Answering Queries over OWL Ontologies with SPARQL

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

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

Abstract. The SPARQL query language is currently being extended by W3C with so-called entailment regimes, which define how queries are evaluated under more expressive semantics than SPARQL's standard simple entailment. We describe a sound and complete algorithm for the OWL Direct Semantics entailment regime. The queries of the regime are very expressive since variables can occur within complex class expressions and can also bind to class or property names. We propose several novel optimizations such as strategies for determining a good query execution order, query rewriting techniques, and show how specialized OWL reasoning tasks and the class and property hierarchy can be used to reduce the query execution time. We provide a prototypical implementation and evaluate the efficiency of the proposed optimizations. For standard conjunctive 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. Several query languages have been designed for this purpose, including RDQL, SeRQL and, most recently, SPARQL. In this paper, we consider the SPARQL [8] query language, which was standardized in 2008 by the World Wide Web Consortium (W3C). The standard query evaluation mechanism is based on subgraph matching and is called simple entailment since it can equally be defined in terms of the simple entailment relation between RDF graphs. In order to use more elaborate entailment relations such as RDFS or OWL entailment [3], SPARQL 1.1 includes several *entailment regimes*. While several methods and implementations for SPARQL under RDFS semantics are available, methods that use OWL semantics have not yet been well-studied.

For some OWL 2 profiles, an implementation of the entailment regime can make use of materialization techniques (e.g., for the OWL RL profile) or of query rewriting techniques (e.g., for the OWL QL profile). These techniques are, however, not applicable in general, and may not deal directly with all kinds of SPARQL queries. In this paper, we present a sound and complete algorithm for answering SPARQL queries under the *OWL 2 Direct Semantics entailment regime* (from now on, SPARQL-OWL), describe a prototypical implementation based on the HermiT reasoner, and use this implementation to investigate a range of optimization techniques that improve query answering performance for different kinds of SPARQL-OWL queries.

SPARQL-OWL allows only distinguished variables (for compatibility with SPARQL 1.0), but it poses significant challenges for implementations, e.g., by allowing variables

that bind to classes or properties and which can even occur within complex class expressions. Amongst the query languages already supported by OWL reasoners, the closest in spirit to SPARQL-OWL is SPARQL-DL, which is implemented in the Pellet OWL reasoner [9]. SPARQL-DL is a subset of SPARQL-OWL that is designed such that queries can be mapped to standard reasoning tasks, such as retrieval of subclasses of a given class. 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.

We have implemented the optimized algorithm in a prototypical system, which 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 [6].

2 SPARQL-OWL Query Answering

We first give a brief introduction to the SPARQL-OWL entailment regime and then describe an algorithm that finds answers to queries under this regime. We abbreviate IRIs using the empty prefix, rdfs, and owl to refer to an imaginary example, the RDFS, and the OWL namespace, respectively. An example of a SPARQL query is

```
SELECT ?d WHERE { :op rdfs:domain ?d }
```

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 ObjectPropertyDomain(:op ?d) if :op is declared as an object property in the queried ontology. For brevity, we directly write BGPs in OWL's functional-style syntax (FSS) [7] in the remainder extended to allow for variables. We call such OWL axioms with variables *axiom templates* and a set of axiom templates a *template ontology*. In the absence of a FROM clause, the query is evaluated over the default ontology of the system. Evaluating a template ontology results in a sequence of solutions that lists possible (atomic) bindings for the variables. More complex WHERE clauses can be built by combining ontology templates using SPARQL operators such as UNION for alternative selection criteria or OPTIONAL to query for optional bindings [8, 4]. Since entailment regimes only change the evaluation of template ontologies (BGPs), whereas complex clauses are evaluated by combining already computed solutions, we focus here on the case where the WHERE clause consists of a single template ontology. We use O to denote the queried ontology and O_q for the template ontology.

Definition 1. *Let* V *be a countably infinite set of* variables*. The set of* variables *in a template ontology* $\mathsf{O_q}$ *is denoted by* $\mathsf{V}(\mathsf{O_q})$*. Let* O *be an ontology with signature* $S(\mathsf{O})$*, i.e.,* $S(\mathsf{O})$ *consists of all class, object property, data property, and individual names and literals occurring in* O*. We assume that* $S(\mathsf{O})$ *also contains OWL's special classes and properties such as* owl:Thing *and* owl:BottomObjectProperty*. A* solution mapping over $S(\mathsf{O})$ *is a partial function* $\mu\colon \mathsf{V} \to S(\mathsf{O})$ *from variables to entities in* $S(\mathsf{O})$*. For a solution mapping* μ *the set of elements on which* μ *is defined is the* domain $\mathsf{dom}(\mu)$ *of* μ*, and the set* $\mathsf{ran}(\mu) \coloneqq \{\mu(x) \mid x \in \mathsf{dom}(\mu)\}$ *is the* range *of* μ*. With* $\mu(\mathsf{O_q})$ *we denote the result of replacing each variable in* $\mathsf{O_q}$ *with a value specified by the mapping* μ*.*

Anonymous individuals in the template ontology are treated as variables whose bindings do not appear in the query's result sequence. This is in contrast to conjunctive queries where they are treated as existential variables. Anonymous individuals in the queried ontology are treated as (Skolem) constants, i.e., they can be returned in a query answer. For brevity, we assume here that neither the template ontology nor the queried ontology contains anonymous individuals. We further assume that all axiom templates that are evaluated by our algorithm can be instantiated into logical axioms since non-logical axioms (e.g., type declarations) do not affect the consequences of an ontology.

Definition 2. *Let* O *be an OWL 2 DL ontology with signature* $S(\mathsf{O})$*. The evaluation of* $\mathsf{O_q}$ *over* O *under OWL 2 Direct Semantics entailment is defined as the solution set* $\{\mu \mid \mathsf{O} \models \mu(\mathsf{O_q}), \mathsf{O} \cup \mu(\mathsf{O_q})$ *is an OWL 2 DL ontology,* $\mathsf{dom}(\mu) = \mathsf{V}(\mathsf{O_q}), \mathsf{ran}(\mu) \subseteq S(\mathsf{O})\}$*. We call* μ compatible with $\mathsf{O_q}$ and O *if* $\mathsf{dom}(\mu) = \mathsf{V}(\mathsf{O_q}), \mathsf{ran}(\mu) \subseteq S(\mathsf{O})$*, and* $\mu(\mathsf{O_q})$ *is such that* $\mathsf{O} \cup \mu(\mathsf{O_q})$ *is an OWL 2 DL ontology.*

Given an OWL 2 DL ontology O and a template ontology $\mathsf{O_q}$ for O, a straightforward algorithm to realize the entailment regime simply tests, for each compatible mapping μ, whether $\mathsf{O} \models \mu(\mathsf{O_q})$. The notion of compatible mappings already reduces the number of possible solutions that have to be tested, but in the worst case, the number of distinct compatible mappings μ is 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.

2.1 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 template ontology 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 axiom templates sequentially. 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 O_q = {ClassAssertion(:A ?x) ObjectPropertyAssertion(:op ?x ?y)} with :op an object property and :A a class. The query belongs to the class of conjunctive queries, i.e., we only query for class and property instances. We assume that the queried ontology contains 100 individuals, only 1 of which belongs to the class :A. This :A instance has 1 :op-successor, while we have overall 200 pairs of individuals related with the property :op. If we first evaluate ClassAssertion(:A ?x), we test 100 mappings (since x is an individual variable), of which only 1 mapping satisfies the axiom template. We then evaluate ObjectPropertyAssertion(:op ?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.

Instead of checking entailment, we can, for several axiom templates, directly retrieve the solutions from the reasoner. For example, to evaluate a query with O_q = {SubClassOf(?x :C)}, we can use standard reasoner methods to retrieve the subclasses. Most methods of reasoners are highly optimized, which can significantly reduce the number of tests that are performed. Furthermore, if the class hierarchy is precomputed, the reasoner can find the answers simply 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:

ClassAssertion(ObjectSomeValuesFrom(:op ?x) ?y)

Algorithm 1 shows how we evaluate template ontologies. 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 connectedComponents (line 2) partitions the axiom templates into sets of connected components, i.e., within a component the templates share common variables,

Algorithm 1 Query Evaluation Procedure

Input: O: the queried ontology, which is an OWL 2 DL ontology
$\mathsf{O_q}$: an OWL 2 DL template ontology
Output: a set of solutions for evaluating $\mathsf{O_q}$ over O under OWL 2 Direct Semantics

1: Axt := rewrite($\mathsf{O_q}$) {create a list Axt of simplified axiom templates from $\mathsf{O_q}$}
2: $\mathsf{Axt}^1, \ldots, \mathsf{Axt}^m$:=connectedComponents($\mathsf{Axt}$)
3: **for** j=1, ..., m **do**
4: $R_j := \{\mu_0 \mid \mathsf{dom}(\mu_0) = \emptyset\}$
5: $\mathsf{axt}_1, \ldots, \mathsf{axt}_n$:= reorder(Axt^j)
6: **for** $i = 1, \ldots, n$ **do**
7: $R_{new} := \emptyset$
8: **for** $\mu \in R_j$ **do**
9: **if** isSimple(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: $B := \{(\mu \cup \mu') \mid \mathsf{dom}(\mu') = \mathsf{V}(\mu(\mathsf{axt}_i)), (\mu \cup \mu')$ is compatible with axt_i and $\mathsf{O}\}$
13: $B := \mathsf{prune}(B, \mathsf{axt}_i, \mathsf{O})$
14: **while** $B \neq \emptyset$ **do**
15: $\mu' := \mathsf{removeNext}(B)$
16: **if** $\mathsf{O} \models \mu'(\mathsf{axt}_i)$ **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

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 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 or includes other solutions, based on easily available information, e.g., from the pre-computed class hierarchy.

2.2 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. Thus, we evaluate simple axiom templates first ordered by their cost, which is computed as the weighted sum of the estimated number of required

Table 1. Axiom templates and their equivalent simpler ones, where $C_{(i)}$ are class expressions (possibly containing variables), a is an individual or variable, and r is an object property expression (possibly containing a variable)

$$\begin{aligned}
\text{ClassAssertion(ObjectIntersectionOf(:C}_1 \ldots \text{:C}_n\text{) :a)} &\equiv \{\text{ClassAssertion(:C}_i\text{ :a)} \mid 1 \leq i \leq n\} \\
\text{SubClassOf(:C ObjectIntersectionOf(:C}_1 \ldots \text{:C}_n\text{))} &\equiv \{\text{SubClassOf(:C :C}_i\text{)} \mid 1 \leq i \leq n\} \\
\text{SubClassOf(ObjectUnionOf(:C}_1 \ldots \text{:C}_n\text{) :C)} &\equiv \{\text{SubClassOf(:C}_i\text{ :C)} \mid 1 \leq i \leq n\} \\
\text{SubClassOf(ObjectSomeValuesFrom(:op owl:Thing) :C)} &\equiv \text{ObjectPropertyDomain(:op :C)} \\
\text{SubClassOf(owl:Thing ObjectAllValuesFrom(:op :C))} &\equiv \text{ObjectPropertyRange(:op :C)}
\end{aligned}$$

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. Since the result sizes for complex templates are difficult to estimate using either the reasoner or the explicitly stated information in O, we order complex templates based only on the number of bindings that need to be tested. For example, the number of bindings for a class variable is the number of the classes appearing in O. 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:

SubClassOf(?x ObjectIntersectionOf(ObjectSomeValuesFrom(:op ?y) :C))

Its evaluation requires a quadratic number of consistency checks in the number of classes (since ?x and ?y are class variables). The rewriting yields:

SubClassOf(?x :C) and SubClassOf(?x ObjectSomeValuesFrom(:op ?y))

The first axiom template is now evaluated with a cheap cache lookup (assuming that the class hierarchy has been precomputed). For the second one, we only have to check the usually few resulting bindings for x combined with all other class names for y. For a complex axiom template such as the one in the last row of Table 1, the rewritten axiom template can be mapped to a specialized task of an OWL reasoner, which internally uses the class hierarchy to compute the domains and ranges with significantly fewer tests. 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.

Class-Property Hierarchy Exploitation The number of consistency checks needed to evaluate a template ontology can be further reduced by taking the class and property hierarchies into account. Once the classes and properties 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:

SubClassOf(:Infection ObjectSomeValuesFrom(:hasCausalLinkTo ?x))

If :C is not a solution and SubClassOf(:B :C) holds, then :B is also not a solution. Thus, when searching for solutions for x, the method removeNext (line 15) chooses the next binding to test by traversing the class hierarchy topdown. When we find a non-solution :C, the subtree rooted in :C of the class hierarchy can safely be pruned, which we do in the method prune in line 17. Queries over ontologies with a large number of classes and a deep class hierarchy can, therefore, gain the maximum advantage from this optimization. We employ similar optimizations using the property hierarchies. It is obvious that we only prune mappings that cannot constitute actual solution mappings, hence, soundness and completeness of Algorithm 1 is preserved.

Exploiting the Domain and Range Restrictions The implicit domains and ranges of the properties in O (in case the reasoner precomputes and stores them) and/or the explicit ones can be exploited to reduce the number of entailment checks that are needed. Let us assume that O contains Axiom (1) and O_q contains Axiom Template (2).

ObjectPropertyRange(:takesCourse :Course) (1)

SubClassOf(:GraduateStudent ObjectSomeValuesFrom(:takesCourse ?x)) (2)

In case at least one solution mapping exists for ?x, the class :Course and its superclasses can immediately be considered solution mappings for ?x. This is done in the method prune (line 13), which again preserves soundness and completeness.

3 System Evaluation

Since entailment regimes only change the evaluation of the template ontologies, standard SPARQL algebra processors can be used to combine the intermediate results, e.g., in unions or joins. Furthermore, standard OWL reasoners can be used to perform the required reasoning tasks.

3.1 The System Architecture

In our system, the queried ontology is loaded into an OWL reasoner and the reasoner performs initial tasks such as class classification before the system accepts queries. We use the ARQ library[3] of the Jena Semantic Web Toolkit for parsing the query and for the SPARQL algebra operations apart from template ontology evaluation. The template ontology is represented in a custom extension of the OWL API [5], which uses the queried ontology for type disambiguation. The resulting axiom templates are 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[4] for OWL reasoning, but only the module that generates statistics and provides cost estimations is HermiT specific.

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

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

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

3.2 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 costly operations such as the realization of the ABox 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.[5]

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 classes, 25 object properties, and 7 data properties. 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 sup-

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

Table 3. Sample complex queries for the GALEN ontology

1	SubClassOf(:Infection ObjectSomeValuesFrom(:hasCausalLinkTo ?x))
2	SubClassOf(:Infection ObjectSomeValuesFrom(?y ?x))
3	SubClassOf(?x ObjectIntersectionOf(:Infection ObjectSomeValuesFrom(:hasCausalAgent ?y)))
4	SubClassOf(:NAMEDLigament ObjectIntersectionOf(:NAMEDInternalBodyPart ?x) SubClassOf(?x ObjectSomeValuesFrom(:hasShapeAnalagousTo ObjectIntersectionOf(?y ObjectSomeValuesFrom(?z :linear))))
5	SubClassOf(?x :NonNormalCondition) SubObjectPropertyOf(?z :ModifierAttribute) SubClassOf(:Bacterium ObjectSomeValuesFrom(?z ?w)) SubObjectProperty(?y :StatusAttribute) SubClassOf(?w :AbstractStatus) SubClassOf(?x ObjectSomeValuesFrom(?y :Status))

ported 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 classes and 413 object properties. The ontology took 1.6 s to load and 4.8 s to classify the classes and properties. The execution time for these queries is shown on the right-hand side of Table 2. For each query, we tested the execution once without optimizations and once for each combination of applicable optimizations from Section 2.

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:

SubClassOf(NAMEDLigament NAMEDInternalBodyPart)
SubClassOf(NAMEDLigament ?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 SubClassOf(?y Shape) to Query 4 reduces the runtime from 68.2 s to 1.6 s.

4 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. Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P.F., Rudolph, S. (eds.): OWL 2 Web Ontology Language: Primer. W3C Recommendation (27 October 2009), available at http://www.w3.org/TR/owl2-primer/
4. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC (2009)
5. 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)
6. Kollia, I., Glimm, B., Horrocks, I.: SPARQL Query Answering over OWL Ontologies. In: Proc. 8th Extended Semantic Web Conf. (ESWC'11) (2011), to appear
7. Motik, B., Patel-Schneider, P.F., Parsia, B. (eds.): OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. W3C Recommendation (27 October 2009), available at http://www.w3.org/TR/owl2-syntax/
8. 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/
9. 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)