On the Scalability of Description Logic Instance Retrieval

Ralf Möller[1], Volker Haarslev[2], and Michael Wessel[1]

[1] Hamburg University of Technology
[2] Concordia University, Montreal

Abstract. Although description logic systems can adequately be used for representing and reasoning about incomplete information (e.g., for John we know he is French or Italian), in practical applications it can be assumed that (only) for some tasks the expressivity of description logics really comes into play whereas for building complete applications, it is often necessary to effectively solve instance retrieval problems with respect to largely deterministic knowledge. In this paper we present and analyze the main results we have found about how to contribute to this kind of scalability problem. We assume familiarity with description logics in general and tableau provers in particular.

1 Introduction

Although description logics (DLs) are becoming more and more expressive (e.g., [14]), our experience has been that it is only for some tasks that the expressivity of description logics really comes into play; for many applications, it is necessary to be able to deal with largely deterministic knowledge very effectively. Thus, in practice, description logic systems offering high expressivity must also be able to handle large bulks of data descriptions (Aboxes with concepts and role assertions) which are largely deterministic. Users expect that DL systems scale w.r.t. these practical needs. In our view there are two kinds of scalability problems: scalability w.r.t. large sets of data descriptions (data description scalability) and scalability w.r.t. high expressivity, which might only be important for small parts of the data descriptions (expressivity scalability).

In the literature, the data description scalability problem has been tackled from different perspectives. We see two main approaches, the layered approach and the integrated approach. In the layered approach the goal is to use databases for storing and accessing data, and exploit description logic ontologies for convenient query formulation. The main idea is to support ontology-based query translation to relational query languages (SQL, datalog). See, e.g., [21,9] (DLDB), [3] (Instance Store), or [4] (DL-Lite). We notice that these approaches are only applicable if reduced expressivity does not matter. Despite the most appealing argument of reusing database technology (in particular services for persistent data), at the current state of the art it is not clear how expressivity can be increased to, e.g., $\mathcal{SHIQ}$ without losing the applicability of database transformation approaches. Hence, while data description scalability is achieved, it is

C. Freksa, M. Kohlhase, and K. Schill (Eds.): KI 2006, LNAI 4314, pp. 188–201, 2007.

not clear how to extend these approaches to achieve expressivity scalability (at least for some parts of the data descriptions).

Tableau-based DL systems are now widely used in practical applications because these systems are quite successful w.r.t. the expressivity scalability problem. Therefore, for investigating solutions to both problems, the expressivity and the data description scalability problem, we pursue the integrated approach that considers query answering with a tableau-based description logic system augmented with new techniques inspired from database systems. For the time being we ignore the problems associated with persistency and investigate specific knowledge bases (see below).

The contribution presents and analyzes the main results we have found about how to start solving the scalability problem with tableau-based prover systems given large sets of data descriptions for a large number of individuals. Note that we do not discuss query answering speed of a particular system but investigate the effect of optimization techniques that could be exploited by any (tableau-based) DL inference system that already exists or might be built. Since DLs are very popular now, and tableau-based systems have been extensively studied in the literature (see [2] for references), we assume the reader is familiar with DLs in general and tableau-based decision procedures in particular (see, e.g., [1]).

2 Lehigh University Benchmark

In order to investigate the data description scalability problem, we use the Lehigh University BenchMark (LUBM, [8,9]). LUBM queries are conjunctive queries referencing concept, role, and individual names from the Tbox. A query language tailored to description logic applications that can express these queries is described in [20].[1] The language is called nRQL and supports a restricted form of conjunctive queries (variables are only bound to individuals mentioned in the Abox and not to "anonymous" individuals that denote objects from the domain that provably must exist). Although some work on standard conjunctive queries is published [5,17,7], to the best of the authors' knowledge, (efficient) algorithms for answering full conjunctive queries for expressive description logics such as $\mathcal{SHIQ}$ [16] are not known.

Below, LUBM queries 9 and 12 are shown in order to illustrate LUBM queries – note that $'www.University0.edu'$ is an individual and *subOrganizationOf* is a transitive role. Please refer to [8,9] for more information about the LUBM queries.

$$Q9 : ans(x, y, z) \leftarrow Student(x), Faculty(y), Course(z), \\ advisor(x, y), takesCourse(x, z), teacherOf(y, z)$$

$$Q12 : ans(x, y) \leftarrow Chair(x), Department(y), memberOf(x, y), \\ subOrganizationOf(y, 'www.University0.edu')$$

[1] In the notation for queries used in this paper we assume that different variables may have the same bindings.

In order to investigate the data description scalability problem, we used a TBox for LUBM with inverse and transitive roles as well as domain and range restrictions but no number restrictions, value restrictions, or disjunctions (after GCI absorption). Among other axioms, the LUBM TBox contains axioms that express necessary *and* sufficient conditions for some concept names. For instance, there is an axiom $Chair \doteq Person \sqcap \exists headOf.Department$. For evaluating optimization techniques for query answering we consider runtimes for a whole query set (queries 1 to 14 in the LUBM case).

3 Optimization Techniques

If the queries mentioned in the previous section are answered in a naive way by evaluating subqueries in the sequence of syntactic notation, acceptable answering times can hardly be achieved. Determining all bindings for a variable (with a so-called generator) is much more costly than verifying a particular binding (with a tester). Treating the one-place predicates *Student*, *Faculty*, and *Course* as generators for bindings for corresponding variables results in combinatorial explosion (cross product computation). Optimization techniques are required that provide for efficient query answering in the average case.

3.1 Query Optimization

The optimization techniques that we investigated are inspired by database join optimizations, and exploit the fact that there are few *Faculties* but many *Students* in the data descriptions. For instance, in case of query Q9 from LUBM, the idea is to use *Faculty* as a generator for bindings for y and then generate the bindings for z following the role $teacherOf$. The heuristics applied here is that the average cardinality of a set of role fillers is rather small. For the given z bindings we apply the predicate *Course* as a tester (rather than as a generator as in the naive approach). Given the remaining bindings for z, bindings for x can be established via the inverse of *takesCourse*. These x bindings are then filtered with the tester *Student*.

If z was not mentioned in the head, i.e., in set of variables for which bindings are to be computed, and the tester *Course* was not used, there would be no need to generate bindings for z at all. One could just check for the existence of a *takesCourse* role filler for bindings w.r.t. x.

In the second example, query Q12, the constant $'www.University0.edu'$ is mentioned. Starting from this individual the inverse of *subOrganizationOf* is applied as a generator for bindings for y which are filtered with the tester *Department*. With the inverse of $memberOf$, bindings for x are computed which are then filtered with *Chair*. Since for the concept *Chair* sufficient conditions are declared in the TBox, instance retrieval reasoning is required if *Chair* is a generator. Thus, it is advantageous that *Chair* is applied as a tester (and only instance tests are performed).

For efficiently answering queries, a query execution plan is determined by a cost-based optimization component (c.f., [6, p. 787ff.]) which orders query atoms

such that queries can be answered effectively. For computing a total order relation on query atoms with respect to a given set of data descriptions (assertions in an ABox), we need information about the number of instances of concept and role names. An estimate for this information can be computed in a preprocessing step by considering given data descriptions, or could be obtained by examining the result set of previously answered queries. We assume that ABox realization is too costly (takes about 6 minutes for LUBM with one university, excluding the initial Abox consistency test), so this alternative is ruled out.

3.2 Indexing by Exploiting Told and Taxonomical Information

In many practical applications that we encountered, data descriptions often directly indicate (some of the) concept names of which an individual is an instance. Therefore, in a preprocessing step, it is useful to compute an index that maps concept names to sets of individuals which are their instances. In a practical implementation this index might be realized with some form of hashtable.

Classifying the TBox yields the set of ancestors for each concept name, and if an individual i is an instance of a concept name A due to explicit data descriptions, it is also an instance of the ancestors of A. This information can be made accessible by an index that maps concept names to instances. The index is organized in such a way that retrieving the instances of a concept name A, or one of its ancestors, requires (almost) constant time. The index is particularly useful to provide bindings for variables if, despite all optimization attempts for deriving query execution plans, concept names must be used as generators. In addition, the index is used to estimate the cardinality of concept extensions. The estimates are used to compute an order relation for query atoms. The smaller the cardinality of a concept or a set of role fillers is assumed to be, the more priority is given to the query atom. Optimizing LUBM query $Q9$ with the techniques discussed above yields the following query execution plan (denoted as a query, substeps to be read from left to right).

$$Q9' : ans(x, y, z) \leftarrow Faculty(y), teacherOf(y, z), Course(z), \\ advisor^{-1}(y, x), Student(x), takesCourse(x, z)$$

Using this kind of rewriting, queries can be answered much more efficiently.

If the TBox contains only GCIs of the form $A \sqsubseteq A_1 \sqcap \ldots \sqcap A_n$, i.e., if the TBox forms a hierarchy, the index-based retrieval discussed in this section is complete (see [3]). However, this is not the case for LUBM. In LUBM, besides domain and range restrictions, axioms are also of the form $A \doteq A_1 \sqcap A_2 \sqcap \ldots \sqcap A_k \sqcap \exists R_1.B_1 \sqcap \ldots \sqcap \exists R_m.B_m$ (actually, $m = 1$). If sufficient conditions with exists restrictions are specified as in the case of $Chair$, optimization is much more complex. In LUBM data descriptions, no individual is explicitly declared as a $Chair$ and, therefore, reasoning is required, which is known to be rather costly. If $Chair$ is used as a generator and not as a tester such as in the simple query $ans(x) \leftarrow Chair(x)$, optimization is even more important. The idea to optimize instance retrieval is to detect an additional number of obvious instances using

further incomplete tests, and, in addition, to determine obvious non-instances. We first present the latter technique and continue with the former afterwards.

3.3 Obvious Non-instances: Exploiting Information from One Completion

The detection of "obvious" non-instances of a given concept C can be implemented using a model merging operator defined for so-called individual pseudo models (aka pmodels) as defined in [10]. Since these techniques have already been published, we just sketch the main idea here for the sake of completeness. The central idea is to compute a pmodel from a completion that is derived by the tableau prover.

For instance, in the DL $\mathcal{ALC}$ a pseudo model for an individual i mentioned in a consistent initial A-box A w.r.t. a Tbox T is defined as follows. Since A is consistent, there exists a set of completions $\mathcal{C}$ of A. Let $A' \in \mathcal{C}$. An *individual pseudo model* M for an individual i in A is defined as the tuple $\langle M^D, M^{\neg D}, M^{\exists}, M^{\forall} \rangle$ w.r.t. A' and A using the following definition.

$$
\begin{aligned}
M^D &= \{D \,|\, i : D \in A',\ D \text{ is a concept name}\} \\
M^{\neg D} &= \{D \,|\, i : \neg D \in A',\ D \text{ is a concept name}\} \\
M^{\exists} &= \{R \,|\, i : \exists R.C \in A'\} \cup \{R \,|\, (i,j) : R \in A\} \\
M^{\forall} &= \{R \,|\, i : \forall R.C \in A'\}
\end{aligned}
$$

Note the distinction between the initial A-box A and its completion A'. It is important that all restrictions for a certain individual are "reflected" in the pmodel. The idea of model merging is that there is a simple sound but incomplete test for showing that adding the assertion $i : \neg C$ to the ABox will not lead to a clash (see [10] for details) and, hence, i is not an instance of the query concept C. Let MS be a set of pmodels. The pmodel merging test is: $atoms\_mergable(MS) \wedge roles\_mergable(MS)$ where *atoms_mergable* tests for a possible primitive clash between pairs of pseudo models. It is applied to a set of pseudo models MS and returns *false* if there exists a pair $\{M_1, M_2\} \subseteq MS$ with $(M_1^D \cap M_2^{\neg D}) \neq \emptyset$ or $(M_1^{\neg D} \cap M_2^D) \neq \emptyset$. Otherwise it returns *true*.

The algorithm *roles_mergable* tests for a possible role interaction between pairs of pseudo models. It is applied to a set of pseudo models MS and returns *false* if there exists a pair $\{M_1, M_2\} \subseteq MS$ with $(M_1^{\exists} \cap M_2^{\forall}) \neq \emptyset$ or $(M_1^{\forall} \cap M_2^{\exists}) \neq \emptyset$. Otherwise it returns *true*. The reader is referred to [11] for the proof of the soundness of this technique and for further details.

It should be emphasized that the complete set of data structures for a particular completion is not maintained by a DL reasoner. The pmodels provide for an appropriate excerpt of a completion needed to determine non-instances.

3.4 Obvious Instances: Exploiting Information from the Precompletion

Another central optimization technique to ensure data description scalability as it is required for LUBM is to also find "obvious" instances with minimum effort.

Given an initial ABox consistency test and a completion one can consider all deterministic restrictions, i.e., one considers only those completion data structures (from now on called constraints) for which there are no choice points in the tableau proof (in other words, consider only those constraints that do not have dependency information attached). These constraints constitute a so-called precompletion.[2] Note that in a precompletion, no constraints are violated because we assume that the precompletion is computed from an existing completion.

Given the precompletion constraints, for each individual i, an approximation of the most-specific concept (MSC) is computed as follows (the approximation is called MSC'). For all constraints representing role assertions of the form $(i, j) : R$ (or $(j, i) : R$) add constraints of the form $i : \exists R.\top$ (or $i : \exists R^{-1}.\top$). Afterwards, constraints for a certain individual i are collected into a set $\{i : C_1, \ldots, i : C_n\}$. Then, $MSC'(i) := C_1 \sqcap \ldots \sqcap C_n$. Now, if $MSC'(i)$ is subsumed by the query concept C, then i must be an instance of C. In the case of LUBM many of the assertions lead to deterministic constraints in the tableau proof which, in turn, results in the fact that for many instances of a query concept C (e.g., $Faculty$ as in query $Q9$) the instance problem is decided with a subsumption test based on the MSC' of each individual. Subsumption tests are known to be fast due to caching and model merging [13]. The more precisely $MSC'(i)$ approximates $MSC(i)$, the more often an individual can be determined to be an obvious instance of a query concept. Obviously, it might be possible to determine obvious instances by directly considering the precompletion data structures. However, at this implementation level a presentation would be too detailed. The main point is that, due to our findings, the crude approximation with MSC' suffices to solve many instance tests in LUBM.

If query atoms are used as testers, in LUBM it is the case that in a large number of cases the test for obvious non-instances or the test for obvious instances determines the result. However, for some individuals i and query concepts C both tests do not determine whether i is an instance of C (e.g., this is the case for $Chair$). Since both of these "cheap" tests are incomplete, for some individuals i a refutational ABox consistency test resulting from adding the claim $i : \neg C$ (refutational instance test) must be decided with a sound and complete tableau prover. For some concepts C, the set of candidates is quite large. Considering the volume of assertions in LUBM (see below for details), it is easy to see that the refutational instance test should not start from the initial, unprocessed ABox in order to ensure scalability.

For large ABoxes and many repetitive instance tests it is a waste of resources to "expand" the very same initial constraints over and over again. Therefore, the precompletion resulting from the initial ABox consistency test is used as a starting point for refutational instance tests. The tableau prover keeps the precompletion in memory. All deterministic constraints are expanded, so if some

[2] Cardinality measures for concept names, required for determining optimized query execution plans, could be made more precise if cardinality information was computed by considering a precompletion. However, in the case of LUBM this did not result in better query execution plans.

constraint is added, only a limited amount of work is to be done. To understand the impact of refutation-based instance tests on the data description scalability problem, a more low-level analysis on tableau provers architectures is required.

3.5 Index Structures for Optimizing Tableau Provers

Tableau provers are fast w.r.t. backtracking, blocking, caching and the like. But not fast enough if applied in a naive way. If a constraint $i : \neg C$ is added to a precompletion, the tableau prover must be able to very effectively determine related constraints for i that already have been processed. Rather than using linear search through lists of constraints, index structures are required for bulk data descriptions.

First of all, it is relatively easy to classify various types of constraints (for exists restrictions, value restrictions, atomic restrictions, negated atomic restrictions, etc.) and access them effectively according to their type. We call the corresponding data structure an active record of constraint sets (one set for each kind of constraint). For implementing a tableau prover, the question for an appropriate data structure for these sets arises. Since ABoxes are not models, (dependency-directed) backtracking cannot be avoided in general. In this case, indexing the set of "relevant" constraints in order to provide algorithms for checking if an item is an element of a set or list (element problem) is all but easy. Indexing requires hashtables (or trees), but backtracking requires either frequent copying of index structures (i.e., hashtables) or frequent insertion and deletion operations concerning hashtables. Both operations are known to be costly.

Practical experiments with LUBM and the DL system RacerPro (see below for a detailed evaluation) indicate that the following approach is advantageous in the average case. For frequent updates of the search space structures during a tableau proof, we found that simple lists for different kinds of constraints are most efficient, thus we have an active record of lists of constraints. New constraints are added to the head of the corresponding list, a very fast operation. During backtracking, the head is chopped off with minimum effort. The list representation is used if there are few constraints, and the element problem can be decided efficiently. However, if these lists of constraints get large, performance decreases due to linear search. Therefore, if some list from the active record of constraints gets longer than a certain threshold, the record is restructured and the list elements are entered into an appropriate index structure (hashtables with individuals as keys). Afterwards the tableau prover continues with a new record of empty lists as the active record. The pair of previous record of lists and associated hashtable is called a generation. From now on, new constraints are added to the new active record of constraints and the list(s) of the first generation are no longer used. For the element problem the lists from the active record are examined first (linear search over small lists) and then, in addition, the hashtable from the first generation is searched (almost linear search). If a list from the active record gets too large again, a new generation is created. Thus, in general we have a sequence of such generations, which are then considered

for the element test in the obvious way. If backtracking occurs, the lists of the appropriate generation are installed again as the active record of lists. This way of dealing with the current search state allows for a functional implementation style of the tableau prover which we prefer for debugging purposes. However, one might also use a destructive way to manage constraints during backtracking. Obviously, all (deterministic) constraints from the initial Abox can be stored in a hashtable. In any case, the main point here is that tableau provers need an individual-based index to efficiently find all constraints in which an individual is involved. In the evaluation of other optimization techniques (see below) we presuppose that a tableau prover is equipped with this technology, and thus we can assume that each refutational instance test is rather fast.

3.6 Transforming Sufficient Conditions into Conjunctive Queries

Up to now we can detect obvious instances based on told and taxonomical information (almost constant time, see Section 3.2) as well as information extracted from the precompletion (linear time w.r.t. the number of remaining candidate individuals and a very fast test, see Section 3.4). Known non-instances can be determined with model merging techniques applied to individual pmodels (also a linear process w.r.t. the number of remaining candidate individuals but with a very fast test, see Section 3.3). However, there might still be some candidates left. Using the results in [10] it is possible to use dependency-directed instance retrieval and binary partitioning. Our findings suggest that in the case of LUBM, for example for the concept *Chair*, the remaining refutational tableau proofs are very fast. However, for *Chair* a considerable number of candidates remain since there are many *Persons* in LUBM. In application scenarios such as those we investigate with LUBM we have 200,000 individuals and more (see the evaluation below) with many *Persons*. Even if each single instance test lasts only a few dozen microseconds, query answering will be too slow, and hence additional techniques should be applied to solve the data description scalability problem.

The central insight for another optimization technique is that in the presence of sufficient conditions for concept names given in the Tbox, query atoms that refer to names might be transformed. Let us consider the query $ans(x) \leftarrow Chair(x)$. For *Chair*, sufficient conditions are given as part of the TBox (see above). Thus, in principle, we are looking for instances of the concept $Person \sqcap \exists headOf.Department$. The key to optimizing query answering becomes apparent if we transform the definition of *Chair* into a conjunctive query and derive the optimized version $Q15'$

$$Q15 : ans(x) \leftarrow Person(x), headOf(x, y), Department(y)$$
$$Q15' : ans(x) \leftarrow Department(y), headOf^{-1}(y, x), Person(x)$$

Because there exist fewer *Departments* than *Persons* in LUBM, search for bindings for x is substantially more focused in $Q15'$ (which is the result of automatic query optimization, see above). In addition, in LUBM, the extension of *Department* can be determined with simple index-based tests only (only

hierarchies are involved) and thus the heuristics of the query optimizer produce optimal results. With the *Chair* example one can easily see that the standard approach for instance retrieval can be optimized dramatically with rewriting concept query atoms if certain conditions are met.

Algorithm 1. $rewrite(tbox, concept, var)$:

if $meta\_constraints(tbox) \neq \emptyset \vee definition(concept) = \top$ **then**
  return $(concept(var))$
else
  $\{atom_1, \ldots, atom_n\} :=$
    $rewrite\_0(tbox, concept, definition(tbox, concept), var, \{\})$
  return $(atom_1, \ldots, atom_n)$

Algorithm 2. $rewrite\_0(tbox, concept, var, exp)$:

if $definition(concept) = \top \vee concept \in exp$ **then**
  return $\{concept(var)\}$
else
  ;; catch installs a marker to which the control flow can be thrown
  catch $not\_rewritable$
    $rewrite\_1(tbox, concept, definition(tbox, concept), var, \{concept\} \cup exp)$

Algorithm 3. $rewrite\_1(tbox, concept\_name, definition, var, exp)$:

if $(definition = A)$ where A is an atomic concept **then**
  return $rewrite\_0(tbox, definition, var, \{definition\} \cup exp)$
else
  if $(definition = \exists R.C)$ **then**
    $filler\_var := fresh\_variable()$
    return $\{R(var, filler\_var)\} \cup rewrite\_0(tbox, C, filler\_var, exp)$
  else
    if $(definition = C_1 \sqcap \ldots \sqcap C_n)$ **then**
      return $rewrite\_1(tbox, concept\_name, C_1, var, exp)$
          $\cup \ldots \cup$
          $rewrite\_1(tbox, concept\_name, C_n, var, exp)$
    else
      ;; throw the control flow out of rewrite_1 recursion
      ;; back to the call to rewrite_1 in rewrite_0 and return {concept_name(var)}
      throw $not\_rewritable\,\{concept\_name(var)\}$

The rewriting algorithm is defined in Algorithms 1, 2, and 3. Every concept query atom $C(x)$ in a conjunctive query used is replaced with $rewrite(query\_tbox, C, x)$ (and afterwards, the query is optimized).

Some auxiliary functions are used. The function $definition(C)$ returns sufficient conditions for a concept name C (the result is a concept), and the function $meta\_constraints(tbox)$ indicates whether there are some meta constraints left after GCI transformation (see [15], the result is a set of concepts). In addition, we use a function $fresh\_variable$ that generates a new variable that was not used before.

If there is no specific definition or there are meta constraints, rewriting is not applied (see Algorithm 1). It is easy to see that the rewriting approach is sound. However, it is complete only under specific conditions, which can be automatically detected. If we consider the Tbox $T = \{D \doteq \exists R.C\}$, the Abox $A = \{i : \exists R.C\}$ and the query $ans(x) \leftarrow D(x)$, then due to the algorithm presented above the query will be rewritten as $ans(x) \leftarrow R(x, y), C(y)$. For variable bindings, the query language nRQL (see above) considers only those individuals that are explicitly mentioned in the Abox. Thus i will not be part of the result set because there is no binding for y in the Abox A. Examining the LUBM Tbox and Abox it becomes clear that in this case for every $\exists R.C$ that is applicable to an individual i in a tableau proof there already exist constraints $(i, j) : R$ and $j : C$ in the original Abox (LUBM was derived from a database scenario). However, even if this is not the case, the technique can be employed under some circumstances.

Usually, in order to construct a model (or a completion to be more precise), tableau provers create a new individual for each constraint of the form $i : \exists R.C$ and add corresponding concept and role assertions. These newly created individuals are called anonymous individuals. Let us assume, during the initial Abox consistency test a completion is found. As we have discussed above, a precompletion is computed by removing all constraints that depend on a choice point. If there is no such constraint, the precompletion is identical to the completion that the tableau prover computed. Then, assuming that blocking is postponed to fit the query with the largest nesting depth, the set of bindings for variables is extended to the anonymous individuals found in the precompletion. The rewriting

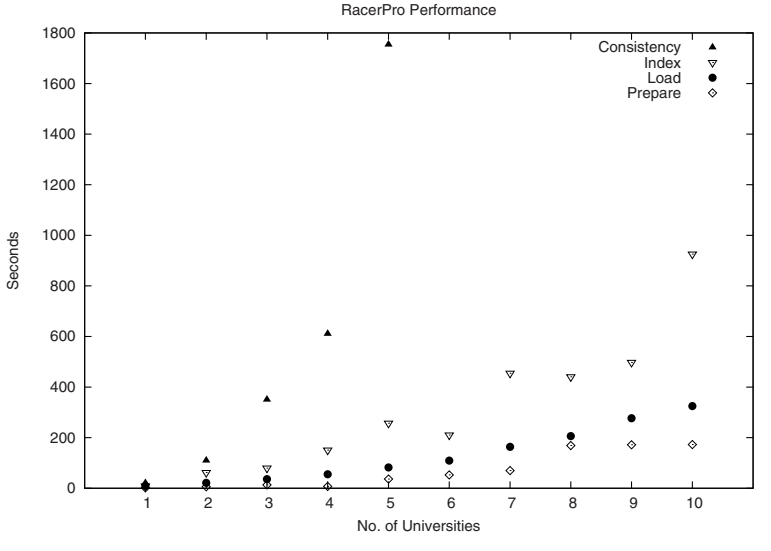

Fig. 1. Runtimes for loading, preparation, abox consistency checking and indexing

technique for concept query atoms is applicable (i.e., is complete) under these conditions. Even if the rewriting technique is not complete (i.e., s.th. is removed from the completion in order to derive the precompletion), it can be employed to reduce the set of candidates for binary partitioning techniques that can speed-up this process considerably in the average case (c.f., [10]).

The transformation approach discussed in this section is reminiscent of an early transformation approach discussed in [19]. In fact, ideas from translational approaches from DLs to disjunctive datalog [18] can also be integrated in tableau-based approaches. In the following section, we will evaluate how the optimization techniques introduced up to now provide a contribution to the data description scalability problem.

4 Evaluation

The significance of the optimization techniques introduced in this contribution is analyzed with the system RacerPro 1.9. RacerPro is freely available for research and educational purposes (`http://www.racer-systems.com`). The runtimes we present in this section are used to demonstrate the order of magnitude of time resources that are required for solving inference problems. They allow us to analyze the impact of proposed optimization techniques. We start with an evaluation of optimizations for (restricted) conjunctive queries with LUBM and turn to instance retrieval w.r.t. applications-specific knowledge bases afterwards.

An overview about the size of the LUBM benchmarks is given in Table 1. The runtimes for loading the data descriptions, transforming them into abstract syntax trees (preparation), and indexing are shown in Figure 1 (AMD 64bit processor, 4GB, Linux OS). It is important to note that these curves are roughly linear, thus, no reasoning is included in these phases. In Figure 1, also the runtimes for checking ABox consistency together with the computation of the precompletion are indicated (Consistency, black triangle). The "quadratic" shape reveals that this phase should be subject to further optimizations.

In Figure 2, average query-answering times for running all 14 LUBM queries on data descriptions for an increasing number of universities are presented (see Table 1 for an overview on the number of individuals, concept assertions, and role assertions). We use different modes (A, B, and C) to indicate the effects of optimization techniques. All modes are complete with respect to the Tbox and data descriptions (Abox) we used for the LUBM experiments in this paper.

Table 1. Linearly increasing number of individuals, concept assertions and role assertions for different numbers of universities

Univs	Inds	Concept Assertions	Role Assertions
1	17174	53738	49336
3	55664	181324	166682
5	102368	336256	309393
10	207426	685569	630753

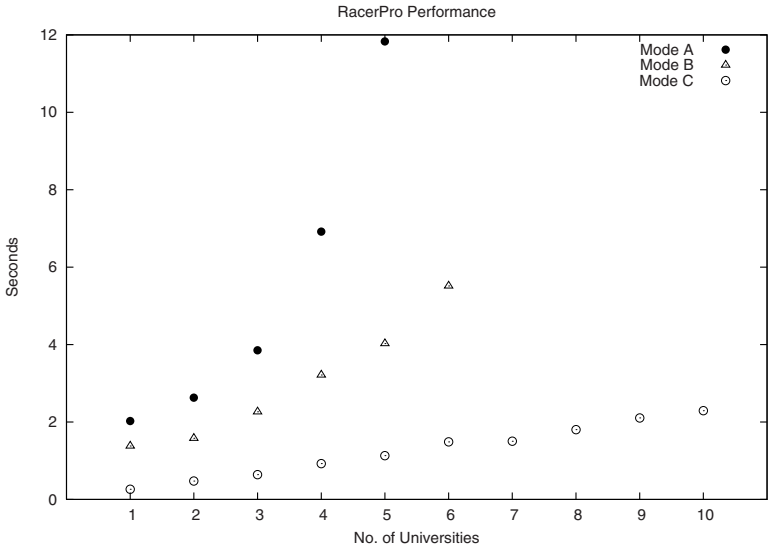

Fig. 2. Runtimes of 14 LUBM queries with different optimization settings (see text)

In mode A and B, concept definitions are not rewritten into conjunctive queries (see Section 3.6). In mode A, full constraint reasoning on OWL datatypes is provided. Thus, datatype properties are encoded in the obvious way as roles referring to individuals which, in turn, refer to values via concrete domain attributes. With concrete domains, arbitrary constraint systems can be specified in an Abox [12]. This means, (multiple) attribute values of *multiple* Abox individuals can constrained. In OWL one can only restrict (multiple) attributes of a *single* individual (nominal). For LUBM, however, only OWL datatypes are used, and no constraint reasoning is required because datatypes are used only to associate individuals with strings in the Abox. In order to answer queries, only "told values" must be retrieved. Therefore, in mode B, told value retrieval is performed only. As Figure 2 shows, this is much more efficient (but less powerful in the general case, of course). Mode C in Figure 2 presents the runtimes achieved when definitions of concept names are rewritten to conjunctive queries (and told value reasoning on datatypes only is employed, as in mode B). The results for mode C indicate that for deterministic knowledge bases such as LUBM, scalability for instance retrieval can be achieved with tableau-based retrieval engines.

5 Conclusion and Future Work

We take LUBM as a representative for largely deterministic data descriptions that can be found in practical applications. The investigations reveal that description logic systems can be optimized to also be able to deal with large bulks of deterministic descriptions quite effectively. Mode C indicates that performance

scales well with an increasing number of data descriptions given the expressivity of the language used in the ontology meets certain requirements. Our work is based on a tableau calculus which has shown to be reliable if expressivity is increased (see the results in mode B). The linear shape of the curve in mode C suggests that the proposed technology ensures that performance scales if high expressivity is not required. LUBM is in a sense too simple but the benchmark allows us to study the data description scalability problem.

Note that we argue that the concept rewriting technique is advantageous not only for RacerPro but also for other tableau-based systems. Future work will investigate optimizations of large Aboxes and more expressive Tboxes. Our work is based on the thesis that for investigating optimization techniques for Abox retrieval w.r.t. more expressive Tboxes, we first have to ensure scalability for Aboxes and Tboxes such as those we discussed in this paper. We have shown that the results are encouraging.

References

1. F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.
2. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
3. S. Bechhofer, I. Horrocks, and D. Turi. The OWL instance store: System description. In *Proceedings CADE-20*, LNCS. Springer Verlag, 2005.
4. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Data complexity of query answering in description logics. In *Proc. of the 2005 Description Logic Workshop (DL 2005)*. CEUR Electronic Workshop Proceedings, http://ceur-ws.org/, 2005.
5. Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the decidability of query containment under constraints. In *Proc. of the 17th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'98)*, pages 149–158, 1998.
6. H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Boook*. Prentice Hall, 2092.
7. Birte Glimm and Ian Horrocks. Handling cyclic conjunctive queries. In *Proc. of the 2005 Description Logic Workshop (DL 2005)*, volume 147. CEUR (`http://ceur-ws.org/`), 2005.
8. Y. Guo, J. Heflin, and Z. Pan. Benchmarking DAML+OIL repositories. In *Proc. of the Second Int. Semantic Web Conf. (ISWC 2003)*, number 2870 in LNCS, pages 613–627. Springer Verlag, 2003.
9. Y. Guo, Z. Pan, and J. Heflin. An evaluation of knowledge base systems for large OWL datasets. In *Proc. of the Third Int. Semantic Web Conf. (ISWC 2004)*, LNCS. Springer Verlag, 2004.
10. V. Haarslev and R. Möller. Optimization techniques for retrieving resources described in OWL/RDF documents: First results. In *Ninth International Conference on the Principles of Knowledge Representation and Reasoning, KR 2004, Whistler, BC, Canada, June 2-5*, pages 163–173, 2004.

11. Volker Haarslev, Ralf Möller, and Anni-Yasmin Turhan. Exploiting pseudo models for tbox and abox reasoning in expressive description logics. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001)*, volume 2083 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2001.
12. Volker Haarslev, Ralf Möller, and Michael Wessel. The description logic $\mathcal{ALCNH}_{R+}$ extended with concrete domains: A practically motivated approach. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001)*, pages 29–44, 2001.
13. I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.
14. I. Horrocks, O. Kutz, and U. Sattler. The even more irresistible $\mathcal{SROIQ}$. Technical report, University of Manchester, 2006.
15. I. Horrocks and S. Tobies. Reasoning with axioms: Theory and practice. In *Proc. of the 7th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 285–296, 2000.
16. Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Reasoning with individuals for the description logic $\mathcal{SHIQ}$. In David McAllester, editor, *Proc. of the 17th Int. Conf. on Automated Deduction (CADE 2000)*, volume 1831 of *Lecture Notes in Computer Science*, pages 482–496. Springer-Verlag, 2000.
17. Ian Horrocks and Sergio Tessaris. A conjunctive query language for description logic ABoxes. In *Proc. of the 17th Nat. Conf. on Artificial Intelligence (AAAI 2000)*, pages 399–404, 2000.
18. B. Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Univ. Karlsruhe, 2006.
19. B. Motik, R. Volz, and A. Maedche. Optimizing query answering in description logics using disjunctive deductive databases. In *Proceedings of the 10th International Workshop on Knowledge Representation Meets Databases (KRDB-2003)*, pages 39–50, 2003.
20. M. Wessel and R. Möller. A high performance semantic web query answering engine. In *Proc. of the 2005 Description Logic Workshop (DL 2005)*. CEUR Electronic Workshop Proceedings, http://ceur-ws.org/, 2005.
21. Z. Zhang. Ontology query languages for the semantic web: A performance evaluation. Master's thesis, University of Georgia, 2005.