SPARQL Query Answering over OWL Ontologies

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 [10] query language, which was standardized in 2008 by the World Wide Web Consortium (W3C) and which is now supported by most RDF triple stores. The query evaluation mechanism defined in the SPARQL Query specification [10] is based on subgraph matching. This form of query evaluation is also 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 those induced by RDF Schema (RDFS) or OWL semantics [4], SPARQL 1.1 includes several *entailment regimes*, including RDFS and OWL. Query answering under such entailment regimes is more complex as it may involve retrieving answers that only follow implicitly from the queried graph. 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 of the less expressive 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

⋆ Work done while at the Oxford University Computing Lab.

G. Antoniou et al. (Eds.): ESWC 2011, Part I, LNCS 6643, pp. 382–396, 2011.

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.

The range of queries that can be formulated in SPARQL-OWL goes beyond standard conjunctive queries, which are already supported by several OWL reasoning systems. SPARQL-OWL does not allow for proper non-distinguished variables but it poses significant challenges for implementations, since, for example, variables can occur within complex class expressions and can also bind to class or property names. 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 [11]. SPARQL-DL is a subset of SPARQL-OWL that is designed such that queries can be mapped to standard reasoning tasks. 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.

2 Preliminaries

We first give a brief introduction to OWL and RDF, followed by the definition of SPARQL's syntax and semantics and the SPARQL-OWL entailment regime. We generally abbreviate International Resource Identifiers (IRIs) using the prefixes rdf, rdfs, and owl to refer to the RDF, RDFS, and OWL namespaces, respectively. The empty prefix is used for an imaginary example namespace.

2.1 Web Ontology Language OWL

For OWL, we use the functional-style syntax (FSS), which directly reflects the OWL objects that are used to define the OWL 2 Direct Semantics. In the following subsection, we clarify how the OWL structural objects can be mapped into RDF triples. We present only several examples of typical OWL axioms; for a full definition of OWL 2, please refer to the OWL 2 Structural Specification and Direct Semantics [8,7].

```
SubClassOf(:DogOwner ObjectSomeValuesFrom(:owns :Dog)))        (1)
SubClassOf(:CatOwner ObjectSomeValuesFrom(:owns :Cat)))        (2)
ObjectPropertyDomain(:owns :Person)                            (3)
```

ClassAssertion(ObjectUnionOf(:DogOwner :CatOwner) :mary) (4)

ObjectPropertyAssertion(:owns :mary _:somePet) (5)

Axioms (1) and (2) make use of existential quantification and state that every instance of the class :DogOwner (:CatOwner) is related to some instance of :Dog (:Cat) via the property :owns. Axiom (3) defines the domain of the property :owns as the class :Person, i.e., every individual that is related to some other individual with the :owns property belongs to the class :Person. Axiom (4) states that :mary belongs to the union of the classes :DogOwner and :CatOwner. Finally, Axiom (5) states that :mary owns some pet. The blank node _:somePet is called an *anonymous individual* in OWL and has an existential semantics. An OWL *ontology* contains a set of logical axioms, as the ones shown above, plus further non-logical statements, e.g., for the ontology header, type declarations (e.g., declaring :owns as an object property), or import directives. We focus on the logical axioms, which determine the logical consequences of the ontology.

More formally, the interpretation of axioms in an OWL ontology O is given by means of two-sorted interpretations over the *object domain* and the *data domain*, where the latter contains concrete values such as integers, strings, and so on. An *interpretation* maps classes to subsets of the object domain, object properties to pairs of elements from the object domain, data properties to pairs of elements where the first element is from the object domain and the second one is from the data domain, individuals to elements in the object domain, datatypes to subsets of the data domain, and literals (data values) to elements in the data domain. For an interpretation to be a *model* of an ontology, several conditions have to be satisfied [7]. For example, if O contains Axiom (4), then the interpretation of :mary must belong to the union of the interpretation of :DogOwner and :CatOwner. If an axiom ax is satisfied in every model of O, then we say that O *entails* ax, written $\mathsf{O} \models \mathsf{ax}$. For example, if O contains Axioms (1) to (5), then we have that O entails ClassAssertion(:Person :mary), i.e., we can infer that Mary is a person. This is because :mary will have an :owns-successor (due to Axiom (5)), which then implies that she belongs to the class :Person due to Axiom (3). In the same way, we say that an ontology O_1 entails another ontology O_2, written $\mathsf{O}_1 \models \mathsf{O}_2$, if every model of O_1 is also a model of O_2. The *vocabulary* $\mathsf{Voc}(\mathsf{O})$ of O is the set of all IRIs and literals that occur in O.

Note that the above axioms cannot be satisfied in a unique canonical model that could be used to answer queries since in one model we would have that :mary is a cat owner, whereas in another model, we would have that she is a dog owner. Thus, we cannot apply techniques such as forward chaining to materialize all consequences of the ontology. To satisfy the existential quantifiers (e.g, ObjectSomeValuesFrom), an OWL reasoner has to introduce new individuals, and in OWL it cannot be guaranteed that the models of an ontology are finite. OWL reasoners build, therefore, finite abstractions of models, which can be expanded into models.

2.2 Mapping to RDF Graphs

Since SPARQL is an RDF query language based on triples, we briefly show how the OWL objects introduced above can be mapped to RDF triples. The reverse direction, which maps triples to OWL objects is equally defined, but makes a well-formedness

Table 1. The RDF representation of Axioms (1), (3), and (4)

```
:DogOwner rdfs:subClassOf _:x .                    :owns rdfs:domain :Person .      (3')
_:x rdf:type owl:restriction .                     :mary rdf:type [ .
_:x owl:onProperty :owns .                           owl:unionOf
_:x owl:someValuesFrom :Dog .   (1')                   (:DogOwner :CatOwner ) ] (4')
```

restriction, i.e., only certain RDF graphs can be mapped into OWL structural objects. We call such graphs *OWL 2 DL graphs*. For further details, we refer interested readers to the W3C specification that defines the mapping between OWL structural objects and RDF graphs [9].

Table 1 gives an RDF representation of Axioms (1), (3), and (4) in Turtle syntax [1]. OWL axioms that only use RDF Schema expressivity, e.g., domain and range restrictions, usually result in a straightforward translation. For example, Axiom (3) is mapped to the single triple (3'). Complex class expressions such as the super class in Axiom (1) usually require auxiliary blank nodes, e.g., we introduce the auxiliary blank node _:x for the superclass expression that is then used as the subject of subsequent triples. In the translation of Axiom (4), we further used Turtle's blank node constructor [ ] and ( ) as a shortcut for lists in RDF.

Note that it is now no longer obvious whether :owns is a data or an object property. This is why an RDF graph that represents an OWL DL ontology has to contain type declarations, i.e., although we did not show the type declarations in our example, we would expect to have a triple such as :owns a owl:ObjectProperty, which corresponds to the non-logical axiom Declaration(ObjectProperty(:owns)) in FSS.

2.3 Syntax and Semantics of SPARQL Queries

We do not recall the complete surface syntax of SPARQL here but simply introduce the underlying algebraic operations using our notation. A detailed introduction to the relationship of SPARQL queries and their algebra is given in [5].

SPARQL supports a variety of *filter expressions*, or just *filters*, built from RDF terms, variables, and a number of built-in functions and operators; see [10] for details.

Definition 1. *We write* I *for the set of all* IRIs*,* L *for the set of all* literals*, and* B *for the set of all* blank nodes*. The set* T *of* RDF terms *is* $\mathsf{I} \cup \mathsf{L} \cup \mathsf{B}$*. Let* V *be a countably infinite set of* variables *disjoint from* T*. A* triple pattern *is member of the set* $(\mathsf{T} \cup \mathsf{V}) \times (\mathsf{I} \cup \mathsf{V}) \times (\mathsf{T} \cup \mathsf{V})$*, and a* basic graph pattern *(BGP) is a set of triple patterns. More complex* graph patterns *are inductively defined to be of the form* BGP*,* $\mathsf{Join}(\mathsf{GP}_1, \mathsf{GP}_2)$*,* $\mathsf{Union}(\mathsf{GP}_1, \mathsf{GP}_2)$*,* $\mathsf{LeftJoin}(\mathsf{GP}_1, \mathsf{GP}_2, F)$*, and* $\mathsf{Filter}(\mathsf{F}, \mathsf{GP})$*, where* BGP *is a BGP,* F *is a filter, and* $\mathsf{GP}_{(i)}$ *are graph patterns that share no blank nodes.*[1] *The sets of* variables *and* blank nodes *in a graph pattern* GP *are denoted by* $\mathsf{V}(\mathsf{GP})$ *and* $\mathsf{B}(\mathsf{GP})$*, respectively.*

We exclude a number of SPARQL features from our discussion. First, we disregard any of the new SPARQL 1.1 query constructs since their syntax and semantics are still

[1] As in [10], disallowing GP_1 and GP_2 to share blank nodes is important to avoid unintended co-references.

under discussion in the SPARQL working group. Second, we do not consider output formats (e.g., SELECT or CONSTRUCT) and solution modifiers (e.g., LIMIT or OFFSET) which are not affected by entailment regimes. Third, we exclude SPARQL datasets that allow SPARQL endpoints to cluster data into several named graphs and a default graph. Consequently, we omit dataset clauses and assume that queries are evaluated over the default graph, called the *active graph* for the query.

Evaluating a SPARQL graph pattern results in a sequence of solutions that lists possible bindings of query variables to RDF terms in the active graph.

Definition 2. *A* solution mapping *is a partial function* $\mu\colon \mathsf{V} \to \mathsf{T}$ *from variables to RDF terms. For a solution mapping* μ *– and more generally for any (partial) function – the set of elements on which* μ *is defined is the* domain $\mathsf{dom}(\mu)$ *of* μ*, and the set* $\mathsf{ran}(\mu) := \{\mu(x) \mid x \in \mathsf{dom}(\mu)\}$ *is the* range *of* μ*. For a BGP* BGP*, we use* $\mu(\mathsf{BGP})$ *to denote the pattern obtained by applying* μ *to all elements of* BGP *in* $\mathsf{dom}(\mu)$*. Two solution mappings* μ_1 *and* μ_2 *are* compatible *if* $\mu_1(x) = \mu_2(x)$ *for all* $x \in \mathsf{dom}(\mu_1) \cap \mathsf{dom}(\mu_2)$*. If this is the case, a solution mapping* $\mu_1 \cup \mu_2$ *is defined by setting* $(\mu_1 \cup \mu_2)(x) = \mu_1(x)$ *if* $x \in \mathsf{dom}(\mu_1)$*, and* $(\mu_1 \cup \mu_2)(x) = \mu_2(x)$ *otherwise.*

This convention is extended in the obvious way to all functions that are defined on variables or terms.

Since SPARQL allows for repetitive solution mappings and since the order of solution mappings is only relevant for later processing steps, we use *solution multisets*. A *multiset* over an *underlying set* $S = \{s_1, \ldots, s_n\}$ is a set of pairs (s_i, m_i) with m_i a positive natural number, called the *multiplicity* of s_i.

We first define the evaluation of BGPs under SPARQL's standard semantics, which is also referred to as simple entailment or subgraph matching. We still need to consider, however, the effect of blank nodes in a BGP. Intuitively, these act like variables that are projected out of a query result, and thus they may lead to duplicate solution mappings. This is accounted for using RDF instance mappings as follows:

Definition 3. *An* RDF instance mapping *is a partial function* $\sigma\colon \mathsf{B} \to \mathsf{T}$ *from blank nodes to RDF terms. The* solution multiset *for a basic graph pattern* BGP *over the active graph* G *is the following multiset of solution mappings:*

$$\{(\mu, n) \mid \mathsf{dom}(\mu) = \mathsf{V}(\mathsf{BGP}) \text{, and } n \text{ is the maximal number such that } \sigma_1, \ldots, \sigma_n \text{ are distinct RDF instance mappings with } \mathsf{dom}(\sigma_i) = \mathsf{B}(\mathsf{BGP}) \text{, for all } 1 \leq i \leq n \text{, and } \mu(\sigma_i(\mathsf{BGP})) \text{ is a subgraph of } \mathsf{G}\}.$$

The algebraic operators that are required for evaluating non-basic graph patterns correspond to operations on multisets of solution mappings, which are the same for all entailment regimes. Thus, we refer interested readers to the SPARQL Query specification [10] or the work about entailment regimes in general [2].

2.4 SPARQL-OWL

The SPARQL-OWL entailment regime[2] specifies how the OWL Direct Semantics entailment relation can be used to evaluate BGPs of SPARQL queries. The regime assumes that the queried RDF graph G as well as the BGP are first mapped to OWL 2

[2] http://www.w3.org/TR/2010/WD-sparql11-entailment-20101014/

structural objects, which are extended to allow for variables. Graphs or BGPs that cannot be mapped since they are not well-formed, are rejected with an error. We use O_G to denote the result of mapping an OWL 2 DL graph G into an OWL ontology.

An *axiom template* is an OWL axiom, which can have variables in place of class, object property, data property, or individual names or literals. In order to map a BGP into a set of axiom templates, the entailment regime specification extends the mapping between RDF triples and structural OWL objects. Type declarations from O_G are used to disambiguate types in BGP, but the regime further requires type declarations for variables. This allows for a unique mapping from a BPG into axiom templates. For example, without variable typing, the BGP :mary ?pred ?obj could be mapped into a data or an object property assertion. By adding the triple ?pred a owl:ObjectProperty, we can uniquely map the BGP to an object property assertion. Given an OWL 2 DL graph G, we call BGP *well-formed w.r.t.* G if it can uniquely be mapped into axiom templates taking also the type declarations from O_G into account. We denote the resulting set of axiom templates with $\mathsf{O}^\mathsf{G}_\mathsf{BGP}$.

SPARQL's standard BGP evaluation trivially guarantees finite answers since it is based on subgraph matching. Since the entailment regimes use an entailment relation in the definition of BGP evaluation, infinite solution mappings that only vary in their use of different blank node labels have to be avoided. Thus, entailment regimes make use of Skolemization, which treats the blank nodes in the queried graph basically as constants, but ones that do not have any particular fixed name. Since Skolem constants should not occur in query results, Skolemization is only used to restrict the solution mappings.

Definition 4. *Let the prefix* skol *refer to a namespace IRI that does not occur as the prefix of any IRI in the active graph or query. The* Skolemization sk(_:b) *of a blank node* _:b *is defined as* sk(_:b) := skol:b. *With* $\mathsf{sk}(\mathsf{O}_\mathsf{G})$ *we denote the result of replacing each blank node b in* O_G *with* sk(b). *Let* G *be an OWL 2 DL graph,* BGP *a BGP that is well-formed w.r.t.* G, *and* Voc(OWL) *the OWL vocabulary. The* answer domain w.r.t. G under OWL Direct Semantics entailment, *written* $\mathsf{AD}_\mathsf{DS}(\mathsf{G})$, *is the set* $\mathsf{Voc}(\mathsf{O}_\mathsf{G}) \cup \mathsf{Voc}(\mathsf{OWL})$. *The evaluation of* $\mathsf{O}^\mathsf{G}_\mathsf{BGP}$ *over* O_G *under OWL 2 Direct Semantics entailment is defined as the solution multiset*

$\{(\mu, n) \mid \mathsf{dom}(\mu) = \mathsf{V}(\mathsf{BGP})$, *and n is the maximal number such that*
$\sigma_1, \ldots, \sigma_n$ *are distinct RDF instance mappings such that, for each* $1 \leq i \leq n$,
i) $\mathsf{O}_\mathsf{G} \cup \mu(\sigma_i(\mathsf{O}^\mathsf{G}_\mathsf{BGP}))$ *is an OWL 2 DL ontology,*
ii) $\mathsf{sk}(\mathsf{O}_\mathsf{G}) \models \mathsf{sk}(\mu(\sigma_i(\mathsf{O}^\mathsf{G}_\mathsf{BGP})))$ *and*
iii) $(\mathsf{ran}(\mu) \cup \mathsf{ran}(\sigma_i)) \subseteq \mathsf{AD}_\mathsf{DS}(\mathsf{G})\}$.

Note that we only use Voc(OWL) for OWL's special class and property names such as owl:Thing or owl:TopObjectProperty.

3 SPARQL-OWL Query Answering

The SPARQL-OWL regime specifies what the answers are, but not how they can actually be computed. In this section, we describe an algorithm for SPARQL-OWL that internally uses any OWL 2 DL reasoner for checking entailment. We further describe optimizations that can be used to improve the performance of the algorithm by reducing

the number of entailment checks and method calls to the reasoner. We 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. Since class variables can only be instantiated with class names, object property variables with object properties, etc., we first define which solution mappings are relevant for our algorithm.

Definition 5. *Let* G *be an OWL 2 DL graph and* BGP *a BGP that is well-formed w.r.t.* G. *By a slight abuse of notation, we write* $\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}} = \{\mathsf{axt}_1, \ldots, \mathsf{axt}_n\}$ *for* $\mathsf{axt}_1, \ldots, \mathsf{axt}_n$ *the logical axiom templates in* $\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}}$. *For* μ *a solution mapping and* σ *an RDF instance mapping, we call* (μ, σ) compatible with $\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}}$ and O_{G} *if* $\mu(\sigma(\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}}))$ *is such that (a)* $\mathsf{O}_{\mathsf{G}} \cup \mu(\sigma(\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}}))$ *is an OWL 2 DL ontology and (b)* $\mu(\sigma(\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}}))$ *is ground and does not contain fresh entities w.r.t.* $\mathsf{sk}(\mathsf{O}_{\mathsf{G}})$.

Condition (a) ensures that condition (i) of the entailment regime is satisfied, which guarantees that the OWL 2 DL constraints are not violated, e.g., only simple object properties can be used in cardinality constraints. Condition (b) makes sure that the variables are only instantiated with the corresponding types since otherwise we would introduce a fresh entity w.r.t. $\mathsf{sk}(\mathsf{O}_{\mathsf{G}})$ (e.g., by using an individual name as a class) or even violate the OWL 2 DL constraints. Furthermore, the condition ensures that $\mu(\sigma(\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}}))$ contains no blank nodes (it is ground) and is Skolemized since all entities in the range of μ and σ are from $\mathsf{sk}(\mathsf{O}_{\mathsf{G}})$. Thus, condition (iii) of entailment regimes holds.

Given an OWL 2 DL graph G and a well-formed BGP BGP for G, a straightforward algorithm to realize the entailment regime now maps G into O_{G}, BGP into $\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}}$, and then simply tests, for each compatible pair (μ, σ), whether $\mathsf{sk}(\mathsf{O}_{\mathsf{G}}) \models \mu(\sigma(\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}}))$. The notion of compatible solutions already reduces the number of possible solutions that have to be tested, but in the worst case, the number of distinct compatible pairs (μ, σ) is exponential in the number of variables in the query, i.e., if m is the number of terms in O_{G} and n is the number of variables in $\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}}$, we test $O(m^n)$ solutions. 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 and blank nodes that can constitute actual solution and instance 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 ontology as a whole and, hence, does not take advantage of relations that may exist between axiom templates. For a more optimized BGP evaluation, we evaluate the BGP axiom template by axiom template. Initially, our solution set contains only the identity mapping, which does not map any variable or blank node 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.

As an example, consider the BGP { ?x rdf:type :A . ?x :op ?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 ?x rdf:type :A (i.e., 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 ?x :op ?y (i.e., 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 BGP { ?x rdfs:subClassOf :C }, which asks for subclasses of the class :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. Examples of complex axiom templates are:

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

Algorithm 1 shows how we evaluate a BGP. The algorithm takes as input an OWL 2 DL graph G and basic graph pattern BGP that is well-formed w.r.t. G. It returns a multiset of solution mappings that is the result of evaluating BGP over G under the OWL 2

Algorithm 1. Query Evaluation Procedure

Input: G: the active graph, which is an OWL 2 DL graph
BGP: an OWL 2 DL BGP
Output: a multiset of solutions for evaluating BGP over G under OWL 2 Direct Semantics

1: $\mathsf{O_G}$:=map(G)
2: $\mathsf{O^G_{BGP}}$:=map(BGP, $\mathsf{O_G}$)
3: Axt := rewrite($\mathsf{O^G_{BGP}}$) {create a list Axt of simplified axiom templates from $\mathsf{O^G_{BGP}}$}
4: $\mathsf{Axt^1}, \ldots, \mathsf{Axt^m}$:=connectedComponents(Axt)
5: **for** j=1, . . . , m **do**
6: $R_j := \{(\mu_0, \sigma_0) \mid \mathsf{dom}(\mu_0) = \mathsf{dom}(\sigma_0) = \emptyset\}$
7: $\mathsf{axt_1}, \ldots, \mathsf{axt_n}$:= reorder($\mathsf{Axt^j}$)
8: **for** $i = 1, \ldots, n$ **do**
9: $R_{new} := \emptyset$
10: **for** $(\mu, \sigma) \in R_j$ **do**
11: **if** isSimple($\mathsf{axt_i}$) **and** $((\mathsf{V}(\mathsf{axt_i}) \cup \mathsf{B}(\mathsf{axt_i})) \setminus (\mathsf{dom}(\mu) \cup \mathsf{dom}(\sigma))) \neq \emptyset$ **then**
12: $R_{new} := R_{new} \cup \{(\mu \cup \mu', \sigma \cup \sigma') \mid (\mu', \sigma') \in \mathsf{callReasoner}(\mu(\sigma(\mathsf{axt_i})))\}$
13: **else**
14: $B := \{(\mu \cup \mu', \sigma \cup \sigma') \mid \mathsf{dom}(\mu') = \mathsf{V}(\mu(\mathsf{axt_i})), \mathsf{dom}(\sigma') = \mathsf{B}(\sigma(\mathsf{axt_i})),$
$(\mu \cup \mu', \sigma \cup \sigma')$ is compatible with $\mathsf{axt_i}$ and $\mathsf{sk}(\mathsf{O_G})\}$
15: B := prune(B, $\mathsf{axt_i}$, $\mathsf{O_G}$)
16: **while** $B \neq \emptyset$ **do**
17: $(\mu', \sigma') :=$ removeNext(B)
18: **if** $\mathsf{O_G} \models \mu'(\sigma'(\mathsf{axt_i}))$ **then**
19: $R_{new} := R_{new} \cup \{(\mu', \sigma')\}$
20: **else**
21: $B :=$ prune($B, \mathsf{axt_i}, (\mu', \sigma')$)
22: **end if**
23: **end while**
24: **end if**
25: **end for**
26: $R_j := R_{new}$
27: **end for**
28: **end for**
29: $R := \{(\mu_1 \cup \ldots \cup \mu_m, \sigma_1 \cup \ldots \cup \sigma_m) \mid (\mu_j, \sigma_j) \in R_j, 1 \leq j \leq m\}$
30: **return** $\{(\mu, m) \mid m > 0$ is the maximal number with $\{(\mu, \sigma_1), \ldots, (\mu, \sigma_m)\} \subseteq R\}$

Direct Semantics. We first explain the general outline of the algorithm and leave the details of the used submethods for the following section. First, G and BGP are mapped to $\mathsf{O_G}$ and $\mathsf{O^G_{BGP}}$, respectively (lines 1 and 2). The function rewrite (line 3) can be assumed to do nothing. Next, the method connectedComponents (line 4) 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 29). For each component, we proceed as described below: we first determine an order (method reorder in line 7). 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 12). Otherwise, we check which compatible solutions yield an entailed axiom (lines 13 to 24). The method prune can again be assumed do nothing.

3.2 Optimized Query Evaluation

Axiom Template Reordering We now explain how we order the axiom templates in the method reorder (line 7). 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. The simple axiom templates are ordered by their cost, 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 and we do this for some simple templates, e.g., ObjectPropertyRange, 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 O_G, we order complex templates based only on the number of bindings that have to be tested, i.e., the number of consistency checks that are needed to evaluate them. 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 2 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))

This axiom template requires a quadratic number of consistency checks in the number of classes in the ontology (since ?x and ?y are class variables). According to Table 2, 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 2, 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 from Table 2 in the method rewrite in line 3 of our algorithm. Our evaluation in Section 4 shows a significant reduction in running time due to this axiom template rewriting. Soundness and completeness is preserved since instantiated rewritten templates are semantically equivalent to the corresponding instantiated complex ones.

Table 2. 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)

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

Class-Property Hierarchy Exploitation The number of consistency checks needed to evaluate a BGP 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. Let us assume that $\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}}$ contains the axiom template:

$$\text{SubClassOf(:Infection ObjectSomeValuesFrom(:hasCausalLinkTo ?x))}$$

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 17) 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 21. 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 object and data property 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 Domain and range restrictions in O_{G} can be exploited to further restrict the mappings for class variables. Let us assume that O_{G} contains Axiom (6) and $\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}}$ contains Axiom Template (7).

$$\text{ObjectPropertyRange(:takesCourse :Course)} \quad (6)$$

$$\text{SubClassOf(:GraduateStudent ObjectSomeValuesFrom(:takesCourse ?x))} \quad (7)$$

Only the class :Course and its subclasses can be solutions for x and we can immediately prune other mappings in the method prune (line 15), which again preserves soundness and completeness.

4 System Evaluation

Since entailment regimes only change the evaluation of basic graph patterns, standard SPARQL algebra processors can be used that allow for custom BGP evaluation. Furthermore, standard OWL reasoners can be used to perform the required reasoning tasks.

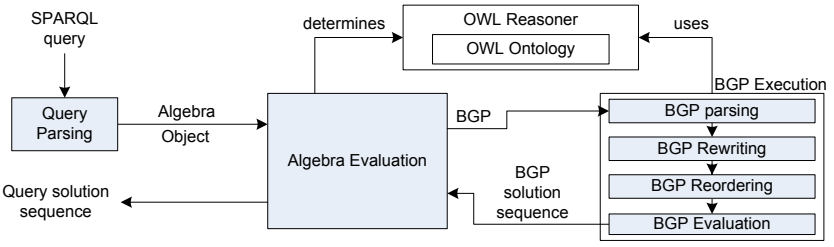

Fig. 1. The main phases of query processing in our system

4.1 The System Architecture

Figure 1 depicts the main phases of query processing in our prototypical system. In our setting, the queried graph is seen as an ontology that is loaded into an OWL reasoner. Currently, we only load the default graph/ontology of the RDF dataset into a reasoner and each query is evaluated using this reasoner. We plan, however, to extend the system to named graphs, where the dataset clause of the query can be used to determine a reasoner which contains one of the named ontologies instead of the default one. Loading the ontology and the initialization of the reasoner are performed 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 our custom BGP evaluation method. The BGP is parsed and mapped into axiom templates by our extension of the OWL API [6], which uses the active 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.

4.2 Experimental Results

We tested our system with the Lehigh University Benchmark (LUBM) [3] 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, 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.[5]

[3] http://jena.sourceforge.net/ARQ/
[4] http://www.hermit-reasoner.com/
[5] http://www.hermit-reasoner.com/2010/sparqlowl/sparqlowl.zip

Table 3. Query answering times in milliseconds for LUBM(1,0) and in seconds for the queries of Table 4 with and without optimizations

LUMB(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 4				
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

We first evaluate the 14 conjunctive ABox queries provided in the LUBM. 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 3 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 4 (in FSS). Note that we omit variable type declarations since the variable types are unambiguous in FSS. 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, 413 object properties, and no individuals or literals. 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 3. For each query, we tested 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.

Table 4. 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))

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.

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 to implement an algorithm that evaluates basic graph patterns under OWL's Direct Semantics. 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.

Future work will include the creation of more accurate cost estimates for the cost-based query reordering, the implementation of caching strategies that reduce the number of tests for different instantiations of a complex axiom template, and an extended evaluation using a broader set of ontologies and queries. Finally, we plan to analyze whether user specific profiles can be used to suggest additional restrictive axiom templates automatically to reduce the number of mappings that have to be checked.

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 (January 14, 2008), http://www.w3.org/TeamSubmission/turtle/
2. Glimm, B., Krötzsch, M.: SPARQL beyond subgraph matching. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 241–256. Springer, Heidelberg (2010)
3. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Semantics 3(2-3), 158–182 (2005)
4. Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P.F., Rudolph, S. (eds.): OWL 2 Web Ontology Language: Primer. W3C Recommendation (October 27, 2009), http://www.w3.org/TR/owl2-primer/
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. Motik, B., Patel-Schneider, P.F., Cuenca Grau, B. (eds.): OWL 2 Web Ontology Language: Direct Semantics. W3C Recommendation (October 27, 2009), http://www.w3.org/TR/owl2-direct-semantics/
8. Motik, B., Patel-Schneider, P.F., Parsia, B. (eds.): OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. W3C Recommendation (October 27, 2009), http://www.w3.org/TR/owl2-syntax/
9. Patel-Schneider, P.F., Motik, B. (eds.): OWL 2 Web Ontology Language: Mapping to RDF Graphs. W3C Recommendation (October 27, 2009), http://www.w3.org/TR/owl2-mapping-to-rdf/
10. Prud'hommeaux, E., Seaborne, A. (eds.): SPARQL Query Language for RDF. W3C Recommendation (January 15, 2008), http://www.w3.org/TR/rdf-sparql-query/
11. 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)