcal{O}$ in the signature of $\mathcal{O}$. Then,

$$\mathcal{O} \cup \mathcal{D} \models Q(\vec{a}) \text{ iff } \Sigma_{\mathcal{O}} \models Q(\vec{a})$$
$$\mathcal{O}_{\vec{a}} \cup \mathcal{O}_{\vec{a}} \models Q(\vec{a}) \text{ iff } \Xi^-(R(Q(\vec{a}), \Xi(\Sigma_{\mathcal{O}}))) \models Q(\vec{a})$$

In order to prove

$$\text{if } \mathcal{O} \cup \mathcal{D} \models Q(\vec{a}) \text{ then } \mathcal{O}_{\vec{a}} \cup \mathcal{D}_{\vec{a}} \models Q(\vec{a}),$$

it suffices to prove the following theorem.

Theorem 2. *Let Σ be a set of datalog$^{\pm}$ rules, let Q be the Boolean conjunctive query of the form $P(\vec{a})$ for a conjunctive query P and a tuple $\vec{a}$, then the following holds*

$$\textit{if } \Sigma \models Q \textit{ then } \Xi^-(R(Q, \Xi(\Sigma))) \models Q$$

Proof. Since $\Sigma \models Q$, we have $\mathsf{sk}(\Sigma) \models Q$, and further there exists an SLD-proof of Q in $\mathsf{sk}(\Sigma)$. Assume it to be the derivation (3) where $r_k \in \Sigma$ for each k.

$$(G_0 = Q) \overset{\mathsf{sk}(r_1),\mu_1}{\rightsquigarrow} G_1 \rightsquigarrow \ldots \rightsquigarrow (G_n = \square) \tag{3}$$
$$(H_0 = Q) \overset{\Xi(r_1),\chi(\mu_1)}{\rightsquigarrow} H_1 \rightsquigarrow \ldots \rightsquigarrow (H_n = \square) \tag{4}$$
$$(P_0 = Q) \overset{\Xi(r_1),\lambda_1}{\rightsquigarrow} P_1 \rightsquigarrow \ldots \rightsquigarrow (P_n = \square) \tag{5}$$

We next prove that there is an SLD-proof of Q in $\Xi(\Sigma)$ of the form (5) that appears in the SLD-derivation tree of Q in $\Xi(\Sigma)$. We divide the following proof into two steps:

Step 1. Given the SLD-proof (3) of G_Q in $\mathsf{sk}(\Sigma)$, there is a standard SLD-refutation (4) of Q in $\Xi(\Sigma)$.
Step 2. Given the standard SLD-refutation (4) of Q in $\Xi(\Sigma)$, there exists an SLD-proof (5) of Q in $\Xi(\Sigma)$.

Step 2 can be proved by Lemma 1. So we only need to prove Step 1.

Proof for Step 1.

We first construct $H_k = \chi(G_k)$ and next show that H_k can be obtained from H_{k-1} by a single SLD-resolution with the substitution $\chi(\mu_k)$ using rule $\Xi(r_k)$ for each k.

Assume that we have the following facts.

$$G_{k-1}: \quad A_1 \wedge A_2 \wedge \ldots \wedge A_m$$
$$r_k \in \Sigma: \quad C_1 \wedge \ldots \wedge C_q \leftarrow B_1 \wedge \ldots \wedge B_p$$

Then,

$$H_{k-1}: \quad \chi(A_1) \wedge \chi(A_2) \wedge \ldots \wedge \chi(A_m)$$

According to Definition 5, there is a substitution θ such that

$$\begin{aligned} \mathsf{sk}(r_k) = r_k\theta: &\quad C_1\theta \wedge \ldots \wedge C_q\theta \leftarrow B_1 \wedge \ldots \wedge B_p \\ \Xi(r_k) = r_k\chi(\theta): &\quad \chi(C_1\theta) \wedge \ldots \wedge \chi(C_q\theta) \leftarrow B_1 \wedge \ldots \wedge B_p \end{aligned}$$

With the substitution μ_k such that $A_1\mu = C_j\theta\mu$, we have

$$G_k: \quad A_2\mu \wedge \ldots \wedge A_p\mu \wedge B_1\mu \wedge \ldots \wedge B_p\mu$$

Since $\chi(A_1)\chi(\mu) = \chi(A_1\mu) = \chi(C_j\theta\mu) = \chi(C_j\theta)\chi(\mu)$, the following goal H_k is obtained with the substitution $\chi(\mu)$.

$$\chi(A_2)\chi(\mu) \wedge \ldots \wedge \chi(A_m)\chi(\mu) \wedge B_1\chi(\mu) \wedge \ldots \wedge B_p\chi(\mu)$$

which is equivalent to $\chi(G_k)$ because of Property 1 and the fact that $\chi(B) = B$ for each function-free atom B. $\diamondsuit$

Therefore, the sequence (4) is a standard SLD-derivation of Q in $\mathsf{sk}(\Sigma)$. Moreover, $H_n = \chi(G_n) = \square$ and hence the SLD-derivation (4) is an SLD-refutation of Q in $\Xi(\Sigma)$.

So $P_0 \overset{\Xi(r_1),\lambda_1}{\rightsquigarrow} P_1 \overset{\Xi(r_2),\lambda_2}{\rightsquigarrow} \ldots \overset{\Xi(r_n),\lambda_n}{\rightsquigarrow} P_n$ is an SLD-proof of Q in $\Xi(\Sigma))$ and hence, $\{\Xi(r_1), \ldots, \Xi(r_n)\} \subseteq R(Q, \Xi(\Sigma))$. Therefore,

$$\{r_1, \ldots, r_n\} \subseteq \Xi^-(R(Q, \Xi(\Sigma))).$$

Because (3) is an SLD-proof of Q in $\mathsf{sk}(\Sigma)$, $\{\mathsf{sk}(r_1), \ldots, \mathsf{sk}(r_n)\} \models Q$ and hence $\{r_1, \ldots, r_n\} \models Q$. Finally, we have $\Xi^-(R(Q, \Xi(\Sigma))) \models Q$. $\square$