# Cost based Query Ordering over OWL Ontologies

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

[1] University of Ulm, Germany, birte.glimm@uni-ulm.de
[2] National Technical University of Athens, Greece, ilianna2@mail.ntua.gr

**Abstract.** The paper presents an approach for cost-based query planning for SPARQL queries issued over an OWL ontology using the OWL Direct Semantics entailment regime of SPARQL 1.1. The costs are based on information about the instances of classes and properties that are extracted from a model abstraction built by an OWL reasoner. A static and a dynamic algorithm are presented which use these costs to find optimal or near optimal execution orders for the atoms of a query. For the dynamic case, we improve the performance by exploiting an individual clustering approach that allows for computing the cost functions based on one individual sample from a cluster. Our experimental study shows that the static ordering usually outperforms the dynamic one when accurate statistics are available. This changes, however, when the statistics are less accurate, e.g., due to non-deterministic reasoning decisions.

## 1 Introduction

Query answering—the computation of answers to users' queries w.r.t. ontologies and data—is an important task in the context of the Semantic Web that is provided by many OWL reasoners. Although much effort has been spent on optimizing the 'reasoning' part of query answering, i.e., the extraction of the individuals that are instances of a class or property, less attention has been given to optimizing the actual query answering part when ontologies in expressive languages are used. The SPARQL query language, which was standardized in 2008 by the World Wide Web Consortium (W3C), is widely used for expressing queries in the context of the Semantic Web. We use the OWL Direct Semantics entailment regime of SPARQL 1.1 according to which RDF triples from basic graph patterns are first mapped to extended OWL axioms which can have variables in place of classes, properties and individuals and are then evaluated according to the OWL entailment relation. We focus only on queries with variables in place of individuals since such queries are very common. We call the extended OWL axioms *query atoms* or *atoms*.

In the context of databases or triple stores, cost-based ordering techniques for finding an optimal or near optimal join ordering have been widely applied [12, 13]. These techniques involve the maintenance of a set of statistics about relations and indexes, e.g., number of pages in a relation, number of pages in an index, number of distinct values in a column, together with formulas for the estimation of the selectivity of predicates and the estimation of the CPU and I/O costs of query execution that depends amongst others, on the number of pages that have to be read from or written to secondary memory. The formulas for the estimation of selectivities of predicates (result output size of query atoms) estimate the data distributions using histograms, parametric or sampling methods or combinations of them.

In the context of reasoning over ontologies, the formulas should take the cost of executing specific reasoner tasks such as entailment checks or instance retrievals into account. The precise estimation of this cost before query evaluation is difficult as this cost takes values from a wide range. For example, the description logic $\mathcal{SROIQ}$, which underpins the OWL 2 DL standard, has a worst case complexity of 2-NExpTime [6] and typical implementations are not worst case optimal. The hypertableau algorithm that we use has a worst-case complexity of 3-NExpTime in the size of the ontology [9, 6].[3]

In this paper we address the optimization task of ordering the query atoms. The optimization goal is to find the execution plan (an order for the query atoms) which leads to the most efficient execution of the query by minimizing the number of needed reasoning tasks and the size of intermediate results. The execution plan which satisfies the above property is determined by means of a cost function that assigns costs to atoms within an execution plan. This cost function is based on heuristics and summaries for statistics about the data, which are extracted from an OWL reasoner model. We explore static and dynamic algorithms together with cluster based sampling techniques that greedily explore the execution plan search space to determine an optimal or near optimal execution plan. Static ordering refers to the finding of a join order before query evaluation starts whereas dynamic ordering determines the ordering of query atoms during query evaluation, taking advantage of already computed query atom results.

## 2 Preliminaries

In this section we briefly present an overview of the model building tableau and hypertableau calculi and give a brief introduction to SPARQL queries. For brevity, we use the description logic (DL) [1] syntax for examples.

Checking whether an individual $s_0$ (pair of individuals $\langle s_0, s_1 \rangle$) is an instance of a class C (property $R$) w.r.t. an ontology $\mathcal{O}$ is equivalent to checking whether the class assertion $\neg C(c_0)$ (the negation of the class assertion $C(s_0)$) or the property assertion $(\forall R.\neg\{s_1\})(s_0)$ ($s_0$ is only $R$-related to individuals that are not $s_1$) is inconsistent w.r.t. $\mathcal{O}$. To check this, most OWL reasoners use a model construction calculus such as tableau or hypertableau. In the remainder, we focus on the hypertableau calculus [9], but a tableau calculus could equally be used and we state how our results can be transferred to tableau calculi. The hypertableau calculus starts from an initial set of assertions and, by applying derivation rules, it tries to construct (an abstraction of) a model of $\mathcal{O}$. Derivation rules usually add new class and property assertion axioms, they may introduce new individuals, they can be nondeterministic, leading to the need to choose between several alternative assertions to add or they can lead to a clash when a contradiction is detected. To show that an ontology $\mathcal{O}$ is (in)consistent, the hypertableau calculus constructs a *derivation*, i.e., a sequence of sets of assertions $\mathcal{A}_0, \ldots, \mathcal{A}_n$, such that $\mathcal{A}_0$ contains all assertions in $\mathcal{O}$, $\mathcal{A}_{i+1}$ is the result of applying a derivation rule to $\mathcal{A}_i$ and $\mathcal{A}_n$ is the final set of assertions where no more rules are applicable. If a derivation exists such that

[3] The 2-NExpTime result for $\mathcal{SHOIQ}$+ increases to 3-NExpTime when adding role chains [6]

$A_n$ does not contain a clash, then $O$ is consistent and $A_n$ is called a *pre-model* of $O$. Otherwise $O$ is inconsistent. Each assertion in a set of assertions $A_i$ is derived either deterministically or nondeterministically. An assertion is *derived deterministically* if it is derived by the application of a deterministic derivation rule from assertions that were all derived deterministically. Any other derived assertion is *derived nondeterministically*. It is easy to know whether an assertion was derived deterministically or not because of the dependency directed backtracking that most (hyper)tableau reasoners employ. In the pre-model, each individual $s_0$ is assigned a label $\mathcal{L}(s_0)$ representing the classes it is (non)deterministically an instance of and each pair of individuals $\langle s_0, s_1\rangle$ is assigned a label $\mathcal{L}(\langle s_0, s_1\rangle)$ representing the properties through which individual $s_0$ is (non)deterministically related to individual $s_1$.

The WHERE clause of a SPARQL query consists of graph patterns. Basic graph patterns (BGPs) can be composed to more complex patterns using operators such as UNION and OPTIONAL for alternative and optional selection criteria. The evaluation of (complex) graph patterns is done by evaluating each BGP separately and combining the results of the evaluation. We only consider the evaluation of BGPs since this is the only thing that is specific to a SPARQL entailment regime. We further focus on conjunctive instance queries, i.e., BGPs that retrieve tuples of individuals, which are instances of the queried classes and properties. Such BGPs are first mapped to OWL class and (object) property assertions that allow for variables in place of individuals [7]. For brevity, we directly write mapped BGPs in DL syntax extended to allow for variables. We use the term *query* in the remainder for such BGPs. W.l.o.g., we assume that queries are connected [2].

**Definition 1.** *Let $O$ be an ontology with signature $S_O = (C_O, R_O, I_O)$, i.e., $S_O$ consists of all class, property and individual names occurring in $O$. We assume that $S_O$ also contains OWL's special classes and properties such as* owl:Thing. *Let $V$ be a countably infinite set of variables disjoint from $C_O$, $R_O$ and $I_O$. A* term *$t$ is an element from $V \cup I_O$. Let $C \in C_O$ be a class, $R \in R_O$ a property, and $t, t' \in I_O \cup V$ terms. A* class atom *is an expression $C(t)$ and a* property atom *is an expression $R(t, t')$. A* query *$q$ is a non-empty set of atoms. We use $Vars(q)$ to denote the set of variables and $Inds(q)$ for the set of individuals occurring in $q$ and set $Terms(q) = Vars(q) \cup Inds(q)$. We use $|q|$ to denote the number of atoms in $q$.*

*Let $q = \{at_1, \ldots, at_n\}$ be a query. A* mapping $\mu$ for $q$ over $O$ *is a total function $\mu\colon Terms(q) \to I_O$ such that $\mu(a) = a$ for each $a \in Inds(q)$. The set $\Gamma_q$ of all possible mappings for $q$ is defined as $\Gamma_q := \{\mu \mid \mu$ is a mapping for $q\}$. A* solution mapping $\mu$ for $q$ over $O$ *is a mapping such that $O \models C(\mu(t))$ for each class atom $C(t) \in q$ and $O \models R(\mu(t), \mu(t'))$ for each property atom $R(t, t') \in q$.*

## 3 Extracting Individual Information from Reasoner Models

The first step in the ordering of query atoms is the extraction of statistics, exploiting information generated by reasoners. We use the labels of an initial pre-model to provide us with information about the classes the individuals belong to or the properties in which they participate. We exploit this information similarly as was suggested for determining known (non-)subsumers for classes during classification [3]. In the hypertableau

**Algorithm 1** initializeKnownAndPossibleClassInstances

```
Require: a consistent $\mathcal{SROIQ}$ ontology $\mathcal{O}$ to be queried
Ensure: sets $K[C]$ ($P[C]$) of known (possible) instances for each class C of $\mathcal{O}$ are computed
$A_n$ := $buildModelFor(\mathcal{O})$
for all $ind \in \mathsf{I}_{\mathcal{O}}$ do
    for all $C \in \mathcal{L}_{A_n}(ind)$ do
        if $C$ was derived deterministically then
            $K[C] := K[C] \cup \{ind\}$
        else
            $P[C] := P[C] \cup \{ind\}$
        end if
    end for
end for
```

calculus, the following two properties hold for each ontology $\mathcal{O}$ and each constructed pre-model $A_n$ for $\mathcal{O}$:

(P1) for each class name $C$ (property $R$), each individual $s_0$ (pair of individuals $\langle s_1, s_2 \rangle$) in $A_n$, if $C \in \mathcal{L}_{A_n}(s_0)$ ($R \in \mathcal{L}_{A_n}(\langle s_1, s_2 \rangle)$) and the assertion $C(s_0)$ ($R(s_1, s_2)$) was derived deterministically, then it holds $\mathcal{O} \models C(s_0)$ ($\mathcal{O} \models R(s_1, s_2)$).

(P2) for an arbitrary individual $s_0$ in $A_n$ (pair of individuals $\langle s_1, s_2 \rangle$ in $A_n$) and an arbitrary class name C (simple property R), if $C \notin \mathcal{L}_{A_n}(s_0)$ ($R \notin \mathcal{L}_{A_n}(\langle s_1, s_2 \rangle)$), then $\mathcal{O} \not\models C(s_0)$ ($\mathcal{O} \not\models R(s_1, s_2)$).

The term simple property refers to a property that is neither transitive nor has a transitive subproperty.

We use these properties to extract information from the pre-model of a satisfiable ontology $\mathcal{O}$ as outlined in Algorithm 1. In our implementation we use a more complicated procedure to only store the direct types of each individual. The information we extract involves the maintenance of the sets of known and possible instances for all classes of $\mathcal{O}$. The *known instances* of a class $C$ ($K[C]$) are the individuals that can be safely considered instances of the class according to the pre-model, i.e., for each individual $i \in K[C]$, $C(i)$ was derived deterministically in the pre-model. Similarly, the *possible instances* of a class $C$ ($P[C]$) are the individuals $i$ such that $C(i)$ was derived nondeterministically. These possible instances require costly consistency checks in order to decide whether they are real instances of the class.

For simple properties, a procedure to find the known and possible instances of a property or, given an individual, the known and possible property successors or predecessors can be defined similarly. For non-simple properties, $\mathcal{O}$ is expanded with additional axioms that capture the semantics of the transitive relations before the *buildModelFor* procedure is applied since (hyper)tableau reasoners typically do not deal with transitivity directly [9]. In particular, for each individual $i$ and non-simple property $R$, new classes $C_i$ and $C_i^R$ are introduced and the axioms $C_i(i)$ and $C_i \sqsubseteq \forall R.C_i^R$ are added to $\mathcal{O}$. The consequent application of the transitivity encoding [9] produces axioms that propagate $C_i^R$ to each individual $s$ that is reachable from $i$ via an $R$-chain. The known and possible $R$-successors for $i$ can then be determined from the $C_i^R$ instances.

The technique presented in this paper can be used with any (hyper)tableau calculus for which properties (P1) and (P2) hold. All (hyper)tableau calculi used in practice that we are aware of satisfy property (P1). Pre-models produced by tableau algorithms as presented in the literature also satisfy property (P2); however, commonly used optimizations, such as lazy unfolding, can compromise property (P2), which we illustrate with the following example. Let us assume we have an ontology $\mathcal{O}$ containing the axioms $A \sqsubseteq \exists R.(C \sqcap D)$, $B \equiv \exists R.C$ and $A(a)$. It is obvious that in this ontology $A$ is a subclass of $B$ (hence $\mathcal{O} \models B(a)$) since every individual that is $R$-related to an individual that is an instance of the intersection of $C$ and $D$ is also $R$-related to an individual that is an instance of the class $C$. However, even though the assertion $A(a)$ occurs in the ABox, the assertion $B(a)$ is not added in the pre-model when we use lazy unfolding. With lazy unfolding, instead of treating $B \equiv \exists R.C$ as two disjunctions $\neg B \sqcup \exists R.C$ and $B \sqcup \forall R.(\neg C)$ as is typically done for complex class inclusion axioms, $B$ is only lazily unfolded into its definition $\exists R.C$ once $B$ occurs in the label of an individual. Thus, although $(\exists R.(C \sqcap D))(a)$ would be derived, this does not lead to the addition of $B(a)$.

Nevertheless, most (if not all) implemented calculi produce pre-models that satisfy at least the following weaker property:

(P3) for an arbitrary individual $s_0$ in $A_n$ and an arbitrary class C where C is primitive in $\mathcal{O}$,[4] if $C \notin \mathcal{L}_{A_n}(s_0)$, then $\mathcal{O} \not\models C(s_0)$.

Hence, properties (P2) and (P3) can be used to extract (non-)instance information from pre-models. For tableau calculi that only satisfy (P3), Algorithm 1 can be modified accordingly. In particular, for each non-primitive class C in $\mathcal{O}$ we need to add to P[C] the individuals in $\mathcal{O}$ that do not include the class C in their label.

The proposed technique for determining known and possible instances of classes and properties can be used in the same way with both tableau and hypertableau reasoners. Since tableau algorithms often introduce more nondeterminism than hypertableau, one might, however, find less deterministic derivations, which results in less accurate statistics.

### 3.1 Individual Clustering

In this section, we describe the procedure for creating clusters of individuals within an ontology $\mathcal{O}$ using the constructed pre-model $A_n$ of $\mathcal{O}$. Two types of clusters are created: class clusters and property clusters. Class clusters contain individuals having the same classes in their label and property clusters contain individuals with the same class and property labels. Property clusters are divided into three categories, those that are based on the first individual of property instances, those based on the second individual and those based on both individuals. First we define, for an ontology $\mathcal{O}$ with pre-model $A_n$, the relations $P_1$ and $P_2$ that map an individual $a$ from $\mathcal{O}$ to the properties for which $a$ has at least one successor or predecessor, respectively:

$$P_1(a) = \{R \mid \exists b \in I_{\mathcal{O}} \text{ such that } R \in \mathcal{L}_{A_n}(\langle a, b\rangle)\}$$
$$P_2(a) = \{R \mid \exists b \in I_{\mathcal{O}} \text{ such that } R \in \mathcal{L}_{A_n}(\langle b, a\rangle)\}$$

[4] A class C is considered primitive in $\mathcal{O}$ if $\mathcal{O}$ is unfoldable [14] and it contains no axiom of the form $C \equiv E$

Based on these relations, we partition $I_O$ into class clusters $CC = \{C^1, \ldots, C^n\}$, property successor clusters $PC_1 = \{C_1^1, \ldots, C_1^n\}$, property predecessor clusters $PC_2 = \{C_2^1, \ldots, C_2^n\}$ and $I_O \times I_O$ into property clusters $PC_{12} = \{C_{12}^1, \ldots, C_{12}^n\}$ such that the clusters satisfy:

$$\begin{aligned}
\forall C \in CC.(\forall a_1, a_2 \in C.(\mathcal{L}_{A_n}(a_1) = \mathcal{L}_{A_n}(a_2)))\\
\forall C \in PC_1.(\forall a_1, a_2 \in C.(\mathcal{L}_{A_n}(a_1) = \mathcal{L}_{A_n}(a_2) \text{ and } P_1(a_1) = P_1(a_2)))\\
\forall C \in PC_2.(\forall a_1, a_2 \in C.(\mathcal{L}_{A_n}(a_1) = \mathcal{L}_{A_n}(a_2) \text{ and } P_2(a_1) = P_2(a_2)))\\
\forall C \in PC_{12}.(\forall \langle a_1, a_2\rangle, \langle a_3, a_4\rangle \in C.(\mathcal{L}_{A_n}(a_1) = \mathcal{L}_{A_n}(a_3), \mathcal{L}_{A_n}(a_2) = \mathcal{L}_{A_n}(a_4) \text{ and}\\
\mathcal{L}_{A_n}(\langle a_1, a_2\rangle) = \mathcal{L}_{A_n}(\langle a_3, a_4\rangle)))
\end{aligned}$$

## 4 Query Answering and Query Atom Ordering

In this section, we describe two different algorithms (a static and a dynamic one) for ordering the atoms of a query based on some costs and then we deal with the formulation of these costs. We first introduce the abstract graph representation of a query $q$ by means of a labeled graph $G_q$ on which we define the computed statistical costs.

**Definition 2.** *A* query join graph *$G_q$ for a query $q$ is a tuple $(V, E, E_L)$, where*

- *$V = q$ is a set of vertices (one for each query atom);*
- *$E \subseteq V \times V$ is a set of edges such that $\langle at_1, at_2\rangle \in E$ iff $Vars(at_1) \cap Vars(at_2) \neq \emptyset$ and $at_1 \neq at_2$;*
- *$E_L$ is a function that assigns a set of variables to each $\langle at_1, at_2\rangle \in E$ such that $E_L(at_1, at_2) = Vars(at_1) \cap Vars(at_2)$.*

In the remainder, we use $a, b$ for individual names, $x, y$ for variables, $q$ for a query $\{at_1, \ldots, at_n\}$ with query join graph $G_q$, and $\Omega_q$ for the solution mappings of $q$.

Our goal is to find a query execution plan, which determines the evaluation order for atoms in $q$. Since the number of possible execution plans is of order $|q|!$, the ordering task quickly becomes impractical. In the following, we focus on greedy algorithms for determining an execution order, which prune the search space considerably. Roughly speaking, we proceed as follows: We define a cost function, which consists of two components (i) an estimate for the reasoning costs and (ii) an estimate for the intermediate result size. Both components are combined to induce an order among query atoms. In this paper, we simply build the sum of the two cost components, but different combinations such as a weighted sum of the two values could also be used. For the query plan construction we distinguish *static* from *dynamic planning*. For the former, we start constructing the plan by adding a minimal atom according to the order. Variables from this atom are then considered bound, which changes the cost function and might induce a different order among the remaining query atoms. Considering the updated order, we again select the minimal query atom that is not yet in the plan and update the costs. This process continues until the plan contains all atoms. Once a complete plan has been determined the atoms are evaluated. The dynamic case differs in that after selecting an atom for the plan, we immediately determine the solutions for the chosen atom, which are then used to update the cost function. While this yields accurate cost estimates, it can be very costly when all solutions are considered for updating the cost function. Sampling techniques can be used to only test a subset of the solutions, but we show in Section 5 that random sampling, i.e., randomly choosing a percentage of the individuals from the so far computed solutions, is not adequate. For this reason, we propose an alternative sampling approach that is based on the use of the previously described individual clusters. We now make the process of query plan construction more precise, but we leave the exact details of defining the cost function and the ordering it induces to later.

**Definition 3.** *A* static (dynamic) cost function w.r.t. *$q$ is a function $s\colon q \times 2^{Vars(q)} \rightarrow \mathbb{R} \times \mathbb{R}$ ($d\colon q \times 2^{\Gamma_q} \rightarrow \mathbb{R} \times \mathbb{R}$). The two costs are combined to yield a* static ordering $\preceq_s$ *(*dynamic ordering $\preceq_d$*), which is a total order over the atoms of $q$.*

*An* execution plan *for $q$ is a duplicate-free sequence of query atoms from $q$. The* initial execution plan *is the empty sequence and a* complete execution plan *is a sequence containing all atoms of $q$. For $P_i = (at^1, \ldots, at^i)$ with $i < n$ an execution plan for $q$ with query join graph $G_q = (V, E, E_L)$, we define the* potential next atoms $q_i$ for $P_i$ w.r.t. *$G_q$ as $q_i = q$ for $P_i$ the initial execution plan and $q_i = \{at \mid \langle at', at\rangle \in E, at' \in \{at^1, \ldots, at^i\}, at \in q \setminus \{at^1, \ldots, at^i\}\}$ otherwise. The static (dynamic) ordering induces an execution plan $P_{i+1} = (at^1, \ldots, at^i, at^{i+1})$ with $at^{i+1} \in q_i$ and $at^{i+1} \preceq_s at$ ($at^{i+1} \preceq_d at$) for each $at \in q_i$ such that $at \neq at^{i+1}$.*

For $i > 0$, the set of potential next atoms only contains atoms that are connected to an atom that is already in the plan since unconnected atoms will cause an unnecessary blowup of the number of intermediate results. Let $P_i = (at_1, \ldots, at_i)$ with $i \leq n$ be an execution plan for $q$. The procedure to find the solution mappings $\Omega_i$ for $P_i$ is recursively defined as follows: Initially, our solution set contains only the identity mapping $\Omega_0 = \{\mu_0\}$, which does not map any variable to any value. Assuming that we have evaluated the sequence $P_{i-1} = (at_1, \ldots, at_{i-1})$ and we have found the set of solution mappings $\Omega_{i-1}$, in order to find the solution mappings $\Omega_i$ of $P_i$, we use the instance retrieval tasks of reasoners to extend the mappings in $\Omega_{i-1}$ to cover the new variables of $at_i$ or the entailment check service of reasoners if $at_i$ does not contain new variables. A detailed description of the method for evaluating a query atom together with optimizations can be found in our previous work [7].

We now define the cost functions $s$ and $d$ more precisely, which estimate the cost of the required reasoner operations (first component) and the estimated result output size (second component) of evaluating a query atom. The intuition behind the estimated value of the reasoner operation costs is that the evaluation of possible instances is much more costly than the evaluation of known instances since possible instances require expensive consistency checks whereas known instances require cheap cache lookups. The estimated result size takes into account the number of known and possible instances and the probability that possible instances are actual instances. Apart from the relations $K[C]$ and $P[C]$ ($K[R]$ and $P[R]$) for the known and possible instances of a class $C$ (property $R$) from Section 3, we use the following auxiliary relations:

**Definition 4.** *Let $R$ be a property and $a$ an individual. We define* $\mathsf{sucK}[R]$ *and* $\mathsf{preK}[R]$ *as the set of individuals with known $R$-successors and $R$-predecessors, respectively:*

$$\mathsf{sucK}[R] := \{i \mid \exists j.\langle i, j\rangle \in K[R]\} \quad \textit{and} \quad \mathsf{preK}[R] := \{i \mid \exists j.\langle j, i\rangle \in K[R]\}.$$

*Similarly, we define* $\mathsf{sucK}[R, a]$ *and* $\mathsf{preK}[R, a]$ *as the known R-successors of a and the known R-predecessors of a, respectively:*

$$\mathsf{sucK}[R, a] := \{i \mid \langle a, i\rangle \in K[R]\} \qquad \textit{and} \qquad \mathsf{preK}[R, a] := \{i \mid \langle i, a\rangle \in K[R]\}.$$

*We analogously define the functions* $\mathsf{sucP}[R]$*,* $\mathsf{preP}[R]$*,* $\mathsf{sucP}[R, a]$*, and* $\mathsf{preP}[R, a]$ *by replacing* $P[C]$ *and* $P[R]$ *with* $K[C]$ *and* $K[R]$*, respectively. We write* $C_L$ *to denote the cost of a cache lookup in the internal structures of the reasoner,* $C_E$ *for the cost of an entailment check, and* $P_{IS}$ *for the possible instance success, i.e, the estimated percentage of possible instances that are actual instances.*

The costs $C_L$ and $C_E$ are determined by recording the average time of previously performed lookups and entailment checks for the queried ontology, e.g., during the initial consistency check, classification, or for previous queries. In the case of $C_E$, we multiply this number with the depth of the class (property) hierarchy since we only store the direct types of each individual (properties in which each individual participates) and, in order to find the instances of a class (property), we may need to check all its subclasses (subproperties) that contain possible instances. The time needed for an entailment check can change considerably between ontologies and even within an ontology (depending on the involved classes, properties, and individuals). Thus, the use of a single constant for the entailment cost is not very accurate, however, the definition of different entailment costs before executing the reasoning task is very difficult.

The possible instance success, $P_{IS}$, was determined by testing several ontologies and checking how many of the initial possible instances were real ones, which was around 50% in nearly all ontologies.

### 4.1 The Static and Dynamic Cost Functions

The static cost function $s$ takes two components as input: a query atom and a set containing the variables of the query atom that are considered bound. The function returns a pair of real numbers for the reasoning cost and result size for the query atom.

Initially, all variables are unbound and we use the number of known and possible instances or successors/predecessors to estimate the number of required lookups and consistency checks for evaluating the query atom and for the resulting number of mappings. For an input of the form $\langle C(x), \emptyset\rangle$ or $\langle R(x, y), \emptyset\rangle$ the resulting pair of real numbers for the computational cost and the estimated result size is computed as

$$\langle |K[at]| \cdot C_L + |P[at]| \cdot C_E, |K[at]| + P_{IS} \cdot |P[at]|\rangle,$$

where $at$ denotes the predicate of the query atom ($C$ or $R$). If the query atom is a property atom with a constant in the first place, i.e., the input to the cost function is of the form $\langle R(a, x), \emptyset\rangle$, we use the relations for known and possible successors to estimate the computational cost and result size:

$$\langle |\mathsf{sucK}[R, a]| \cdot C_L + |\mathsf{sucP}[R, a]| \cdot C_E, |\mathsf{sucK}[R, a]| + P_{IS} \cdot |\mathsf{sucP}[R, a]|\rangle.$$

Analogously, we use $\mathsf{preK}$ and $\mathsf{preP}$ instead of $\mathsf{sucK}$ and $\mathsf{sucP}$ for an input of the form $\langle R(x, a), \emptyset\rangle$. Finally, if the atom contains only constants, i.e., the input to the cost function is of the form $\langle C(a), \emptyset\rangle$, $\langle R(a, b), \emptyset\rangle$, the function returns $\langle C_L, 1\rangle$ if the individual is a known instance of the class or property, $\langle C_E, P_{IS} \rangle$ if the individual is a possible instance and $\langle C_L, 0 \rangle$ otherwise, i.e., if the individual is a known non-instance.

After determining the cost of an initial query atom, at least one variable of a consequently considered atom is bound, since during the query plan construction we move over atoms sharing a common variable and we assume that the query is connected. We now define the cost functions for atoms with at least one variable bound. We make the assumption that atoms with unbound variables are more costly to evaluate than atoms with all their variables bound. For a query atom $R(x, y)$ with only $x$ bound, i.e., function inputs of the form $\langle R(x, y), \{x\} \rangle$, we use the average number of known and possible successors of the property to estimate the computational cost and result size:

$$\left\langle \frac{|K[R]|}{|\mathsf{sucK}[R]|} \cdot C_L + \frac{|P[R]|}{|\mathsf{sucP}[R]|} \cdot C_E, \frac{|K[R]|}{|\mathsf{sucK}[R]|} + \frac{|P[R]|}{|\mathsf{sucP}[R]|} \cdot P_{IS} \right\rangle$$

In case only $y$ in $R(x, y)$ is bound, we use the predecessor functions $\mathsf{preK}$ and $\mathsf{preP}$ instead of $\mathsf{sucK}$ and $\mathsf{sucP}$. Note that we now work with an estimated average number of successors (predecessors) for *one individual*.

For the remaining cases (atoms with all their variables bound), we use formulas that are comparable to the ones above for an initial plan, but normalized to estimate the values for one individual. The normalization is important for achieving compatibility with the formulas described above for inputs of the form $\langle R(x, y), \{x\} \rangle$ and $\langle R(x, y), \{y\} \rangle$. For an input query atom of the form $C(x)$ with $x$ a bound variable we use

$$\left\langle \frac{|K[C]|}{|I_O|} \cdot C_L + \frac{|P[C]|}{|I_O|} \cdot C_E, \frac{|K[C]| + P_{IS} \cdot |P[C]|}{|I_O|} \right\rangle$$

Such a simple normalization is not always accurate, but leads to good results in most cases as we show in Section 5. Similarly, we normalize the formulae for property atoms of the form $R(x, y)$ such that $\{x, y\}$ is the set of bound variables of the atom. The two cost components for these atoms are computed as

$$\left\langle \frac{|K[R]|}{|I_O|} \cdot C_L + \frac{|P[R]|}{|I_O|} \cdot C_E, \frac{|K[R]| + P_{IS} \cdot |P[R]|}{|I_O| \cdot |I_O|} \right\rangle$$

For property atoms with a constant and a bound variable, i.e., atoms of the form $R(a, x)$ ($R(x, a)$) with $x$ a bound variable, we use $\mathsf{sucK}[R, a]$ and $\mathsf{sucP}[R, a]$ ($\mathsf{preK}[R, a]$ and $\mathsf{preP}[R, a]$) instead of $K[R]$ and $P[R]$ in the above formulae.

The dynamic cost function $d$ is based on the static function $s$, but only uses the first equations, where the atom contains only unbound variables or constants. The function takes a pair $\langle at, \Omega \rangle$ as input, where $at$ is a query atom and $\Omega$ is the set of solution mappings for the atoms that have already been evaluated, and returns a pair of real numbers using matrix addition as follows:

$$d(at, \Omega) = \sum_{\mu \in \Omega} s(\mu(at), \emptyset)$$

When sampling techniques are used, we compute the costs for each of the potential next atoms for an execution plan by only considering one individual of each relevant

**Table 1.** Query Ordering Example

| | Atom Sequences | Known Instances | Possible Instances | Real from Possible Instances |
|---|---|---|---|---|
| 1 | C(x) | 200 | 350 | 200 |
| 2 | R(x,y) | 200 | 200 | 50 |
| 3 | D(y) | 700 | 600 | 400 |
| 4 | R(x,y), C(x) | 100 | 150 | 100 |
| 5 | R(x,y), D(y) | 50 | 50 | 40 |
| 6 | R(x,y), D(y), C(x) | 45 | 35 | 25 |
| 7 | R(x,y), C(x), D(y) | 45 | 40 | 25 |

cluster. Which cluster is relevant depends on the query atom for which we compute the cost function and the previously computed bindings. For instance, if we compute the cost of a property atom $R(x, y)$ and we already determined bindings for $x$, we use the property successor cluster $PC_1$. Among the $x$ bindings, we then just check the cost for one binding per cluster and assign the same cost to all other $x$ bindings of the same cluster.

A motivating example showing the difference between static and dynamic ordering and justifying why dynamic ordering can be beneficial in our setting is shown below. Let us assume that a query $q$ consists of the three query atoms: $C(x)$, $R(x, y)$, $D(y)$. Table 1 gives information about the known and possible instances of these atoms within a sequence. In particular, the first column enumerates possible execution sequences $S_i = (at_1, \ldots, at_i)$ for the atoms of $q$. Column 2 (3) gives the number of mappings to known (possible) instances of $at_i$ (i.e., the number of known (possible) instances of $at_i$) that satisfy at the same time the atoms $(at_1, \ldots, at_{i-1})$. Column 4 gives the number of possible instances of $at_i$ from Column 3 that are real instances (that belong to $\Omega_i$). Let us assume that we have 10,000 individuals in our ontology $O$. We will now explain via the example what the formulas described above are doing. We assume that $C_L \leq C_E$ which is always the case since a cache lookup is less expensive than a consistency check. In both techniques (static and dynamic) the atom $R(x, y)$ will be chosen first since it has the least number of possible instances (200) while it has the same (or smaller) number of known instances (200) as the other atoms:

$$\begin{aligned} s(R(x, y), \emptyset) = d(R(x, y), \{\mu_0\}) &= \langle 200 \cdot C_L + 200 \cdot C_E, 200 + P_{IS} \cdot 200\rangle, \\ s(C(x), \emptyset) = d(C(x), \{\mu_0\}) &= \langle 200 \cdot C_L + 350 \cdot C_E, 200 + P_{IS} \cdot 350\rangle, \\ s(D(y), \emptyset) = d(D(y), \{\mu_0\}) &= \langle 700 \cdot C_L + 600 \cdot C_E, 700 + P_{IS} \cdot 600\rangle. \end{aligned}$$

In the case of static ordering, the atom $C(x)$ is chosen after $R(x, y)$ since $C$ has less possible (and known) instances than $D$ (350 versus 600):

$$\begin{aligned} s(C(x), \{x\}) &= \left\langle \frac{200}{10,000} \cdot C_L + \frac{350}{10,000} \cdot C_E, \frac{200 + 350 \cdot P_{IS}}{10,000} \right\rangle, \\ s(D(y), \{y\}) &= \left\langle \frac{700}{10,000} \cdot C_L + \frac{600}{10,000} \cdot C_E, \frac{700 + 600 \cdot P_{IS}}{10,000} \right\rangle. \end{aligned}$$

Hence, the order of evaluation in this case will be $P = (R(x, y), C(x), D(y))$ leading to 200 (row 2) + 150 (row 4) + 40 (row 7) entailment checks. In the dynamic case, after the evaluation of $R(x, y)$, which gives a set of solutions $\Omega_1$, the atom $D(y)$ has fewer known and possible instances (50 known and 50 possible) than the atom $C(x)$ (100 known and 150 possible) and, hence, a lower cost:

$$d(D(y), \Omega_1) = \langle 50 \cdot C_L + 150 \cdot C_L + 50 \cdot C_E, 50 + 0 + 50 \cdot P_{IS} \rangle,$$
$$d(C(x), \Omega_1) = \langle 100 \cdot C_L + 0 \cdot C_L + 150 \cdot C_E, 100 + 0 + 150 \cdot P_{IS} \rangle.$$

Note that applying a solution $\mu \in \Omega_1$ to $D(y)$ ($C(x)$) results in a query atom with a constant in place of $y$ ($x$). For $D(y)$, it is the case that out of the 250 $R$-instances, 200 can be handled with a look-up (50 turn out to be known instances and 150 turn out not to be instances of $D$), while 50 require an entailment check. Similarly, when considering $C(x)$, we need 100 lookups and 150 entailment checks. Note that we assume the worst case in this example, i.e., that all values that $x$ and $y$ take are different. Therefore, the atom $D(y)$ will be chosen next leading to the execution of the query atoms in the order $P = (R(x, y), D(y), C(x))$ and the execution of 200 (row 2) + 50 (row 5) + 35 (row 6) entailment checks.

## 5 Evaluation

We tested our ordering techniques with the Lehigh University Benchmark (LUBM) [4] as a case where no disjunctive information is present and with the more expressive University Ontology Benchmark (UOBM) [8] using the HermiT[5] hypertableau reasoner. All experiments were performed on a Mac OS X Lion machine with a 2.53 GHz Intel Core i7 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. The ontologies and all code required to perform the experiments are available online.[6]

We first used the 14 conjunctive ABox queries provided in LUBM. From these, queries 2, 7, 8, 9 are the most interesting ones in our setting since they contain many atoms and ordering them can have an effect in running time. We tested the queries on LUBM(1,0) and LUBM(2,0) which contain data for one or two universities respectively, starting from index 0. LUBM(1,0) contains 17,174 individuals and LUBM(2,0) contains 38,334 individuals. LUBM(1,0) took 19 s to load and 0.092 s for classification and initialization of known and possible instances of classes and properties. The clustering approach for classes took 1 s and resulted in 16 clusters. The clustering approach for properties lasted 4.9 s and resulted in 17 property successor clusters, 29 property predecessor clusters and 87 property clusters. LUBM(2,0) took 48.5 s to load and 0.136 s for classification and initialization of known and possible instances. The clustering approach for classes took 3.4 s and resulted in 16 clusters. The clustering approach for properties lasted 16.3 s and resulted in 17 property successor clusters, 31 property predecessor clusters and 102 property clusters. Table 2 shows the execution time for each of the four queries for LUBM(1,0) and LUBM(2,0) for four cases: i) when we use the static algorithm (columns 2 and 6), ii) when we use the dynamic algorithm (columns

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

[6] http://code.google.com/p/query-ordering/

**Table 2.** Query answering times in milliseconds for LUBM(1,0) and LUBM(2,0) using i) the static algorithm ii) the dynamic algorithm, iii) 50% random sampling (RSampling), iv) the constructed individual clusters for sampling (CSampling)

| | LUBM(1,0) | | | | LUBM(2,0) | | | |
|---|---|---|---|---|---|---|---|---|
| Query | Static | Dynamic | RSampling | CSampling | Static | Dynamic | RSampling | CSampling |
| ∗2 | 51 | 119 | 390 | 37 | 162 | 442 | 1,036 | 153 |
| 7 | 25 | 29 | 852 | 20 | 70 | 77 | 2,733 | 64 |
| 8 | 485 | 644 | 639 | 551 | 622 | 866 | 631 | 660 |
| ∗9 | 1,099 | 2,935 | 3,021 | 769 | 6,108 | 23,202 | 14,362 | 3,018 |

**Table 3.** Statistics about the constructed plans and chosen orderings and running times in milliseconds for the orderings chosen by Pellet and for the worst constructed plans

| Query | PlansNo | Chosen Plan Order | | | Pellet Plan | Worst Plan |
|---|---|---|---|---|---|---|
| | | Static | Dynamic | Sampling | | |
| 2 | 336 | 2 | 1 | 1 | 51 | 4,930 |
| 7 | 14 | 1 | 1 | 1 | 25 | 7,519 |
| 8 | 56 | 1 | 1 | 1 | 495 | 1,782 |
| 9 | 336 | 173 | 160 | 150 | 1,235 | 5,388 |

3 and 7), iii) when we use random sampling, i.e., taking half of the individuals that are returned (from the evaluation of previous query atoms) in each run, to decide about the next cheapest atom to be evaluated in the dynamic case and iv) using the proposed sampling approach that is based on clusters constructed from individuals in the queried ontology (columns 4 and 8). The queries marked with (*) are the queries where the static and dynamic algorithms result in a different ordering. In queries 7 and 8 we observe an increase in running time when the dynamic technique is used (in comparison to the static) which is especially evident on query 8 of LUBM(2,0), where the number of individuals in the ontology and the intermediate result sizes are larger. Dynamic ordering also behaves worse than static in queries 2 and 9. This happens because, although the dynamic algorithm chooses a better ordering than the static algorithm, the intermediate results (that need to be checked in each iteration to determine the next query atom to be executed) are quite large and hence the cost of iterating over all possible mappings in the dynamic case far outweighs the better ordering that is obtained. We also observe that a random sampling for collecting the ordering statistics in the dynamic case (checking only 50% of individuals in $\Omega_{i-1}$ randomly for detecting the next query atom to be executed) leads to much worse results in most queries than plain static or dynamic ordering. This happens since random sampling often leads to the choice of a worse execution order. The use of the cluster based sampling method performs better than the plain dynamic algorithm in all queries. In queries 2 and 9, the gain we have from the better ordering of the dynamic algorithm is much more evident. This is the case since we use at most one individual from every cluster for the cost functions computation and the number of clusters is much smaller than the number of the otherwise tested individuals in each run.

In order to show the effectiveness of our proposed cost functions we compared the running times of all the valid plans (plans constructed according to Definition 3) with the running time of the plan chosen by our method. In the following we show the results for LUBM(1, 0), but the results for LUBM(2,0) are comparable. In Table 3 we show, for each query, the number of plans that were constructed (column 2), the order of the plan chosen by the static, dynamic, and cluster based sampling methods if we order the valid plans by their execution time (columns 3,4,5; e.g., a value of 2 indicates that the ordering method chose the second best plan), the running time of HermiT for the plan that was created by Pellet[7] (column 6) as well as the running time of the worst constructed plan (column 7). The comparison of our ordering approach with the approach followed by other reasoners that support conjunctive query answering such as Pellet or Racer[8] is not very straightforward. This is the case because Pellet and Racer have many optimizations for instance retrieval [11, 5], which HermiT does not have. Thus, a comparison between the execution times of these reasoners and HermiT would not convey much information about the effectiveness of the proposed query ordering techniques. The idea of comparing only the orderings computed by other reasoners with those computed by our methods is also not very informative since the orderings chosen by different reasoners depend much on the way that queries are evaluated and on the costs of specific tasks in these reasoners and, hence, are reasoner dependent, i.e., an ordering that is good for one reasoner and which leads to an efficient evaluation may not be good for another reasoner. We should note that when we were searching for orderings according to Pellet, we switched off the simplification optimization that Pellet implements regarding the exploitation of domain and range axioms of the queried ontology for reducing the number of query atoms to be evaluated [10]. This has been done in order to better evaluate the difference of the plain ordering obtained by Pellet and HermiT since our cost functions take into account all the query atoms.

We observe that for all queries apart from query 9 the orderings chosen by our algorithms are the (near)optimal ones. For queries 2 and 7, Pellet chooses the same ordering as our algorithms. For query 8, Pellet chooses an ordering which, when evaluated with HermiT, results in higher execution time. For query 9, our algorithms choose plans from about the middle of the order over all the valid plans w.r.t. query execution time, which means that our algorithms do not perform well in this query. This is because of the greedy techniques we have used to find the execution plan which take into account only local information to choose the next query atom to be executed. Interestingly, the use of cluster based sampling has led to the finding of a better ordering, as we can see from the running time in Table 2 and the better ordering of the plan found with cluster based sampling techniques compared to static or plain dynamic ordering (Table 3). The ordering chosen by Pellet for query 9 does also not perform well. We see that, in all queries, the worst running times are many orders of magnitude greater than the running times achieved by our ordering algorithms. In general, we observe that in LUBM static techniques are adequate and the use of dynamic ordering does not improve the execution time much compared to static ordering.

Unlike LUBM, the UOBM ontology contains disjunctions and the reasoner makes also nondeterministic derivations. In order to reduce the reasoning time, we removed the nominals and only used the first three departments containing 6,409 individuals. The

[7] http://clarkparsia.com/pellet/

[8] http://www.racer-systems.com

**Table 4.** Query answering times in seconds for UOBM (1 university, 3 departments) and statistics

| Query | Static | Dynamic | CSampling | PlansNo | Chosen Plan Order | | | Pellet Plan | Worst Plan |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | Static | Dynamic | Sampling | | |
| 4 | 13.35 | 13.40 | 13.41 | 14 | 1 | 1 | 1 | 13.40 | 271.56 |
| 9 | 186.30 | 188.58 | 185.40 | 8 | 1 | 1 | 1 | 636.91 | 636.91 |
| 11 | 0.98 | 0.84 | 1.67 | 30 | 1 | 1 | 1 | 0.98 | > 30 min |
| 12 | 0.01 | 0.01 | 0.01 | 4 | 1 | 1 | 1 | 0.01 | > 30 min |
| 14 | 94.61 | 90.60 | 93.40 | 14 | 2 | 1 | 1 | > 30 min | > 30 min |
| $q_1$ | 191.07 | 98.24 | 100.25 | 6 | 2 | 1 | 1 | > 30 min | > 30 min |
| $q_2$ | 47.04 | 22.20 | 22.51 | 6 | 2 | 1 | 1 | 22.2 | > 30 min |

resulting ontology took 16 s to load and 0.1 s to classify and initialize the known and possible instances. The clustering approach for classes took 1.6 s and resulted in 356 clusters. The clustering approach for properties lasted 6.3 s and resulted in 451 property successor clusters, 390 property predecessor clusters and 4,270 property clusters. We ran our static and dynamic algorithms on queries 4, 9, 11, 12 and 14 provided in UOBM, which are the most interesting ones because they consist of many atoms. Most of these queries contain one atom with possible instances. As we see from Table 4, static and dynamic ordering show similar performance in queries 4, 9, 11 and 12. Since the available statistics in this case are quite accurate, both methods find the optimal plans and the intermediate result set sizes are small. For both ordering methods, atoms with possible instances for these queries are executed last. In query 14, the dynamic algorithm finds a better ordering which results in improved performance. The effect that the cluster based sampling technique has on the running time is not as obvious as in the case of LUBM. This happens because in the current experiment the intermediate result sizes are not very large and, most importantly, because the gain obtained due to sampling is in the order of milliseconds whereas the total query answering times are in the order of seconds obscuring the small improvement in running time due to sampling. In all queries the orderings that are created by Pellet result in the same or worse running times than the orderings created by our algorithms.

In order to illustrate when dynamic ordering performs better than static, we also created the two custom queries:

$q_1$ = { isAdvisedBy(x,y), GraduateStudent(x), Woman(y) }
$q_2$ = { SportsFan(x), GraduateStudent(x), Woman(x) }

In both queries, $P$[GraduateStudent], $P$[Woman] and $P$[isAdvisedBy] are non-empty, i.e., the query concepts and properties have possible instances. The running times for dynamic ordering are smaller since the more accurate statistics result in a smaller number of possible instances that have to be checked during query execution. In particular, for the static ordering, 151 and 41 possible instances have to be checked in query $q_1$ and $q_2$, respectively, compared to only 77 and 23 for the dynamic ordering. Moreover, the intermediate results are generally smaller in dynamic ordering than in static leading to a significant reduction in the running time of the queries. Interestingly, query $q_2$ could not be answered within the time limit of 30 minutes when we transformed the three query classes into a conjunction, i.e., when we asked for instances of the intersection of the three classes. This is because for complex classes the reasoner can no longer use the information about known and possible instances and falls back to a more naive way of computing the class instances. Again, for the same reasons as before, the sampling techniques have no apparent effect on the running time of these queries.

## 6 Related work

The problem of finding good orderings for the atoms of a query issued over an ontology has already been preliminarily studied [10, 5].

Similarly to our work, Sirin et al. exploit reasoning techniques and information provided by reasoner models to create statistics about the cost and the result size of query atom evaluations within execution plans. A difference is that they use cached models for cheaply finding obvious class and property (non-)instances, whereas in our case we do not cache any model or model parts. Instead we process the pre-model constructed for the initial ontology consistency check and extract the known and possible instances of classes and properties from it. We subsequently use this information to create and update the query atom statistics. Moreover, Sirin et al. compare the costs of complete execution plans —after heuristically reducing the huge number of possible complete plans — and choose the one that is most promising before the beginning of query execution. This is different from our cheap greedy algorithm that finds, at each iteration, the next most promising query atom. Our experimental study shows that this is equally effective as the investigation of all possible execution orders. Moreover, in our work we have additionally used dynamic ordering combined with clustering techniques, apart from static ones, and have shown that these techniques lead to better performance particularly in ontologies that contain disjunctions and do now allow for purely deterministic reasoning.

Haarslev et al. discuss by means of an example the ordering criteria they use to find efficient query execution plans. In particular, they use traditional database cost based optimization techniques, which means that they take into account only the cardinality of class and property atoms to decide about the most promising ordering. As previously discussed, this can be inadequate especially for ontologies with disjunctive information.

## 7 Conclusions

In the current paper, we presented a method for ordering the atoms of a conjunctive instance query that is issued over an OWL ontology. We proposed a method for the definition of cost formulas that are based on information extracted from models of a reasoner (in our case HermiT). We have devised two algorithms, a static and a dynamic one, for finding a good order and show through an experimental study that static techniques are quite adequate for ontologies in which reasoning is deterministic. When reasoning is nondeterministic, however, dynamic techniques often perform better. The use of cluster based sampling techniques can improve the performance of the dynamic algorithm when the intermediate result sizes of queries are sufficiently large, whereas random sampling was not beneficial and often led to suboptimal query execution plans.

Future work will include the definition of additional cost measures and sampling criteria for ordering query atoms and the evaluation of our ordering techniques on a broader set of ontologies and queries.

**Acknowledgements** This work was done within the Transregional Collaborative Research Centre SFB/TRR 62 "Companion-Technology for Cognitive Technical Systems" funded by the German Research Foundation (DFG).

## References

1. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, second edn. (2007)
2. Glimm, B., Horrocks, I., Lutz, C., Sattler, U.: Conjunctive query answering for the description logic SHIQ. Journal of Artificial Intelligence Research 31, 151–198 (2008)
3. Glimm, B., Horrocks, I., Motik, B., Shearer, R., Stoilos, G.: A novel approach to ontology classification. Journal of Web Semantics: Science, Services and Agents on the World Wide Web, Accepted (2012)
4. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Semantics 3(2-3), 158–182 (2005)
5. Haarslev, V., Möller, R.: On the scalability of description logic instance retrieval. J. Autom. Reason. 41(2), 99–142 (Aug 2008)
6. Kazakov, Y.: $\mathcal{RIQ}$ and $\mathcal{SROIQ}$ are harder than $\mathcal{SHOIQ}$. In: Brewka, G., Lang, J. (eds.) Proc. 11th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'08). pp. 274–284. AAAI Press (2008)
7. Kollia, I., Glimm, B., Horrocks, I.: SPARQL query answering over OWL ontologies. In: Proceedings of the 8th Extended Semantic Web Conference (ESWC 2011). pp. 382–396. Lecture Notes in Computer Science, Springer-Verlag (2011)
8. Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y., Liu, S.: Towards a complete OWL ontology benchmark. In: The Semantic Web: Research and Applications, chap. 12, pp. 125–139. Lecture Notes in Computer Science, Springer (2006)
9. Motik, B., Shearer, R., Horrocks, I.: Hypertableau reasoning for description logics. Journal of Artificial Intelligence Research 36, 165–228 (2009)
10. Sirin, E., Parsia, B.: Optimizations for answering conjunctive ABox queries: First results. In: Proc. of the Int. Description Logics Workshop DL (2006)
11. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. of Web Semantics 5(2), 51–53 (2007)
12. Steinbrunn, M., Moerkotte, G., Kemper, A.: Heuristic and randomized optimization for the join ordering problem. VLDB Journal 6, 191–208 (1997)
13. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th international conference on World Wide Web. pp. 595–604. WWW '08, ACM, New York, NY, USA (2008)
14. Tsarkov, D., Horrocks, I., Patel-Schneider, P.F.: Optimizing terminological reasoning for expressive description logics. J. Autom. Reasoning 39(3), 277–316 (2007)