SPARQL-DL Implementation Experience

Petr Křemen[1] and Evren Sirin[2]

[1] Czech Technical University in Prague, Czech Republic
`kremen@labe.felk.cvut.cz`
[2] Clark & Parsia, LLC, Washington, DC, USA
`evren@clarkparsia.com`

Abstract. Recently, SPARQL-DL was introduced in [7] as a rich query language for OWL-DL ontologies. It provides an OWL-DL-like semantics for SPARQL basic graph patterns which involves as special cases both conjunctive ABox queries and mixed TBox/RBox/ABox queries over Description Logic (DL) ontologies. This paper describes the implementation of a SPARQL-DL query engine and discusses several optimizations. We investigate the new challenges brought by the additional expressivity of SPARQL-DL by extending a well-known benchmark.

1 Introduction

Query answering is a crucial inference service for ontology-based systems. *Conjunctive ABox queries* is the standard query service for DL-based systems, and is especially useful for applications with dominating ABox component.

Recently, the SPARQL-DL language was introduced in [7], as a language for posing *nonatomic mixed TBox/RBox/ABox queries* over OWL-DL ontologies. The ability to combine queries about the schema (classes and properties) and the data (individuals) brings new challenges to query answering.

This paper presents experience from prototypical implementation of two SPARQL-DL engines inside the open-source OWL-DL reasoner Pellet [3] We first present a naive approach where we separate the evaluation of schema queries and data queries. This approach is easier to implement using existing ABox query engines and can handle bnodes (undistinguished variables) in the query. The second approach is a mixed query evaluation strategy that cannot handle bnodes in queries but can be optimized better for SPARQL-DL queries.

We also present optimization techniques for answering SPARQL-DL queries. We generalize ABox query optimization techniques for SPARQL-DL; in short, queries are simplified during a preprocessing step and query atoms are reordered based on statistical analysis. We also provide a novel optimization technique for SPARQL-DL that make use of the class and property hierarchies to prune the search space during query evaluation.

In this paper, we also describe an initial design for a benchmark of SPARQL-DL queries based on the the existing ABox query benchmark LUBM [2]. Using this setup, we provide a preliminary evaluation of the query engines and the SPARQL-DL optimizations we implemented in Pellet.

[3] http://pellet.owldl.com

2 SPARQL-DL Language Overview

In this section we provide a brief description of SPARQL-DL and refer the reader to [7] for more details. Having an OWL-DL [1] ontology $\mathcal{O}$ with vocabulary $\mathcal{V}_O = (\mathcal{V}_{cls}, \mathcal{V}_{op}, \mathcal{V}_{dp}, \mathcal{V}_{ap}, \mathcal{V}_{ind}, \mathcal{V}_D, \mathcal{V}_{lit})$ and $\mathcal{V}_{var}$ (resp. $\mathcal{V}_{bnode}$) a set of variables (resp. bnodes) we define a *SPARQL-DL atom* q using the following expansion:

$$\begin{aligned} q \leftarrow\ & \mathsf{Type}(a, C) \mid \mathsf{PropertyValue}(a, p, d) \mid \mathsf{SameAs}(a_1, a_2) \mid \mathsf{DifferentFrom}(a_1, a_2) \\ & \mid \mathsf{SubClassOf}(C_1, C_2) \mid \mathsf{EquivalentClass}(C_1, C_2) \mid \mathsf{DisjointWith}(C_1, C_2) \\ & \mid \mathsf{ComplementOf}(C_1, C_2) \mid \mathsf{SubPropertyOf}(p_1, p_2) \mid \mathsf{EquivalentProperty}(p_1, p_2) \\ & \mid \mathsf{InverseOf}(p_1, p_2) \mid \mathsf{ObjectProperty}(p) \mid \mathsf{DataProperty}(p) \mid \mathsf{Annotation}(b_1, r, b_2) \\ & \mid \mathsf{Functional}(p) \mid \mathsf{InverseFunctional}(p) \mid \mathsf{Symmetric}(p) \mid \mathsf{Transitive}(p) \end{aligned}$$

where $a_{(i)} \in \mathcal{V}_{uri} \cup \mathcal{V}_{var} \cup \mathcal{V}_{bnode}$, $d \in \mathcal{V}_{uri} \cup \mathcal{V}_{var} \cup \mathcal{V}_{bnode} \cup \mathcal{V}_{lit}$, $C_{(i)} \in \mathcal{V}_{var} \cup S_c$, $p_{(i)} \in \mathcal{V}_{uri} \cup \mathcal{V}_{var}$, $b_i \in \mathcal{V}_{uri}$ and $r \in \mathcal{V}_{uri}$. The set S_c denotes the set of all OWL-DL concepts (both atomic and complex) built upon $\mathcal{V}_O$ (see [7]).

A SPARQL-DL query Q is a set $\{q_i\}_{i=1}^n$, also denoted as $q_1, \ldots, q_n$, interpreted as their conjunction. Each SPARQL-DL query allows for bnodes only in individual/literal position while allowing for distinguished variables at all other positions. In Section 4.1, we will show how we can get rid of bnodes mentioned in SameAs atoms, but in our implementation we assume that no bnodes appear in DifferentFrom atoms.

Having $\mathcal{O}$ and Q as above, we define SPARQL-DL semantics as follows. A *solution to query* Q is a mapping $\mu : \mathcal{V}_{var} \to \mathcal{V}_{cls} \cup \mathcal{V}_{op} \cup \mathcal{V}_{dp} \cup \mathcal{V}_{lit}$, such that, when applied to all variables in Q we get a *semiground query* $\mu(Q)$ (i.e. query containing no variables, but possibly containing bnodes), for which $\mathcal{O} \models \mu(Q)$. To decide whether $\mathcal{O} \models \mu(Q)$ we need to ensure that for each model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{O}$ there exists an evaluation function $\sigma : \mathcal{V}_{ind} \cup \mathcal{V}_{bnode} \cup \mathcal{V}_{lit} \to \Delta^{\mathcal{I}}$, that coincides with $\cdot^{\mathcal{I}}$ on literals and individuals, but provides *a mapping for a bnode to a domain element*, that satisfies ($\mathcal{I} \models_\sigma \mu(q_i)$) each semiground atom $\mu(q_i) \in Q$, see Table 1). The mapping σ ensures that bnodes behave like proper *undistinguished variables*, i.e. variables that can be bound not only to asserted individuals but also to inferred ones.

3 SPARQL-DL Query Examples

In this section, we provide some example queries that show how data and schema queries can be combined in SPARQL-DL. Our examples are based on the widely used LUBM dataset [2] which we extended for the evaluation of our implementation (see Section 6).

The LUBM dataset describes the university domain with information about departments, courses, students, and faculty. This dataset comes with 16 pure ABox queries with different characteristics (low vs. high selectivity, small vs. large input, etc.). In a similar fashion, we constructed 10 SPARQL-DL queries (all queries are available online[4]). Our primary goal was to exercise the novel

[4] http://svn.versiondude.net/clark-parsia/datasets/lubm/query-owled

Table 1. Interpretation of SPARQL-DL semi-ground atoms. The query atoms are abbreviated with their capital letters (e.g. SubClassOf is written as SCO) with the exception of Transitive atom which is abbreviated as Tr.

semi-ground $\mu(q)$	$\mathcal{I} \models_\sigma \mu(q)$ if:	semi-ground $\mu(q)$	$\mathcal{I} \models_\sigma \mu(q)$ if:
$\mathsf{T}(a, C)$	$\sigma(a) \in C^\mathcal{I}$	$\mathsf{SPO}(p_1, p_2)$	$p_1^\mathcal{I} \subseteq p_2^\mathcal{I}$
$\mathsf{PV}(a, p, d)$	$(\sigma(a), \sigma(d)) \in p^\mathcal{I}$	$\mathsf{EP}(p_1, p_2)$	$p_1^\mathcal{I} = p_2^\mathcal{I}$
$\mathsf{SA}(a_1, a_2)$	$\sigma(a_1) = \sigma(a_2)$	$\mathsf{IO}(p_1, p_2)$	$p_1^\mathcal{I} = (p_2^\mathcal{I})^-$
$\mathsf{DF}(a_1, a_2)$	$\sigma(a_1) \neq \sigma(a_2)$	$\mathsf{F}(p)$	$p^\mathcal{I}$ is a function
$\mathsf{SCO}(C_1, C_2)$	$C_1^\mathcal{I} \subseteq C_2^\mathcal{I}$	$\mathsf{IF}(p)$	$(p^\mathcal{I})^-$ is a function
$\mathsf{EC}(C_1, C_2)$	$C_1^\mathcal{I} = C_2^\mathcal{I}$	$\mathsf{S}(p)$	$p^\mathcal{I}$ is symmetric
$\mathsf{DW}(C_1, C_2)$	$C_1^\mathcal{I} \cap C_2^\mathcal{I} = \emptyset$	$\mathsf{Tr}(p)$	$p^\mathcal{I}$ is transitive
$\mathsf{CO}(C_1, C_2)$	$C_1^\mathcal{I} = \Delta^\mathcal{I} \setminus C_2^\mathcal{I}$	$\mathsf{A}(b_1, r, b_2)$	$(b_1^\mathcal{I}, b_2^\mathcal{I}) \in r^\mathcal{I}$

features of SPARQL-DL and identify the new challenges introduced by the additional expressivity SPARQL-DL provides over traditional ABox queries. The following examples demonstrate some of the features we used in these queries.

Example 1 (Variables in property position). Find all the graduate students that are related to a course and find what kind of relationship (e.g. `takesCourse`, `teachingAssistantOf`) it is:

$$\mathsf{Type}(?x, \texttt{GraduateStudent}), \mathsf{PropertyValue}(?x, ?y, ?z), \mathsf{Type}(?z, \texttt{Course})$$

Example 2 (Mixed ABox/TBox query). Find all the students who are also employees and find what kind of employee (e.g. `ResearchAssistant`) they are:

$$\mathsf{Type}(?x, \texttt{Student}), \mathsf{Type}(?x, ?C), \mathsf{SubClassOf}(?C, \texttt{Employee})$$

Asked against the LUBM dataset this query returns students who work as `ResearchAssistants`.

Example 3 (Mixed ABox/RBox query). Find all the members of `Dept0` and what kind of membership (e.g. `worksFor`, `headOf`) it is:

$$\mathsf{Type}(?x, \texttt{Person}), \mathsf{PropertyValue}(?x, ?y, \texttt{Dept0}), \mathsf{SubPropertyOf}(?y, \texttt{memberOf})$$

4 SPARQL-DL Query Answering

4.1 Preprocessing

It is possible to simplify a query by removing redundant atoms, i.e. atoms that are necessarily satisfied due to other atoms in the query. One such simplification is the *domain/range simplification* introduced in [6]. In addition, we can recursively get rid of SameAs atoms with bnodes. Algorithm 1 takes Q and returns a transformed Q' without bnodes in SameAs atoms such that $\mathcal{O} \models \mu(Q)$ iff $\mathcal{O} \models \mu(Q')$.

Algorithm 1 Preprocessing of SameAs atoms with bnodes

```
1: function simplifySameAs($Q$)
2:     remove from $Q$ all $q = \mathsf{SameAs}(a, a)$, where $a \notin \mathcal{V}_{var}$ or $a$ occurs just in $q$.
3:     if $q = \mathsf{SameAs}(b, a) \in Q$, or $q = \mathsf{SameAs}(a, b) \in Q$, where $b \in \mathcal{V}_{bnode}$ then
4:         apply $b \mapsto a$ to $Q$
5:         $Q \leftarrow$ simplifySameAs($Q$)
6:     return $Q$
7: end function
```

The procedure first removes all trivially satisfied SameAs atoms from the query, skipping those with variables without any other occurence in the query. Next, whenever a query Q in the set S contains a SameAs atoms with bnodes, all occurences of the bnode (or any of the bnodes) are replaced with the other term. For example $\mathsf{SameAs}(b, ?x), \mathsf{Type}(b, \mathtt{Person})$ is transformed to $\mathsf{Type}(?x, \mathtt{Person})$. However, this preprocessing cannot be used for SameAs atoms without bnodes, since $\mathcal{O}$ might contain individuals explicitly stated to be `owl : sameAs` in $\mathcal{O}$. For example, in $\mathsf{SameAs}(?x, ?y)$ it is necessary to bind to $?x$ and $?y$ to all combinations of individuals in the equivalence set induce by the `owl : sameAs` relation in $\mathcal{O}$. *Correctness* follows from the semantics of the SameAs atom (see Table 1) and the following idea: Since it is necessary to construct one σ evaluation for each interpretation $\mathcal{I}$, we can always choose $\sigma(b) = \sigma(\mu(a))$, if a is a variable, and $\sigma(b) = \sigma(a)$ otherwise.

The syntactic overhead of SPARQL-DL makes other simplifications possible. We can remove trivially satisfied atoms: (i) reflexive atoms with the same argument from the query that trivially hold, for example $\mathsf{SubClassOf}(\mathtt{C}, \mathtt{C})$, or $\mathsf{SubClassOf}(?x, ?x)$ whenever $?x$ is used at some other atom in the query; (ii) atoms with `owl:Thing` or `owl:Nothing` in a particular way, for example as in $\mathsf{SubClassOf}(\bullet, \mathtt{owl : Thing})$. On the other hand we can immediately fail queries for irreflexive atoms (e.g. DisjointWith, DifferentFrom) with the same argument. These preprocessing steps are not exhaustive but they are all quite cheap and since they decrease the number of atoms in the query, they are valuable especially w.r.t. the cost based reordering presented in Section 5.1.

The next preprocessing step is to split the query into connected components in order to avoid computing cross-products of their results. From now on, w.l.o.g. we will assume that a SPARQL-DL query consists of just one connected component.

4.2 Query Evaluation

Separated Evaluation To evaluate a SPARQL-DL query Q (possibly with bnodes) preprocessed using the above techniques we can make use of an *existing ABox query engine*. A SPARQL-DL query Q can be partitioned into two parts:

1. Q_c that contains all Type and PropertyValue atoms from Q, and
2. Q_s that contains all other atoms from Q.

Intuitively, Q_c represents an ABox (data) query whereas Q_s represents the TBox/RBox (schema) query. However, note that, Q_c is not a proper ABox query in the traditional sense since there might be variables in class or property positions. Furthermore, there might not be atoms in Q_s that mentions such variables. For this reason, we augment Q_s with atoms of the form $\mathsf{SubClassOf}(?x, ?x)$ (resp. $\mathsf{SubPropertyOf}(?x, ?x)$) for each $?x$ that occurs in $\mathsf{Type}(\bullet, ?x) \in Q_c$ (resp. $\mathsf{PropertyValue}(\bullet, ?x, \bullet) \in Q_c$) but does not occur in Q_s. Note that, these additional atoms have no effect to the query results but they ensure that Q_s will contain all class and property variables mentioned in the query.

Algorithm 2 is used to answer Q_s that consists of just one connected component (if there are more connected components we can evaluate each of them separately as in Section 4.1). The algorithm proceeds as follows: First, given an ordering W of query atoms, an atom q is chosen ($\textsc{next}(\mathcal{O}, W, B)$) for evaluation. Second, in q we replace variables bound in B with corresponding values. Resulting query atom q is then evaluated using function $\textsc{evalAtom}(\mathcal{O}, q, W_r, B, R)$. This function (i) checks whether $\mathcal{O} \models q$, if q is *ground*, (ii) explores possible bindings for all variables of q in other cases, using the following interface to the OWL-DL reasoner [5]:

allC$(\mathcal{O})$**, allP**$(\mathcal{O})$**, allI**$(\mathcal{O})$ return all named classes, resp. properties, resp. individuals defined in $\mathcal{O}$.

en$(\mathcal{O}, q)$ checks whether $\mathcal{O} \models q$, for a *ground SPARQL-DL atom* q. As SPARQL-DL atoms correspond to OWL-DL constructs/axioms their reduction to the basic inference services is straightforward from [7] and [4].

subC$(\mathcal{O}, \mathtt{C})$**, supC**$(\mathcal{O}, \mathtt{C})$**, eqC**$(\mathcal{O}, \mathtt{C})$ returns all subclasses, resp. superclasses, resp. equivalent classes of `C` in $\mathcal{O}$.

subP$(\mathcal{O}, \mathtt{p})$**, superP**$(\mathcal{O}, \mathtt{p})$**, eqP**$(\mathcal{O}, \mathtt{p})$ returns all subproperties, resp. superproperties, resp. equivalent properties to `p` in $\mathcal{O}$.

All atoms from Q_s are evaluated using the above reasoner interface. As an example, let's consider a call $\textsc{evalAtom}(\mathcal{O}, \mathsf{DisjointWith}(?x, ?y), W_r, , R)$. For each possible binding $\mathtt{C} \in \textsc{allC}(\mathcal{O})$ for $?x$ we try to get disjoint classes `D` of `C` using $\textsc{subC}(\mathcal{O}, \neg\mathtt{C})$. For each such binding we call $\textsc{eval}(\mathcal{O}, W_r, B \cup \{?x \mapsto \mathtt{C}, ?y \mapsto \mathtt{D}\}, R)$ to check the rest of the query (See the technical report [5] for a more detailed description of the $\textsc{evalAtom}(\bullet)$ algorithm).

Evaluation of Q_s generates a collection R of result bindings. Each $B \in R$ is then applied to Q_c, possibly binding some class/property variables, resulting in query Q_c^*. This query is then evaluated using the underlying conjunctive ABox query engine. One way to avoid unnecessary evaluation of Q_s is to evaluate it as soon as all of its variables get bound in B. Let's take an example, $\mathsf{Type}(?z, ?x), \mathsf{SubClassOf}(?x, \mathtt{Person}), \mathsf{EquivalentClasses}(?x, ?y)$. This query is split into $Q_c = \mathsf{Type}(?z, ?x)$ and $Q_s = \mathsf{SubClassOf}(?x, \mathtt{Person}), \mathsf{EC}(?x, ?y)$. Whenever a binding `C` for $?x$ is found during evaluation of $\mathsf{SubClassOf}$ we can

[5] Although these operations, together with those in Section 4.2, could be reduced to several consistency checks, this more high-level interface allows for well-known optimizations of classification/instance retrieval operations [3], [8]

Algorithm 2 SPARQL-DL Query Evaluation Procedure

```
1: function EVAL($\mathcal{O}, W, B, R$) ▷ $W$ is a list of query atoms, $B$ is the current variable binding, $R$ is the set of found bindings.
2:     if $W = \emptyset$ then return $R \cup \{B\}$
3:     $[q|W_r] \leftarrow$ NEXT($\mathcal{O}, W, B$)
4:     apply $B$ to the atom $q$.
5:     return EVALATOM($\mathcal{O}, q, W_r, B, R$)
6: end function
```

immediately evaluate $Q_c^* = \mathsf{Type}(?z, \mathtt{C})$, thus protecting the same query from being evaluated for each $?y$ binding.

The main advantage of this approach is that it makes use of an existing conjunctive ABox engine, thus providing a support for evaluation of bnodes for free. On the other hand, this approach might not have an optimal performance w.r.t. the reordering optimization introduced in Section 5.1.

Mixed Evaluation For queries without bnodes we can make a simple extension of (i) the EVALATOM(•) method for handling also Type and PropertyValue atoms and of (ii) the OWL-DL reasoner interface:

ir($\mathcal{O}$, C) returns all instances of class C in $\mathcal{O}$. Several strategies and optimizations can be considered here, e.g. linear vs. binary instance retrieval [3].

cr($\mathcal{O}$, a) returns all named classes that are types of a in $\mathcal{O}$. Strategies similar to those for IR($\mathcal{O}$, C) can be used.

This approach performs better w.r.t. the query reordering optimization described below. Unlike separated evaluation where Q_s is executed first and reordered separately from Q_c, in mixed evaluation the query is reordered as a whole and schema and the data parts can be evaluated in any order.

5 Optimizations

5.1 Cost-based Query Reordering

A query reordering optimization for ABox queries without bnodes is described in [6]. The idea is to estimate the cost of evaluating a query atom by estimating the number of answers to that query atom. The estimates are based on statistics computed by cheap preprocessing of the ontology. These estimates are then used to find a permutation of query atoms that will provide optimal execution.

For the purposes of SPARQL-DL, we generalized the reordering strategy mentioned above. The query is reordered and the ordering with minimal cost is chosen according to the Algorithm 3. The cost computation for given atom ordering is shown in Algorithm 4. For each query atom q we need two functions: EC($\mathcal{O}, B, q$), that estimates the cost needed to evaluate an atom q, and

$\text{EB}(\mathcal{O}, B, q)$, that estimates number of execution branches generated by evaluating q. Both functions take as an argument a collection B of bound variables, binding of which is unknown at given execution point.

Algorithm 3 Static Query Reordering

```
 1: function NEXT($\mathcal{O}, W, B$)
Require: $W \neq \emptyset$
 2:     if $B \neq \emptyset$ then return $W$
 3:     $W^* \leftarrow W$
 4:     $cost^* \leftarrow \infty$
 5:     for $W_p \in$ PERM($W$) do                ▷ PERM($W$) returns all permutations of $W$
 6:         if $cost^* >$ STATICCOST($W_p, \emptyset$) then
 7:             $cost^* \leftarrow$ STATICCOST($W_p, \emptyset$)
 8:             $W^* \leftarrow W_p$
 9:     return $W^*$
10: end function
```

Algorithm 4 Static Ordering Cost Computation

```
    function STATICCOST($W, B$)
        if $W = \emptyset$ then return 1
        $[q|W_r] \leftarrow W$
        return $\text{EC}(B, q) + \text{EB}(B, q) \cdot \text{STATICCOST}(W_r, B \cup \{\text{all variables in } q\})$
 5: end function
```

Knowledge Base Operation Costs For each knowledge base operation described, a typical cost can be estimated. We reduce each of these cost estimates to *four atomic costs*, which are parameters of the optimizer: $noSat$, $oneSat$, $classify$ and $realize$ for operations requiring respectively no consistency check, one consistency check, ontology classification and ontology realization. As an example let's take an atom $\mathsf{DisjointWith}(?x, \mathsf{C})$, evaluation of which is reduced to the $\text{SUBC}(\neg\mathsf{C})$. Cost estimates are reasoner dependent, for Pellet the call $\text{SUBC}(\bullet)$ would require ontology classification, thus $\text{EC}(\emptyset, \mathsf{DisjointWith}(?x, \mathsf{C})) = classify$. Having a non-empty B it is necessary to estimate *typical performance* of the operations, thus $\text{EC}(\{?x\}, \mathsf{DisjointWith}(?x, \mathsf{C})) = oneSat$.

Estimating number of candidates While the costs of knowledge base operations are dependent on the form of their arguments and on the KB state (classified, realized), the number of candidates for a given query atom can be estimated using a cheap preprocessing of $\mathcal{O}$, making use of two structures :

Information cached during consistency checking is used to get estimates for query atoms Type, PropertyValue, SameAs, DifferentFrom, like number of *types*, or number of *same individuals* for given individual, etc.

Told axioms are used to get estimates for other atoms, like number of *named subclasses* for given class, number of functional properties in $\mathcal{O}$, etc.

For example, to compute the number of branches for atom $\mathsf{SubClassOf}(?x, \mathtt{C})$, we would need to estimate how many *named subclasses* C has in $\mathcal{O}$. If C is a named class, we can simply use the told class taxonomy for this estimate. If C is a complex class then some kind of heuristics can be used, but we did not investigate this possibility in detail and focused on named concepts in queries.

5.2 Down-monotonic Variables

In addition to the general cost-based reordering of the query, we can also exploit the query structure. For example, consider the query

$$\mathsf{SubClassOf}(?x, \mathtt{Person}), \mathsf{Type}(?y, ?x)$$

which tries to retrieve all instances $?y$ of the class Person together with their actual type $?x$. When evaluating this query in this order we can exploit information from the class hierarchy and prevent variable $?x$ to be bound to subclasses of a class C which produced no result when used as a binding for $?x$. Thus, we can start exploring the class hierarchy from the top and prune the parts of the class hierarchy if the root of a part does produce no results.

For this purpose we introduce the following notion. At a given execution point, a *down-monotonic variable* is any variable $?x \in \mathcal{V}_{var}$ that occurs in a $\mathsf{Type}(\bullet, ?x)$, or $\mathsf{PropertyValue}(\bullet, ?x, \bullet)$, later in the query Q. Whenever the engine is about evaluating an atom q that contains a down-monotonic variable $?x$, it can safely perform the above described pruning.

6 Experiments

We have implemented both the separated and the mixed query evaluation strategies in the open-source OWL-DL reasoner Pellet. As mentioned in Section 3, instead of creating a new benchmark from scratch we decided to build our experiments on top of the LUBM dataset and queries. LUBM is a data-oriented benchmark with a smal schema (only several tens of classes and properties) and large number of instances (17000 individuals in the LUBM(1) dataset we used).

We ran the 10 SPARQL-DL queries we created against both the separated and the mixed query engines. In the mixed query engine we ran the experiments both with and without down monotonic variable optimization. Figure 1 shows the query execution times we got for these three instances of query engines.

The results show that the separated query evaluation can match the performance of mixed query engine in some cases but is mostly outperformed. Interestingly, there is a case (Q1 which corresponds to Example 1 from Section 3) where

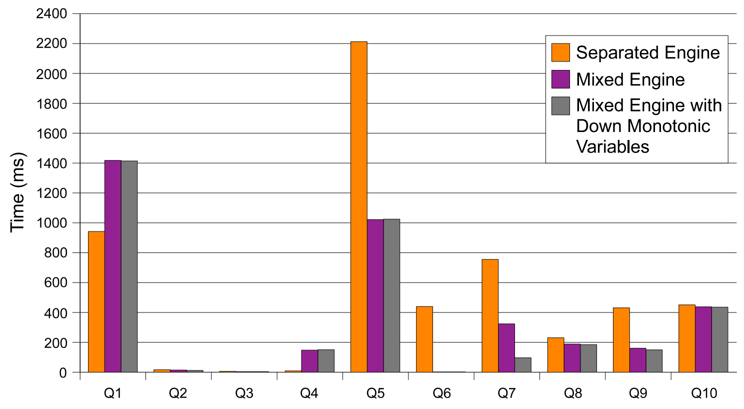

Fig. 1. Query answering times (in miliseconds) for LUBM(1) using different query engine configurations. All the timings are computed as the average of 10 independent runs on a Intel Core 2 Duo 2.33GHz machine with Java heap size set to 512mb.

separated engine performs significantly better. Investigating the query evaluation for this query revealed that sometimes evaluating TBox/RBox atoms first and then reordering the ABox query based on those results provide better results. For Q1, the separated engine tries properties in the KB one by one and changes the ordering of the ABox query for different properties. These reorderings are much more precise because they use the statistics for the actual property used while evaluating the query atom. Mixed evaluation, on the other hand, computes the reordering once at the beginning based on the overall statistics averaged over all properties and uses the same ordering for every property. This analysis indicates that dynamic reordering strategy in mixed evaluation would be valuable.

The results show that there is only one case (query 7) where the optimization for down-monotonic variables improves the performance. This is not too surprising considering that this optimization exploits the structure of the class/property hierarchies which are both shallow and narrow in the case of LUBM. The query 7 refers to `owl` : `Thing` (top of the class hierarchy) in a TBox axiom making the whole class hierarchy relevant for this query and that makes the effect of down monotonism most visible. It is also important to note that the optimization for down monotonic variable does not introduce any overhead for the other queries.

7 Conclusions and Future Work

In this paper, we have presented SPARQL-DL evaluation scheme and experience from its implementation in the Pellet reasoner. Two engines were shown: (i) a separated query engine that makes use of an existing ABox query engine as a black box, and (ii) a mixed query engine that answers queries without undistinguished variables. For both engines we have introduced two optimizations; a cost based query reordering and a notion of down-monotonic variables that prunes the search space of the query execution making use of concept/role hierarchies.

Our preliminary experiments show that the separated query evaluation works reasonably well in most cases. There are even some cases where it outperforms the mixed query engine which suggests the mixed query engine can be improved by a dynamic reordering method that we will investigate in the future. Also the cost-based reordering currently implemented is a preliminary prototype and will be revised to be applicable to ontologies with different characteristics.

Although not mentioned in the paper, the engine can be augumented in a simple way with non-monotonic atoms (like DirectSubClassOf), see [7]. These atoms seem to be useful from the practical point of view, where it is often valuable to get just the top-most/bottom-most element in the concept or role the taxonomy satisfying given constraints.

There are several other directions for future work. First, we will likely expand SPARQL-DL and Pellet's implementation of SPARQL-DL to cover more of the SPARQL language, like FILTERs and OPTIONALs. Second, we want to extend the mixed query engine with evaluation of queries with bnodes in the Type and PropertyValue atoms. Handling DifferentFrom atoms with bnodes is an open issue.

8 Acknowledgements

This work was done during Petr Křemen's internship at Clark & Parsia, LLC during the Semantic Summer 2007 program and was partially funded by the NATO Consultation, Command and Control Agency (NC3A) and by the grant No. MSM 6840770038 "Decision Making and Control for Manufacturing III" of the Ministry of Education, Youth and Sports of the Czech Republic; we thank Dave Clarke, Alex Tucker, Sven Kuehne, and Arno Bijl for their support. The authors would like to thank Bijan Parsia for insightful discussions about SPARQL-DL.

References

[1] OWL Web Ontology Language Semantics and Abstract Syntax. W3C Recommendation. http://www.w3.org/TR/2004/REC-owl-semantics-20040210/, 2004.

[2] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

[3] V. Haarslev and R. Moeller. Optimization strategies for instance retrieval, 2002.

[4] Ian Horrocks and Peter F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. *J. Web Sem.*, 1(4):345–357, 2004.

[5] Petr Kremen and Evren Sirin. Evaluating SPARQL-DL : First Experience. Technical Report GL 195/08, Czech Technical University in Prague, 2008.

[6] Evren Sirin and Bijan Parsia. Optimizations for answering conjunctive abox queries. In *Description Logics*, 2006.

[7] Evren Sirin and Bijan Parsia. SPARQL-DL: SPARQL Query for OWL-DL. In *OWLED*, 2007.

[8] Dmitry Tsarkov, Ian Horrocks, and Peter F. Patel-Schneider. Optimizing terminological reasoning for expressive description logics. *J. of Automated Reasoning*, 39(3):277–316, 2007.