On the Scalability of Description Logic Instance Retrieval

Ralf Möller, Volker Haarslev, Michael Wessel

1 Introduction

Although description logics (DLs) are becoming more and more expressive, 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 (scalability problem). In the literature, the scalability problem has been tackled from different perspectives. We see two main approaches, the layered approach and the integrated approach. In the former 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 expansion and optimization. See, e.g., [9, 4] (DLDB), [1] (Instance Store), or [2] (DL-Lite). We acknowledge the interesting progress of these approaches, but we note that the corresponding techniques 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 advantage of fast runtimes. Therefore, in this paper, we pursue the integrated approach that considers query answering with a description logic system. For the time being we ignore the problems associated with persistency and investigate specific knowledge bases (see below).

Continuing our work in [5], in this paper we investigate the constraints imposed on the architecture of description logic reasoning systems, and we investigate the assumptions underlying the heuristics used in instance retrieval algorithms. The contribution presents and analyzes the main results we have found about how to solve 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. We assume the reader is familiar with DLs in general and tableau-based decision procedures in particular.

2 Lehigh University Benchmark

In order to investigate the scalability problem, we use the Lehigh University BenchMark (LUBM, [3, 4]). LUBM data descriptions are automatically generated, but we do not exploit this in this paper (e.g., by tailoring optimization techniques to regular patterns stemming from generator peculiarities). The LUBM queries are formalized as 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 [8] (the language is called nRQL). Variables are only bound to individuals mentioned in the Abox. In the notation of nRQL queries used in this paper we assume that different variables may have the same bindings. In addition, variables that do not occur in the head of a query are assumed to be existentially quantified and are are also only bound to individual names mentioned in the

Abox.

Below, LUBM queries 9 and 12 are shown in order to present a flavor of the kind of query answering problems – note that $'www.University0.edu'$ is an individual and $subOrganizationOf$ is a transitive role. Please refer to [3, 4] for more information about the LUBM queries.

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

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

In order to investigate the 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. Among other axioms, the LUBM TBox contains axioms that express necessary and sufficient conditions on $Chair$. For instance, there is an axiom $Chair \doteq Person \sqcap \exists headOf.Department$. For evaluating optimization techinques 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 cannot 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. For the discussion of optimization techniques, we assume that for transitive and inverse roles, the Abox is extended in the obvious way such that implicit role assertions are made explicit. In addition, the conclusions from domain and range restrictions are made explicit in the Abox as well.

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 heuristic 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 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, an ordered query execution plan must be determined. 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, 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 of which concept 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. The index is organized in such a way that retrieving the instances of a concept A or one of its ancestors requires (almost) constant time. The index is particularly useful to provide bindings for variables if, despite all optimization attemps 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 query $Q9$ using told information yields the following query execution plan.

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

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

If the TBox contains only role inclusion axioms and GCIs of the form $A \sqsubseteq A_1 \sqcap \ldots \sqcap A_n$, i.e., if the TBox forms a hierachy, the index-based retrieval discussed in this section is complete (see [1]). 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.

3.3 Obvious Non-Instances: Exploiting Information from One Completion

The detection of "obvious" non-instances of a given concept C can be optimized by using a model merging operator defined for so-called individual pseudo models (aka pmodels) as specified in [5]. The central idea is to compute a pmodel from a completion that is derived by the tableau prover. Note that 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 [5] for details) and, hence, i is not an instance of C. Note that, usually, the complete set of data structures for a particular completion is not maintained by a reasoner. The pmodels provide for an excerpt of a completion needed to determine non-instances.

3.4 Obvious Instances: Exploiting Information from the Precompletion

A central optimization technique to ensure scalability as it is required for LUBM is to also find "obvious" instances with minimum effort. Given an initial ABox consistency test one can consider all deterministic restrictions, i.e., consider only those tableau structures (from now on called constraints) for which there are no *previous* choice points in the tableau proof (in other words, consider only those constraints that do not have dependency information

attached). These constraints, which are computed during the tableau proof, describe a so-called precompletion.[1] Note that in a precompletion, no restrictions are violated.

Given the precompletion constraints, for each individual an approximation of the most-specific concept (MSC') of an individual i is computed as follows. 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. 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, these are technical details. The main point is that, due to our findings, the crude approximation described above suffices to detect all "obvious" instances 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 tests are incomplete, for some individuals i a refutational ABox consistency test resulting from adding the claim $i : \neg C$ must be decided with a sound and complete tableau prover. For some concepts C, the set of remaining instances i for which the "expensive" ABox instance test must be used is quite large, or, considering C as a generator, the set of candidates is quite large. In any case, considering the volume of assertions in LUBM (see below for details), it is easy to see that the refutational ABox consistency test must not start from the initial, unprocessed ABox in order to ensure scalability.

For large ABoxes and many repetitive instance tests it is mandatory not 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 constraint is added, only a limited amount of work is to be done. 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.

3.5 Index Structures for Optimizing Tableau Provers

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 set data structure 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

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

backtracking requires either frequent copying of index structures (i.e., hashtables) or frequent inserting and deleting items from hashtables. Both operations are known to be costly.

Practical experiments with LUBM and many other knowledge bases 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 an individual is involved in. In the evaluation of other optimization techniques (see below) we presuppose that a tableau prover is equipped with this technology.

3.6 Transforming Sufficient Conditions into Conjunctive Queries

Up to now we have discussed the optimization of concept atoms used as testers in the query execution plan. If concepts are used as generators, indeed a retrieval problem rather than only an instance test problem has to be solved. Using the results in [5] it is possible to linearly iterate over all individuals, check for obvious non-instances and obvious instances, and then, for the set of remaining candidates dependency-directed instance retrieval and binary partitioning can be used (see [5]). Our findings suggest that in the case of LUBM, for example for the concept $Chair$, the remaining tableau proofs are very fast. Nevertheless it is the case that the whole procedure is based on a linear iteration over all individuals. In application scenarios such as those we investigate with LUBM we have 200,000 individuals and more, and even if each single test lasts only a few dozen microseconds, query answering will be too slow, and hence additional techniques must be applied to solve the scalability problem.

The central insight for another optimization technique is that conjunctive queries can be optimized according to the above-mentioned arguments whereas for concept-based retrieval queries, optimization is much harder to achieve. 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 $Q15'$

$Q15 : ans(x) \leftarrow Person(x), headOf(x, y), Department(y)$

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

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

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

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). With the *Chair* example one can easily see that the standard approach for instance retrieval can be optimized dramatically if query atoms are rewritten as indicated in the above example.

If there is no specific definition or there are meta constraints, rewriting is not applied. 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. Hence, 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 $i : \exists R.C$ appearing in a tableau proof there already exist constraints $(i, j) : R$ and $j : C$ in the original Abox. 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, the set of bindings for variables is extended to the anonymous individuals found in the precompletion. The rewriting 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 to derive the precompletion), it can be employed to reduce the set of candidates for binary partitioning techniques (c.f., [5]) that can speed of this process considerably in the average case.

The transformation approach discussed in this section is reminiscent to an early transformation approach discussed in [7]. In fact, ideas from translational approaches from DLs to disjunctive datalog [6] 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 and Conclusion

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.

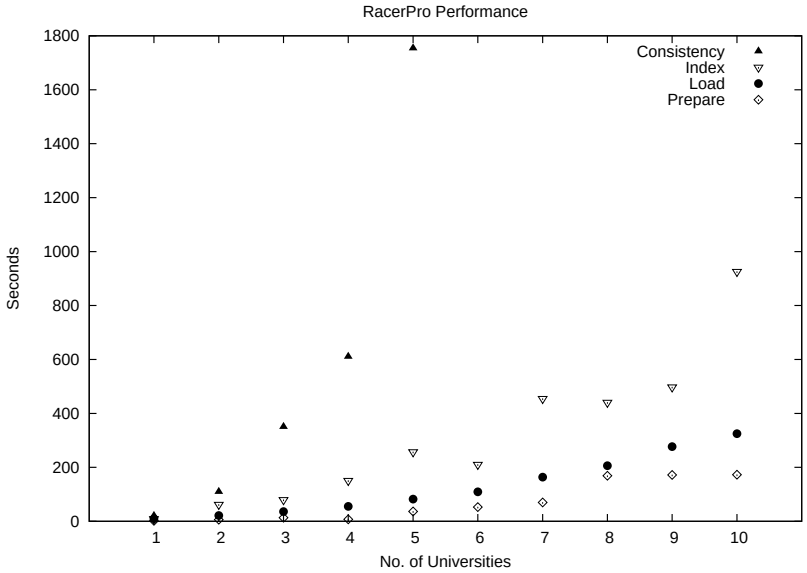

Figure 1: Runtimes for loading, preparation, abox consistency checking and indexing.

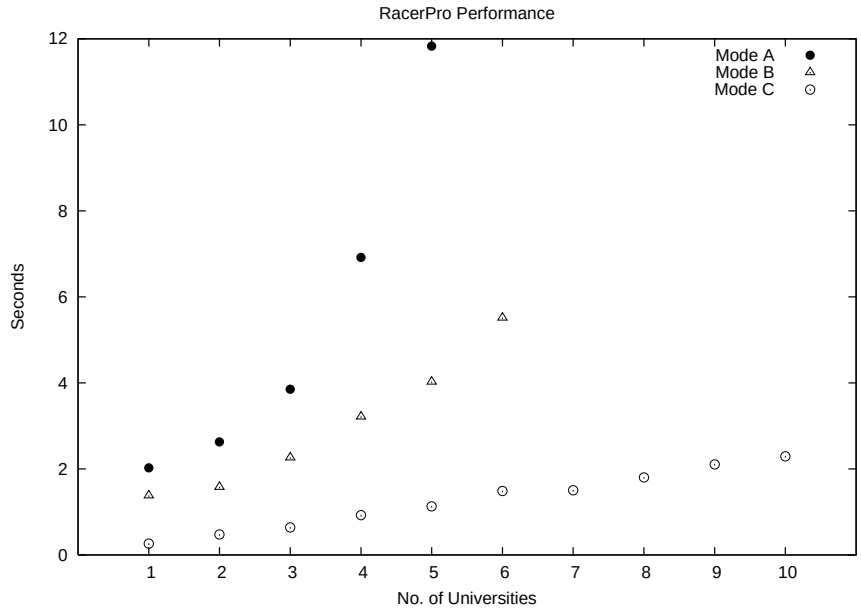

Figure 2: Runtimes of 14 LUBM queries with different optimization settings (see text).

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 the curves are roughly linear, thus, no reasoning is included in these phases. In

Figure 1, the runtimes for checking ABox consistency and precompletion data structures (see above) are indicated (the shape, which appears to be quadratic, reveals that this phase are 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). In mode A and B, concept definitions are not rewritten into conjunctive queries (see Section 3.6). In mode A, full constraint reasoning on datatypes is provided. However, this is not required for LUBM. Therefore, in mode B, told value retrieval is performed only. As Figure 2 shows, this is much more efficient. 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 is enabled). Mode C indicates that scalability for instance retrieval can be achieved with tableau-based retrieval engines. 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 that we discussed in this paper. We have shown that the results are encouraging.

References

[1] S. Bechhofer, I. Horrocks, and D. Turi. The OWL instance store: System description. In *Proceedings CADE-20*, LNCS. Springer Verlag, 2005.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] B. Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Univ. Karlsruhe, 2006.

[7] 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.

[8] 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.

[9] Z. Zhang. Ontology query languages for the semantic web: A performance evaluation. Master's thesis, University of Georgia, 2005.