Query Answering over $\mathcal{SROIQ}$ Knowledge Bases with SPARQL

Ilianna Kollia[1], Birte Glimm[2], and Ian Horrocks[2]

[1] National Technical University of Athens, Greece
[2] Oxford University Computing Laboratory, UK

Abstract. W3C currently extends the SPARQL query language with so-called entailment regimes, which define how queries are evaluated using logical entailment relations. We describe a sound and complete algorithm for the OWL Direct Semantics entailment regime. Since OWL's Direct Semantics is based on Description Logics (DLs), this results in an expressive query language for DL knowledge bases. The query language differs from the commonly studied conjunctive queries in that it only has distinguished variables. Furthermore, variables can occur within complex concepts and can also bind to concept or role names. We provide a prototypical implementation and propose several novel optimizations strategies. We evaluate the efficiency of the proposed optimizations and find that for ABox queries our system performs comparably to already deployed systems. For complex queries an improvement of up to three orders of magnitude can be observed.

1 Introduction

Query answering is important in the context of the Semantic Web, since it provides a mechanism via which users and applications can interact with ontologies and data. Although SPARQL [12] was standardized in 2008 by the World Wide Web Consortium for querying Semantic Web data, only the simple semantics of RDF is supported by SPARQL 1.0, which does not allow for any reasoning.

There is not yet a standardized query language for OWL knowledge bases (KBs). Several of the widely deployed systems support, however, some query language. Pellet supports SPARQL-DL [13], which is a subset of SPARQL, adapted to work with OWL's Direct Semantics. Similarly, KAON2 supports [9] SPARQL, but restricted to ABox queries. Racer Pro [3] has a proprietary query language, called nRQL [4], which allows for queries that go beyond ABox queries, e.g., one can retrieve sub- or super-concepts of a given concept. TrOWL is another system that supports ABox SPARQL queries, but the reasoning in TrOWL is approximate, i.e., an OWL DL ontology is rewritten into an ontology that uses a less expressive language before reasoning is applied [14]. Furthermore, there are systems such as QuOnto[3] or Requiem,[4] which support profiles of OWL 2, and which support conjunctive queries, e.g., written in SPARQL syntax. Of the systems that support all of OWL 2 DL, only Pellet supports non-distinguished variables as long as they are not used in cycles, which is a measure to ensure decidability.

[3] http://www.dis.uniroma1.it/~quonto/
[4] http://www.comlab.ox.ac.uk/projects/requiem/home.html

The SPARQL W3C working group is currently devising version 1.1 of SPARQL, which also includes several *entailment regimes*. These entailment regimes redefine the semantics of SPARQL queries based on standard semantic web entailment relations such as RDFS or OWL Direct Semantics entailment. This allows for using SPARQL also as a query language over OWL ontologies with query answers also including solutions that are implicit consequences of the queried ontology or knowledge base.

In this paper, we present an implementation and optimization techniques for the SPARQL OWL 2 Direct Semantics entailment regime, which we call SPARQL-OWL for brevity. SPARQL-OWL only allows for distinguished variables, but it poses significant challenges for implementations, e.g., by allowing variables that bind to concepts or roles and which can even occur within complex concepts. Our implementation supports ontologies (knowledge bases) in OWL 2 DL and is based on the HermiT reasoner.[5] Most of the devised optimization techniques are also applicable when using another OWL reasoner. In our algorithm, we extend the techniques used for conjunctive query answering to deal with arbitrary SPARQL-OWL queries and propose a range of novel optimizations in particular for SPARQL-OWL queries that go beyond SPARQL-DL.

Our prototypical system is the first to fully support SPARQL-OWL, and we have performed a preliminary evaluation in order to investigate the feasibility of our algorithm and the effectiveness of the proposed optimizations. This evaluation suggests that, in the case of standard conjunctive queries, our system performs comparably to existing ones. It also shows that a naive implementation of our algorithm behaves badly for some non-standard queries, but that the proposed optimizations can dramatically improve performance, in some cases by as much as three orders of magnitude.

An extended version of this paper is accepted at ESWC'11 [10].

2 Preliminaries

In this section we give a brief introduction to the SPARQL-OWL entailment regime and in the next section we describe an algorithm that finds answers to queries under this regime.

2.1 The Relationship between RDF, SPARQL, and OWL

SPARQL is originally an RDF query language and the WHERE clause of a SPARQL query consists of an RDF graph, where some nodes or edges are replaced by variables. There is, however, a close relationship between OWL and RDF since OWL ontologies can be represented as RDF graphs. Furthermore, OWL's RDF-Based Semantics is a direct extension of the RDF and RDFS semantics. We focus here, however, on OWL's Direct Semantics, which is based on the DL $\mathcal{SROIQ}$ [8] and which is only defined for certain well-formed RDF graphs. Well-formedness guarantees that the RDF graph can be mapped into an OWL 2 DL ontology [11], which can be seen as a $\mathcal{SROIQ}$ KB.

An example of a SPARQL query is

```
SELECT ?i FROM <ontologyIRI> WHERE { ?i rdf:type C }
```

[5] http://www.hermit-reasoner.com

where the triple in the WHERE clause is called a basic graph pattern (BGP) and is written in Turtle [1]. Since the Direct Semantics of OWL is defined in terms of OWL structural objects, such a BGP is mapped into structural objects, which can have variables in place of class, object property, data property, or individual names or literals. For example, the above BGP is mapped to ClassAssertion(C ?i) in functional-style syntax or C(?i) in DL syntax.

OWL DL is a typed language and to map RDF triples into OWL structural objects, one often has to know the type of a term. For example, in order to map the triple p rdfs:subpropertyOf p′ into an OWL structural object, we have to know whether p is an abstract or a concrete role (an object or a data property), in the former case, the mapping results in SubObjectPropertyOf(p p′), whereas in the latter case, we get SubDataPropertyOf(p p′). In DL notation, we get $\mathsf{p} \sqsubseteq \mathsf{p}'$, but p and p′ would either be abstract or concrete roles. In many cases, the typing information from the queried KB can be used to disambiguate the mapping process. For variables that map to concepts or roles, however, typing information is usually required and has to be added to the BGP. For example,

a rdf:type [rdf:type owl:Restricion ; owl:onProperty ?x ; owl:someValuesFrom ?y]

could be mapped to either (1) or (2).

$$\mathsf{ClassAsserion(ObjectSomeVauesFrom(?x\ ?y)\ a)} \tag{1}$$

$$\mathsf{ClassAsserion(DataSomeVauesFrom(?x\ ?y)\ a)} \tag{2}$$

In such a case, a triple such as ?x rdf:type owl:ObjectProperty can be added to disambiguate the mapping process. Although the SPARQL specification uses Turtle, other query syntaxes can also be defined. Pellet accepts, for example, queries where the BGP is written in Manchester Syntax [7].

For further details, we refer interested readers to the W3C specification that defines the mapping between OWL structural objects and RDF graphs [11] and to the SPARQL-OWL entailment regime[6] that defines the extension of this mapping between BGPs and OWL objects with variables.

2.2 SPARQL-OWL Queries

In the following, we directly write BGPs in DL notation extended to allow for variables in place of concept, role and individual names in axioms. For simplicity, we do not consider concrete roles (data properties) here.

Anonymous individuals in the query are treated as variables whose bindings do not appear in the query's result sequence. This is motivated by the way SPARQL handles anonymous individuals (known as blank nodes in RDF terminology). This is in contrast to conjunctive queries where they are treated as existential variables. Furthermore, anonymous individuals in the queried KB are treated as (Skolem) constants and can be returned in a query answer. For brevity, we assume here that neither the query nor the queried KB contains anonymous individuals.

[6] http://www.w3.org/TR/sparql11-entailment/

Definition 1. *Let N_C, N_R, N_I, V_C, V_R, and V_I be countable, infinite, and pairwise disjoint sets of* concept names, role names, individual names, concept variables, role variables, *and* individual variables, *respectively. We call $\mathcal{S} = (N_C, N_R, N_I, V_C, V_R, V_I)$ a* signature. *A SPARQL-OWL* query *w.r.t. $\mathcal{S}$ consists of* axiom templates, *which are $\mathcal{SROIQ}$ axioms where in place of concept names once can use names from $N_C \cup V_C$, in place of role names, one can use names from $N_R \cup V_R$, and in place of individual names, on can use names form $N_I \cup V_I$. A $\mathcal{SROIQ}$* knowledge base *uses only terms from N_C, N_R, and N_I. The restriction of $\mathcal{S}$ to terms that occur in a knowledge base $\mathcal{K}$ (a query q) is denoted as $\mathcal{S}_\mathcal{K}$ ($\mathcal{S}_q$); we write* $\mathsf{V(q)}$ *to denote the set of all variables in q.*

Given a knowledge base $\mathcal{K}$ with $\mathcal{S}_\mathcal{K} = (N_C^\mathcal{K}, N_R^\mathcal{K}, N_I^\mathcal{K}, \emptyset, \emptyset, \emptyset)$ and a query q over $(N_C^\mathcal{K}, N_R^\mathcal{K}, N_I^\mathcal{K}, V_C, V_R, V_I)$, a solution mapping μ for q over $\mathcal{K}$ *is a partial function $\mu\colon V_C \cup V_R \cup V_I \to N_C^\mathcal{K} \cup N_R^\mathcal{K} \cup N_I^\mathcal{K}$ such that $\mathsf{dom}(\mu) = \mathsf{V(q)}$, $\mu(v) \in N_C^\mathcal{K}$ for each $v \in V_C \cap \mathsf{dom}(\mu)$, $\mu(v) \in N_R^\mathcal{K}$ for each $v \in V_R \cap \mathsf{dom}(\mu)$, and $\mu(v) \in N_I^\mathcal{K}$ for each $v \in V_I \cap \mathsf{dom}(\mu)$, where $\mathsf{dom}(\mu)$ denotes the domain of μ; we write $\mu(q)$ to denote the result of replacing each variable v in q with $\mu(v)$.*

The evaluation of q over $\mathcal{K}$ yields a set of solution mappings μ with

$$\{\, \mu \mid \mathcal{K} \cup \mu(q) \text{ is a } \mathcal{SROIQ} \text{ knowledge base and } \mathcal{K} \models \mu(q)\}$$

More complex WHERE clauses, which use operators such as UNION for alternative selection criteria or $\mathsf{OPTIONAL}$ to query for optional bindings [12, 5], can be evaluated simply by combining solution mappings obtained by the BGP/query evaluation. Therefore, we focus here on BGP evaluation only.

In the remainder, we use $\mathcal{K}$ to denote the $\mathcal{SROIQ}$ KB obtained from a queried RDF graph, and q for the query obtained from mapping a BGP into axiom templates. We further assume that the signature of $\mathcal{K}$ is $\mathcal{S}_\mathcal{K} = (N_C^\mathcal{K}, N_R^\mathcal{K}, N_I^\mathcal{K}, \emptyset, \emptyset, \emptyset)$ and a query uses symbols from $(N_C^\mathcal{K}, N_R^\mathcal{K}, N_I^\mathcal{K}, V_C, V_R, V_I)$.

3 Evaluation of SPARQL-OWL Queries

A straightforward algorithm to realize the entailment regime simply tests, for each possible solution mapping μ, whether $\mathcal{K} \models \mu(q)$. Since only terms that are used in $\mathcal{K}$ can occur in the range of solution mappings, there are finitely many mappings to test. In the worst case, however, the number of mappings that have to be tested is still exponential in the number of variables in the query. Such an algorithm is sound and complete if the reasoner used to decide entailment is sound and complete since we check all mappings for variables that can constitute actual solution mappings.

3.1 General Query Evaluation Algorithm

Optimizations cannot easily be integrated in the above sketched algorithm since it uses the reasoner to check for the entailment of the instantiated query as a whole and, hence, does not take advantage of relations that may exist between axiom templates. For a more optimized evaluation, we evaluate the query axiom template by axiom template. Initially, our solution set contains only the identity mapping, which does not map any variable to a value. We then pick our first axiom template, extend the identity mapping to cover the variables of the chosen axiom template and use the reasoner to check which of the mappings instantiate the axiom template into an entailed axiom. We then pick the next axiom template and again extend the mappings from the previous round to cover all variables and check which of those mappings lead to an entailed axiom. Thus, axiom templates which are very selective and are only satisfied by very few solutions reduce the number of intermediate solutions. Choosing a good execution order, therefore, can significantly affect the performance.

For example, let $q = \{\mathsf{C}(?\mathsf{x}), \mathsf{r}(?\mathsf{x}\ ?\mathsf{y})\}$ with $\mathsf{r} \in N_R, \mathsf{C} \in N_C, ?\mathsf{x}, ?\mathsf{y} \in V_I$. The query belongs to the class of conjunctive queries. We assume that the queried KB contains 100 individuals, only 1 of which belongs to the concept C. This C instance has 1 r-successor, while we have overall 200 pairs of individuals related with the role r. If we first evaluate C(?x), we test 100 mappings (since ?x is an individual variable), of which only 1 mapping satisfies the axiom template. We then evaluate r(?x ?y) by extending the mapping with all 100 possible mappings for ?y. Again only 1 mapping yields a solution. For the reverse axiom template order, the first axiom template requires the test of $100 * 100$ mappings. Out of those, 200 remain to be checked for the second axiom template and we perform $10,200$ tests instead of just 200.

The importance of the execution order is well known in relational databases and cost based optimization techniques are used to find good execution orders. Ordering strategies as implemented in databases or triple stores are, however, not directly applicable in our setting. In the presence of expressive schema level axioms, we cannot rely on counting the number of occurrences of triples. We also cannot, in general, precompute all relevant inferences to base our statistics on materialized inferences. Furthermore, we should not only aim at decreasing the number of intermediate results, but also take into account the cost of checking or computing the solutions. This cost can be very significant with OWL reasoning.

For several kinds of axiom templates we can, instead of checking entailment, directly retrieve the solutions from the reasoner. For example, for C(?x), reasoners typically have a method to retrieve concept instances. Although this might internally trigger several tests, most methods of reasoners are highly optimized and avoid as many tests as possible. Furthermore, reasoners typically cache several results such as the computed concept hierarchy and retrieving sub-concepts can then be realized with a cache lookup. Thus, the actual execution cost might vary significantly. Notably, we do not have a straight correlation between the number of results for an axiom template and the actual cost of retrieving the solutions as is typically the case in triple stores or databases. This requires cost models that take into account the cost of the specific reasoning operations (depending on the state of the reasoner) as well as the number of results.

As motivated above, we distinguish between *simple* and *complex* axiom templates, where simple axiom templates are those that correspond to dedicated reasoning tasks. Complex axiom templates are, in contrast, evaluated by iterating over the compatible mappings and by checking entailment for each instantiated axiom template. An example of a complex axiom template is $(\exists \mathsf{r}.?\mathsf{x})(?\mathsf{y})$.

Algorithm 1 shows how we evaluate queries. We first explain the general outline of the algorithm and leave the details of the used submethods for the following section. We first simplify axiom templates where possible (rewrite, line 1). Next, the method

Algorithm 1 Query Evaluation Procedure

```
Input: $\mathcal{K}$: the queried knowledge base, which is a $\mathcal{SROIQ}$ knowledge base
       $q$: a $\mathcal{SROIQ}$ query
Output: a set of solutions for evaluating $q$ over $\mathcal{K}$
 1: $\mathsf{Axt} := \mathsf{rewrite}(\mathsf{K_q})$ {create a list Axt of simplified axiom templates from $q$}
 2: $\mathsf{Axt}^1, \ldots, \mathsf{Axt}^m := \mathsf{connectedComponents}(\mathsf{Axt})$
 3: for j=1, ..., m do
 4:   $R_j := \{\mu_0 \mid \mathsf{dom}(\mu_0) = \emptyset\}$
 5:   $\mathsf{axt}_1, \ldots, \mathsf{axt}_n := \mathsf{reorder}(\mathsf{Axt}^j)$
 6:   for $i = 1, \ldots, n$ do
 7:     $R_{new} := \emptyset$
 8:     for $\mu \in R_j$ do
 9:       if $\mathsf{isSimple}(\mathsf{axt_i})$ and $\mathsf{V}(\mathsf{axt_i}) \setminus \mathsf{dom}(\mu) \neq \emptyset$ then
10:         $R_{new} := R_{new} \cup \{(\mu \cup \mu') \mid \mu' \in \mathsf{callReasoner}(\mu(\mathsf{axt_i}))\}$
11:       else
12:         $B := \{\mu' \mid \mu'$ extends $\mu, \mu'$ is a solution mapping for $\mathsf{axt_i}$ and $\mathcal{K}\}$
13:         $B := \mathsf{prune}(B, \mathsf{axt_i}, \mathcal{K})$
14:         while $B \neq \emptyset$ do
15:           $\mu' := \mathsf{removeNext}(B)$
16:           if $\mathcal{K} \models \mu'(\mathsf{axt_i})$ then $R_{new} := R_{new} \cup \{\mu'\}$
17:           else $B := \mathsf{prune}(B, \mathsf{axt_i}, \mu')$
18:         end while
19:       end if
20:     end for
21:     $R_j := R_{new}$
22:   end for
23: end for
24: $R := \{\mu_1 \cup \ldots \cup \mu_m \mid \mu_j \in R_j, 1 \leq j \leq m\}$
25: return $R$
```

connectedComponents (line 2) partitions the axiom templates into sets of connected components, i.e., within a component the templates share common variables, whereas between components there are no shared variables. Unconnected components unnecessarily increase the amount of intermediate results and, instead, we can simply combine the results for the components in the end (line 24). For each component, we proceed as described below: we first determine an order (method reorder in line 5). For a simple axiom template, which contains so far unbound variables, we then call a specialized reasoner method to retrieve entailed results (callReasoner in line 10). Otherwise, we check which compatible solutions yield an entailed axiom (lines 11 to 19). The method prune (lines 13 and 17) excludes mappings that cannot lead to entailed axioms.

3.2 Optimized Query Evaluation

Axiom Template Reordering We now explain how we order the axiom templates in the method reorder (line 5). Since complex axiom templates can only be evaluated with costly entailment checks, our aim is to reduce the number of bindings before we check the complex templates. The simple axiom templates are ordered by their cost,

Table 1. Axiom templates and their equivalent simpler ones, where $C_{(i)}$ are complex concepts (possibly containing variables), a is an individual or variable

$$\mathsf{C_1 \sqcap \ldots \sqcap C_n(a) \equiv \{C_i(a) \mid 1 \leq i \leq n\}}$$
$$\mathsf{C \sqsubseteq C_1 \sqcap \ldots \sqcap C_n \equiv \{C \sqsubseteq C_i \mid 1 \leq i \leq n\}}$$
$$\mathsf{C_1 \sqcup \ldots \sqcup C_n \sqsubseteq C \equiv \{C_i \sqsubseteq C \mid 1 \leq i \leq n\}}$$

which is computed as the weighted sum of the estimated number of required consistency checks and the estimated result size. These estimates are based on statistics provided by the reasoner and this is the only part where our algorithm depends on the specific reasoner that is used. In case the reasoner cannot give estimates, one can still work with statistics computed from explicitly stated information. We do this for some simple templates, e.g., queries for domains and ranges of properties, for which the reasoner does not provide result size estimations. Since the result sizes for complex templates are difficult to estimate using either the reasoner or the explicitly stated information in $\mathcal{K}$, we order complex templates based only on the number of bindings that have to be tested. It is obvious that the reordering of axiom templates does not affect soundness and completeness of Algorithm 1.

Axiom Template Rewriting Some costly to evaluate axiom templates can be rewritten into axiom templates that can be evaluated more efficiently and yield an equivalent result. Such axiom templates are shown on the left-hand side of Table 1 and their equivalent simplified form is shown on the right-hand side. To understand the intuition behind such transformation, we consider a query with only the axiom template: $\mathsf{?x \sqsubseteq \exists r.?y \sqcap C}$. Its evaluation requires a quadratic number of consistency checks in the number of concepts (since $\mathsf{?x}$ and $\mathsf{?y}$ are concept variables). The rewriting yields: $\mathsf{?x \sqsubseteq C}$ and $\mathsf{?x \sqsubseteq \exists r.?y}$. The first axiom template is now evaluated with a cheap cache lookup (assuming that the concept hierarchy has been precomputed). For the second one, we only have to check the usually few resulting bindings for $\mathsf{?x}$ combined with all other concept names for $\mathsf{?y}$. We apply the rewriting in the method rewrite in line 1 of our algorithm. Soundness and completeness is preserved since instantiated rewritten templates are semantically equivalent to the corresponding instantiated complex ones.

Concept and Role Hierarchy Exploitation The number of consistency checks required to evaluate a query can be further reduced by taking the concept and role hierarchies into account. Once the concepts and roles are classified (this can ideally be done before a system accepts queries), the hierarchies are stored in the reasoner's internal structures. We further use the hierarchies to prune the search space of solutions in the evaluation of certain axiom templates. We illustrate the intuition with an example: $\mathsf{Infection \sqsubseteq \exists hasCausalLinkTo.?x}$ If C is not a solution and $\mathsf{B \sqsubseteq C}$ holds, then B is also not a solution. Thus, when searching for solutions for $\mathsf{?x}$, the method removeNext (line 15) chooses the next binding to test by traversing the concept hierarchy topdown. When we find a non-solution C, the subtree rooted in C of the concept hierarchy can safely be pruned, which we do in the method prune in line 17. Queries over knowledge bases with a large number of concepts and a deep concept hierarchy can, therefore, gain the maximum advantage from this optimization. We employ similar optimizations using the role hierarchies. It is obvious that we only prune mappings that cannot constitute actual solution and instance mappings, hence, soundness and completeness of Algorithm 1 is preserved.

Exploiting the Domain and Range Restrictions The implicit domains and ranges of the roles in $\mathcal{K}$ (in case the reasoner precomputes and stores them) and/or the explicit ones can be exploited to reduce the number of entailment checks that need to be performed in order to evaluate a query.

Let us assume that $\mathcal{K}$ contains $\top \sqsubseteq \forall \mathsf{takesCourse}.\mathsf{Course}$, expressing a range restriction, and q contains $\mathsf{GraduateStudent} \sqsubseteq \exists \mathsf{takesCourse}.?\mathsf{x}$. In case at least one solution mapping exists for $?\mathsf{x}$, the concept Course and its super-concepts can immediately be considered solution mappings for $?\mathsf{x}$. Moreover, if the reasoner precomputes the disjoint concepts, this information can be used to prune the possible concepts for $?\mathsf{x}$ that are disjoint from the concept Course. This is done in the method prune (line 13), which again preserves soundness and completeness.

4 System Evaluation

Since SPARQL's entailment regimes only change the evaluation of BGPs, standard SPARQL algebra processors can be used to combine the intermediate results, e.g., in unions or joins. Furthermore, standard OWL reasoners such as HermiT, Pellet, or FaCT++ can be used to perform the required reasoning tasks.

4.1 The System Architecture

In our system, the queried KB is loaded into an OWL reasoner and the reasoner performs initial tasks such as concept classification before the system accepts queries. We use the ARQ library[7] of the Jena Semantic Web Toolkit for parsing the SPARQL queries and for the SPARQL algebra operations apart from the BGP evaluation. The BGPs are mapped to queries (as in Def. 1) and represented in a custom extension of the OWL API [6]. The query is then passed to a query optimizer, which applies the axiom template rewriting and then searches for a good query execution plan based on statistics provided by the reasoner. We use the HermiT reasoner for OWL reasoning, but only the module that generates statistics and provides cost estimations is HermiT specific.

4.2 Experimental Results

We tested our system with the Lehigh University Benchmark (LUBM) [2] and a range of custom queries that test complex axiom template evaluation over the more expressive GALEN ontology. All experiments were performed on a Windows Vista machine with a double core 2.2 GHz Intel x86 32 bit processor and Java 1.6 allowing 1GB of Java heap space. We measure the time for one-off tasks such as classification separately since such tasks are usually performed before the system accepts queries. Whether more

[7] http://jena.sourceforge.net/ARQ/

Table 2. Query answering times in milliseconds for LUBM(1,0) and in seconds for the queries of Table 3 with and without optimizations

LUBM(1, 0)	
Query	Time
1	20
2	46
3	19
4	19
5	32
6	58
7	42
8	353
9	4,475
10	23
11	19
12	28
13	16
14	45

GALEN queries from Table 3				
Query	Reordering	Hierarchy Exploitation	Rewriting	Time
1				2.1
1		x		0.1
2				780.6
2		x		4.4
3				>30 min
3		x		119.6
3			x	204.7
3		x	x	4.9
4	x		x	>30 min
4	x	x		361.9
4		x	x	>30 min
4	x	x	x	68.2
5	x			>30 min
5		x		>30 min
5	x	x		5.6

costly operations such as the realization of the ABox, which computes the types for all individuals, are done in the beginning, depends on the setting and the reasoner. Since realization is relatively quick in HermiT for LUBM (GALEN has no individuals), we also performed this task upfront. The given results are averages from executing each query three times. The ontologies and all code required to perform the experiments are available online.[8]

We first evaluate the 14 LUBM queries. These queries are simple ones and have variables only in place of individuals and literals. The LUBM ontology contains 43 concepts, 25 abstract roles, and 7 concrete roles. We tested the queries on LUBM(1,0), which contains data for one university starting from index 0, and which contains 16,283 individuals and 8,839 literals. The ontology took 3.8 s to load and 22.7 s for classification and realization. Table 2 shows the execution time for each of the queries. The reordering optimization has the biggest impact on queries 2, 7, 8, and 9. These queries require much more time or are not answered at all within the time limit of 30 min without this optimization (758.9 s, 14.7 s, >30 min, >30 min, respectively).

Conjunctive queries are supported by a range of OWL reasoners. SPARQL-OWL allows, however, the creation of very powerful queries, which are not currently supported by any other system. In the absence of suitable standard benchmarks, we created a custom set of queries as shown in Table 3. Since the complex queries are mostly based on complex schema queries, we switched from the very simple LUBM ontology to the GALEN ontology. GALEN consists of 2,748 concepts and 413 abstract roles. The ontology took 1.6 s to load and 4.8 s to classify (concepts and roles). The execution time for these queries is shown on the right-hand side of Table 2. For each query, we tested

[8] http://www.hermit-reasoner.com/2010/sparqlowl/sparqlowl.zip

Table 3. Sample complex queries for the GALEN ontology

1	$\mathsf{Infection} \sqsubseteq \exists \mathsf{hasCausalLinkTo}.?x$
2	$\mathsf{Infection} \sqsubseteq \exists ?y.?x$
3	$?x \sqsubseteq \mathsf{Infection} \sqcap \exists \mathsf{hasCausalAgent}.?y$
4	$\mathsf{NAMEDLigament} \sqsubseteq \mathsf{NAMEDInternalBodyPart} \sqcap ?x$
	$?x \sqsubseteq \exists \mathsf{hasShapeAnalagousTo}?y \ \sqcap \ \exists ?z.\mathsf{linear}$
5	$?x \sqsubseteq \mathsf{NonNormalCondition}$
	$?z \sqsubseteq \mathsf{ModifierAttribute}$
	$\mathsf{Bacterium} \sqsubseteq \exists ?z.?w$
	$?y \sqsubseteq \mathsf{StatusAttribute}$
	$?w \sqsubseteq \mathsf{AbstractStatus}$
	$?x \sqsubseteq \exists ?y.\mathsf{Status}$

the execution once without optimizations and once for each combination of applicable optimizations from Section 3.

As expected, an increase in the number of variables within an axiom template leads to a significant increase in the query execution time because the number of mappings that have to be checked grows exponentially in the number of variables. This can, in particular, be observed from the difference in execution time between Query 1 and 2. From Queries 1, 2, and 3 it is evident that the use of the hierarchy exploitation optimization leads to a decrease in execution time of up to two orders of magnitude and, in combination with the query rewriting optimization, we can get an improvement of up to three orders of magnitude as seen in Query 3. Query 4 can only be completed in the given time limit if at least reordering and hierarchy exploitation is enabled. Rewriting splits the first axiom template into the following two simple axiom templates, which are evaluated much more efficiently:

$$\mathsf{NAMEDLigament} \sqsubseteq \mathsf{NAMEDInternalBodyPart} \quad \text{and} \quad \mathsf{NAMEDLigament} \sqsubseteq ?x$$

After the rewriting, the reordering optimization has an even more pronounced effect since both rewritten axiom templates can be evaluated with a simple cache lookup. Without reordering, the complex axiom template could be executed before the simple ones, which leads to the inability to answer the query within the time limit of 30 min. Without a good ordering, Query 5 can also not be answered, but the additional use of the class and property hierarchy further improves the execution time by three orders of magnitude.

Although our optimizations can significantly improve the query execution time, the required time can still be quite high. In practice, it is, therefore, advisable to add as many restrictive axiom templates for query variables as possible. For example, the addition of $?y \sqsubseteq \mathsf{Shape}$ to Query 4 reduces the runtime from 68.2 s to 1.6 s.

5 Discussion

We have presented a sound and complete query answering algorithm and novel optimizations for SPARQL's OWL Direct Semantics entailment regime. Our prototypical query answering system combines existing tools such as ARQ, the OWL API, and the HermiT OWL reasoner. Apart from the query reordering optimization—which uses (reasoner dependent) statistics provided by HermiT—the system is independent of the reasoner used, and could employ any reasoner that supports the OWL API.

We evaluated the algorithm and the proposed optimizations on the LUBM benchmark and on a custom benchmark that contains queries that make use of the very expressive features of the entailment regime. We showed that the optimizations can improve query execution time by up to three orders of magnitude.

Acknowledgements This work was supported by EPSRC in the project *HermiT: Reasoning with Large Ontologies*.

References

1. Beckett, D., Berners-Lee, T.: Turtle – Terse RDF Triple Language. W3C Team Submission (14 January 2008), available at http://www.w3.org/TeamSubmission/turtle/
2. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Semantics 3(2-3), 158–182 (2005)
3. Haarslev, V., Möller, R.: Racer system description. In: Gor, R., Leitsch, A., Nipkow, T. (eds.) Proc. 1st Int. Joint Conf. on Automated Reasoning (IJCAR'01). LNCS, vol. 2083, pp. 701–705. Springer (2001)
4. Haarslev, V., Möller, R., Wessel, M.: Querying the semantic web with Racer + nRQL. In: Proc. KI-2004 International Workshop on Applications of Description Logics (2004)
5. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC (2009)
6. Horridge, M., Bechhofer, S.: The OWL API: A Java API for working with OWL 2 ontologies. In: Patel-Schneider, P.F., Hoekstra, R. (eds.) Proc. OWLED 2009 Workshop on OWL: Experiences and Directions. CEUR Workshop Proceedings, vol. 529. CEUR-WS.org (2009)
7. Horridge, M., Patel-Schneider, P.F. (eds.): OWL 2 Web Ontology Language: Manchester Syntax. W3C Working Group Note (27 October 2009), available at http://www.w3.org/TR/owl2-manchester-syntax/
8. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible $\mathcal{SROIQ}$. In: Doherty, P., Mylopoulos, J., Welty, C.A. (eds.) Proc. 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'06). pp. 57–67. AAAI Press (2006)
9. Hustadt, U., Motik, B., Sattler, U.: Reducing $\mathcal{SHIQ}^-$ description logic to disjunctive datalog programs. In: Proc. 9th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'04). pp. 152–162. AAAI Press (2004)
10. Kollia, I., Glimm, B., Horrocks, I.: SPARQL Query Answering over OWL Ontologies. In: Proc. 8th Extended Semantic Web Conf. (ESWC'11) (2011), to appear
11. Patel-Schneider, P.F., Motik, B. (eds.): OWL 2 Web Ontology Language: Mapping to RDF Graphs. W3C Recommendation (27 October 2009), available at http://www.w3.org/TR/owl2-mapping-to-rdf/
12. Prud'hommeaux, E., Seaborne, A. (eds.): SPARQL Query Language for RDF. W3C Recommendation (15 January 2008), available at http://www.w3.org/TR/rdf-sparql-query/
13. Sirin, E., Parsia, B.: SPARQL-DL: SPARQL query for OWL-DL. In: Golbreich, C., Kalyanpur, A., Parsia, B. (eds.) Proc. OWLED 2007 Workshop on OWL: Experiences and Directions. CEUR Workshop Proceedings, vol. 258. CEUR-WS.org (2007)
14. Thomas, E., Pan, J.Z., Ren, Y.: TrOWL: Tractable OWL 2 reasoning infrastructure. In: Proceedings of the Extended Semantic Web Conference (ESWC'10) (2010)