Cost Based Query Ordering over OWL Ontologies

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

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

1 Introduction

Query answering—the computation of answers to users' queries w.r.t. ontologies and data—is an important task provided by many description logic (DL) reasoners. Although much effort has been spent on optimizing the 'reasoning' part of query answering, i.e., the extraction of the individuals that satisfy a concept or role atom, less attention has been given to optimizing the actual query answering part when ontologies in expressive languages are used.

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 [9, 10]. 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 ontologies, the formulas should be extended to take into account another important cost component, i.e., the cost of executing specific reasoner tasks such as entailment checks or instance retrievals. It is much more difficult to estimate this cost precisely before query evaluation as this cost varies and takes values from a wide range. For example, $\mathcal{SROIQ}$ has a worst case complexity of 2-NExpTime [4] and typical implementations are not worst case optimal. The hypertableau satisfiability checking algorithm for $\mathcal{SROIQ}$ that we use in this paper has a worst-case complexity of 3-NExpTime in the size of the ontology [7, 4].[3] Instead, a subset of possible mappings for the variables of a query can be sampled and, based on the statistics extracted from these samples, the most efficient join ordering can be estimated. This, however, requires that the samples are selected according to an ontology based criterion and not randomly. Preliminary efforts for finding good join ordering in knowledge bases have already been made [8].

In this paper we address the issue of query atom ordering, which constitutes an optimization task, for conjunctions of instance queries issued over ontologies with expressivity up to $\mathcal{SROIQ}$. The optimization goal is to find the execution plan (an order for query atoms) which leads to the most efficient execution of the query. This involves

[3] The 2-NExpTime result for $\mathcal{SHOIQ}$+ increases to 3-NExpTime when adding role chains [4]

the minimization of 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 query atoms within an execution plan. This cost function is based on heuristics and summaries for statistics about the data which are extracted from a DL reasoner model. We explore static and dynamic algorithms 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 an introduction to conjunctive instance queries.

It is known that checking whether an individual s_0 (pair of individuals $\langle s_0, s_1\rangle$) is an instance (are instances) of a concept C (role R) w.r.t. an ontology $\mathcal{O}$ is equivalent to checking whether $\mathcal{O} \cup \{\neg C(c_0)\}$ ($\mathcal{O} \cup \{(\forall R.\neg\{s_1\})(s_0)\}$) is inconsistent w.r.t. $\mathcal{O}$. In order to perform this action, most DL reasoners use a model construction calculus such as tableau or hypertableau. In the remainder, we focus on the hypertableau calculus [7], 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 concept and role 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 $\mathcal{A}_n$ does not contain a clash, then $\mathcal{O}$ is consistent and $\mathcal{A}_n$ is called a *pre-model* of $\mathcal{O}$. Otherwise $\mathcal{O}$ is inconsisent. Each assertion in a set of assertions $\mathcal{A}_i$ 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 concepts 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 roles through which individual s_0 is (non)deterministically related to individual s_1.

Definition 1. *Let $S = (N_C, N_R, N_I)$ be a signature of an ontology $\mathcal{O}$, N_V a countably infinite set of variables disjoint from N_C, N_R and N_I and $S_\mathcal{O} = (C_\mathcal{O}, R_\mathcal{O}, I_\mathcal{O})$ the restriction of S to terms that occur in $\mathcal{O}$. A* term *t is an element from $N_V \cup N_I$. Let $C \in C_\mathcal{O}$ be a concept, $r \in R_\mathcal{O}$ a role, and $t, t' \in I_\mathcal{O} \cup N_V$ terms. An* atom *is an expression*

$C(t)$ or $r(t,t')$ and we refer to these types of atoms as concept *and* role atoms*, respectively. A* query *q is a non-empty set of atoms. We use $Vars(q)$ to denote the set of variables occurring in q, $Inds(q)$ to denote the set of individual names occurring in q, and $Terms(q) = Vars(q) \cup Inds(q)$ for the set of terms in q. We use $|q|$ to denote the number of atoms in q.*

Let $q = \{at_1, \ldots, at_n\}$ be a query. A mapping μ 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 Γ_q of all possible mappings for q is defined as $\Gamma_q := \{\mu \mid \mu \text{ is a mapping for } q\}$. A* solution mapping μ for q over O *is a mapping such that $O \models C(\mu(t))$ for each concept atom $C(t) \in q$ and $O \models r(\mu(t), \mu(t'))$ for each role atom $r(t,t') \in q$.*

According to the above definition, we deal with conjunctive queries with only distinguished variables. Without loss of generality, we assume that queries are connected. In case they are not, the connected components of a query can be evaluated independently and the results of these evaluations can be combined in the end [1].

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.

As has been mentioned in Section 2, if an ontology is consistent and a pre-model is constructed, individuals are assingned labels in this pre-model. These labels can provide us with information about the concepts the individuals belong to or the roles in which they participate. We exploit this information similarly as was suggested for determining known (non-)subsumers for classes during classification [2]. In the hypertableau calculus, the following two properties hold for each ontology O and each constructed pre-model A_n for O:

(P1) for each atomic concept C (role R), each individual s_0 (each 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 $O \models C(s_0)$ ($O \models R(s_1, s_2)$).

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

We use these properties to extract information from the pre-model of a satisfiable ontology 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 atomic concepts of O. The known instances of a concept C ($K[C]$) are the individuals that can be safely considered instances of the concept according to the pre-model, i.e., the individuals of the assertions referring to concept C that have been derived deterministically. The possible instances of a concept C ($P[C]$) are the individuals of the assertions referring to C that have been derived nondeterministically. These individuals require costly consistency checks in order to decide whether they are real instances of the respective concept.

Algorithm 1 initializeKnownAndPossibleConceptInstances

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

The procedure to find the known and possible instances of simple and complex (transitive roles or roles having a transitive subrole) roles or, given an individual, the known and possible role successors or predecessors, can be defined similarly. In the case of complex roles, however, before the *buildModelFor* procedure is applied, $\mathcal{O}$ is expanded with additional axioms that capture the semantics of the transitive relations since (hyper)tableau reasoners typically do not deal with transitivity directly [7]. In particular, for each individual *ind* and each complex role p, the new concepts C^p_{ind} and C_{ind} are created and the axioms $C_{ind}(ind)$ and $C_{ind} \sqsubseteq \forall p.C^p_{ind}$ are added to $\mathcal{O}$. Intuitively, the consequent application of the transitivity encoding [7] produces axioms $C^p_{ind}(s)$ that propagate to each individual s that is reachable from *ind* via a p-chain. The known and possible p-successors for *ind* can then be determined from concept assertions $C^p_{ind}(s)$ in the pre-model.

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 subconcept 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 concept 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 general concept inclusion axioms (GCIs), 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 concept 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 concept C in $\mathcal{O}$ we need to add to P[C] the individuals in $\mathcal{O}$ that do not include the concept C in their label.

Even though the proposed technique for determining known and possible instances of concepts and roles can be used in the same way with both tableau and hypertableau reasoners, the effect that it will have when tableau algorithms are used is less intense. This happens because tableau algorithms often introduce more nondeterminism than hypertableau. In particular, in tableau algorithms a disjunction is added to each individual for each GCI in $\mathcal{O}$ and, when optimizations such as lazy unfolding are used, these compromise property (P2) and we have to use the weaker property (P3) or even consider all concepts that do not occur in the label of an individual as possible types, which results in less accurate statistics.

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 q for a query $\{at_1, \ldots, at_n\}$, G_q for the according query join graph and Ω_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: an estimate for the reasoning costs and 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

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

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. Dynamic ordering is a costly but accurate procedure. Sampling techniques can be used so that not all mappings are used for the update of the cost function but only a subset of them. In Section 5 we show that random sampling is not adequate and a more sophisticated sampling criterion is needed. However, the definition of such criterion is out of the scope of the current paper. 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 $s\colon q \times 2^{Vars(q)} \to \mathbb{R} \times \mathbb{R}$ ($d\colon q \times 2^{\Gamma_q} \to \mathbb{R} \times \mathbb{R}$). The two costs are combined to yield a* static ordering $\preceq_s$ *(a* 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 we follow to find the solution mappings Ω_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 Ω_{i-1}, in order to find the solution mappings Ω_i of P_i, we use instance retrieval tasks of reasoners to extend the mappings in Ω_{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 we are using for the evaluation of an execution plan together with optimizations can be found in our previous work [5].

We now define the cost functions s and d more precisely, which estimate the cost of the required reasoner operations and the estimated result output size of evaluating a query atom. The intuition behind the estimated value of the reasoner operation costs (the functions' first component) 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 (the functions' second component) takes into account the number of known and possible instances and the probability that possible instances are actual instances. The static cost function has more cases since for atoms we might only know that their variables are bound, without knowing to which individuals they are bound to. The functions depend on several factors:

- $K[C]$ ($K[R]$) and $P[C]$ ($P[R]$) for known and possible instances of a concept C (a role R) from Section 3
- $\mathsf{sucK}[R] := \{i \mid \exists j.\langle i, j\rangle \in K[R]\}$ ($\mathsf{preK}[R] := \{i \mid \exists j.\langle j, i\rangle \in K[R]\}$) for the individuals with known successors (predecessors) for a role R. We define analogous functions $\mathsf{sucP}[R]$ and $\mathsf{preP}[R]$ for the individuals with possible successors and predecessors for a role R
- $\mathsf{sucK}[R, a] := \{i \mid \langle a, i\rangle \in K[R]\}$ ($\mathsf{preK}[R, a] := \{i \mid \langle i, a\rangle \in K[R]\}$) for known R-successors (R-predecessors) of an individual a. We define analogous functions $\mathsf{sucP}[R, a]$ and $\mathsf{preP}[R, a]$ for possible R-successors and R-predecessors of an individual a
- C_L for the cost of a cache lookup in the reasoner's internal structures
- C_E for the cost of an entailment check
- P_{IS} for the possible instance success, i.e, an estimate for percentage of possible instances that are actual instances

The values C_L, C_E and P_{IS} are determined experimentally. The time needed for a cache lookup is much less than the time needed for an entailment check with the difference between the two depending on the ontology and even within an ontology on the queried concept (role). The two costs (C_L and C_E) were determined by taking the average time of the previous performed checks (lookup or entailment). In the case of C_E, we multiply this number with the depth of the concept (role) hierarchy. The depth of the concept (role) hierarchy should be taken into account for the estimation of C_E since we only store the direct types of each individual (roles in which each individual participates). In order to find the instances of a concept (role), we may need to check all its subconcepts (subroles) that contain possible instances. 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.

In the following, we use a, b for individual names and x, y for variables. We first define the static cost function s, which takes a pair $\langle at(\vec{t}), VarsB\rangle$ as input, where $at(\vec{t})$ is a query atom and $VarsB$ is the set of (bound) variables from $\mathrm{Vars}(at(\vec{t}))$, and returns a pair of real numbers as follows:

- for $\langle at(\vec{t}), VarsB\rangle \in \{\langle C(x), \emptyset\rangle, \langle R(x, y), \emptyset\rangle\}$

$$\langle |K[at]| \cdot C_L + |P[at]| \cdot C_E, |K[at]| + P_{IS} \cdot |P[at]|\rangle \tag{1}$$

- for $\langle at(\vec{t}), VarsB\rangle \in \{\langle R(a, x), \emptyset\rangle\}$

$$\langle |\mathsf{sucK}[at, a]| \cdot C_L + |\mathsf{sucP}[at, a]| \cdot C_E, |\mathsf{sucK}[at, a]| + P_{IS} \cdot |\mathsf{sucP}[at, a]|\rangle \tag{2}$$

- for $\langle at(\vec{t}), \mathrm{VarsB}\rangle \in \{\langle R(x, a), \emptyset\rangle\}$ $\mathsf{preK}[a]$nd $\mathsf{preP}[i]$nstead of $\mathsf{sucK}[a]$nd $\mathsf{sucP}[i]$n

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

– for $\langle at(\vec{t}), VarsB\rangle \in \{\langle C(a), \emptyset\rangle, \langle R(a,b), \emptyset\rangle\}$

$$\begin{array}{l} \langle C_L, 1\rangle \text{ if } \vec{t} \in K[at] \\ \langle C_E, P_{IS}\rangle \text{ if } \vec{t} \in P[at] \\ \langle C_L, 0\rangle \text{ otherwise} \end{array} \quad (3)$$

– for $\langle at(\vec{t}), VarsB\rangle \in \{\langle C(x), \{x\}\rangle, \langle R(x,y), \{x,y\}\rangle, \langle R(a,x), \{x\}\rangle, \langle R(x,a), \{x\}\rangle\}$

$$\left\langle \frac{|K[at]|}{|I_O|} \cdot C_L + \frac{|P[at]|}{|I_O|} \cdot C_E, \frac{|K[at]| + P_{IS} \cdot |P[at]|}{|I_O| \cdot |I_O|} \right\rangle \quad (4)$$

– for $\langle at(\vec{t}), VarsB\rangle = \langle R(x,y), \{x\}\rangle$

$$\left\langle \frac{|K[at]|}{|\mathsf{sucK}[at]|} \cdot C_L + \frac{|P[at]|}{|\mathsf{sucP}[at]|} \cdot C_E, \frac{|K[at]|}{|\mathsf{sucK}[at]|} + \frac{|P[at]|}{|\mathsf{sucP}[at]|} \cdot P_{IS} \right\rangle \quad (5)$$

– for $\langle at(\vec{t}), \text{VarsB}\rangle = \langle R(x,y), \{y\}\rangle$ we use $\mathsf{preK}[a]$nd $\mathsf{preP}[i]$nstead of $\mathsf{sucK}[a]$nd $\mathsf{sucP}[i]$n 5

The dynamic cost function d is based on the static function s, but only applies to cases (1), (4) and (3). The function takes a pair $\langle at(\vec{t}), \Omega\rangle$ as input, where $at(\vec{t})$ is a query atom and Ω 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:

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

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 Ω_i). Let us assume that we have 10,000 individuals in our ontology O. We will now explain 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) with the other atoms ($s(R(x, y), \emptyset) = d(R(x, y), \{\mu_0\}) = \langle 200 \cdot C_L + 200 \cdot C_E, 200 + P_{IS} \cdot 200\rangle$, $s(C(x), \emptyset) = d(C(x), \{\mu_0\}) = \langle 200 \cdot C_L + 350 \cdot C_E, 200 + P_{IS} \cdot 350\rangle$, $s(D(y), \emptyset) = d(D(y), \{\mu_0\}) = \langle 700 \cdot C_L + 600 \cdot C_E, 700 + P_{IS} \cdot 600\rangle$). Then, in the case of static ordering, after $R(x, y)$, the atom $C(x)$ is chosen since C has less possible (and known) instances than D (350 versus 600). Indeed, $s(C(x), \{x\}) = \langle \frac{200}{10{,}000} \cdot C_L + \frac{350}{10{,}000} \cdot C_E, \frac{200+350 \cdot P_{IS}}{10^8}\rangle$, $s(D(y), \{y\}) = \langle \frac{700}{10{,}000} \cdot C_L + \frac{600}{10{,}000} \cdot C_E, \frac{700+600 \times P_{IS}}{10^8}\rangle$. 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 Ω_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. Indeed, $d(D(y), \Omega_1) = \langle 50 \cdot C_L + 150 \cdot C_L + 50 \cdot C_E, 50 + 0 + 50 \cdot P_{IS}\rangle$, $d(C(x), \Omega_1) = \langle 100 \cdot C_L + 0 \cdot C_L + 150 \cdot C_E, 100 + 0 + 150 \cdot P_{IS}\rangle$. Therefore, 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) [3] as a case where no disjunctive information is present and with the more expressive University Ontology Benchmark (UOBM) [6] using the HermiT[5] hypertableau reasoner. All experiments were performed on a Windows 7 machine with a double core 2.53 GHz Intel x86 64 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. 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 16,283 individuals and LUBM(2,0) contains 38,334 individuals. LUBM(1,0) took 3.8 s to load and 22.7 s for classification and initialization of known and possible instances of concepts and roles. LUBM(2,0) took 15.8 s to load and 146 s for classification and initialization of known and possible instances. Table 2 shows the execution time for each of the four queries for LUBM(1,0) and LUBM(2,0). The queries marked with (*) are the queries where the static and dynamic algorithms result in the same ordering. In these queries we observe an increase in

[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) and in seconds for UOBM (1 university, dep 0-2) using i) the static algorithm ii) the dynamic algorithm and iii) (only for LUBM) 50% sampling

	LUMB(1, 0)			LUBM(2,0)		
Query	Static	Dynamic	Sampling	Static	Dynamic	Sampling
2	110	203	721	386	917	12,767
∗7	47	66	1,743	138	127	9,653
∗8	686	873	867	2,434	4,237	2,488
9	1,670	13,056	13,372	17,960	91,787	94,087

	UOBM	
Query	Static	Dynamic
4	23.81	24.08
9	701.24	690.51
11	2.22	2.07
12	0.13	0.16
14	212.28	215.56
q_1	668.82	347.25
q_2	179.92	85.65

running time when the dynamic technique is used (in comparison to the static) which is especially evident on LUBM(2,0) Query 8, 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 of individuals for collecting the ordering statistics in the dynamic case (checking only 50% of individuals in Ω_{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.

From the nondeterministic UOBM ontology we removed the nominals and only used the first three departments containing 6,409 individuals. The ontology took 6.4 s to load and 30.3 s to classify and initiliaze the known and possible instances. 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. Static and dynamic ordering show similar performance because intermediate result set sizes are small and the available statistics in this case are quite accurate and result in the same ordering for both methods. For both ordering methods, atoms with possible instances for these queries are executed last. In order to illustrate when dynamic ordering performs better than static, we also created the two custom queries:

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

In both queries, P[GraduateStudent], P[Woman] and P[isAdvisedBy] are non-empty. The running times for dynamic ordering is 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. All of the presented queries could not be answered in the time limit of 30 minutes in case no ordering algorithm was used.

6 Conclusions

In the current paper, we have dealt with the definition of cost formulas that are based on information extracted from reasoners' models for ordering the atoms of a conjunctive instance query that is issued over an OWL ontology. 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 deterministic ontologies, however, when disjunctive knowledge is present, dynamic techniques often perform better. The proposed query ordering costs can be used with either tableau or hypertableau reasoners, however, in the case of tableau reasoners they can be less accurate. Future work will include the definition of additional cost measures and the use of appropriate criteria for sampling the individuals and use only these samples for extracting the costs for dynamic ordering.

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. 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)
2. 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)
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. 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)
5. 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)
6. 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)
7. Motik, B., Shearer, R., Horrocks, I.: Hypertableau reasoning for description logics. Journal of Artificial Intelligence Research 36, 165–228 (2009)
8. Sirin, E., Parsia, B.: Optimizations for answering conjunctive ABox queries: First results. In: Proc. of the Int. Description Logics Workshop DL (2006)
9. Steinbrunn, M., Moerkotte, G., Kemper, A.: Heuristic and randomized optimization for the join ordering problem. VLDB Journal 6, 191–208 (1997)
10. 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)