Towards a Scalable and Efficient Middleware for Instance Retrieval Inference Services

Tobias Berger, Alissa Kaplunova, Atila Kaya, Ralf Möller

Hamburg University of Science and Technology
Hamburg, DE
{tobias.berger, al.kaplunova, at.kaya, r.f.moeller}@tuhh.de

Abstract. We argue for the usefulness of convenient optimization criteria, when building scalable and efficient instance retrieval inference services for ontology-based applications. We discuss several optimization criteria, especially a dedicated load balancing strategy, evaluate the approach with practical experiments on a particular implementation and present results.

1 Introduction

Over the past years the Web Ontology Language (OWL) has successfully been used by an increasing number of applications for building ontology-based systems. Description Logic (DL) reasoners allow for formal domain modeling with OWL, and support decidable reasoning problems [1]. Experiences show that many practical applications require reasoning, especially instance retrieval inference services[1], on large knowledge bases (KBs)[2]. Furthermore, they require the quality of service to remain stable by increasing retrieval request frequency.

Recently, techniques to meet the first requirement have been introduced. In [13] optimization techniques for retrieval inference services on large KBs are investigated and promising results are presented. However, to the best of our knowledge, the question how DL systems can deal with an increasing number of parallel retrieval requests is still open. Indeed, today's reasoners are not well-equipped to offer the same quality of service if the frequency of retrieval requests increases, e.g., if multiple clients pose queries concurrently. This is due to the fact that, at the current state of the art, reasoners cannot process multiple queries at the same time, and thus cannot reduce the answering times compared to the sequential execution of queries. Therefore, in this paper, we address the scaling problem of reasoners with regard to increased retrieval request frequency. For the time being we ignore the data integration problem that has been investigated by the database community [10] and others [5]. We address scenarios where the integration of data from different ontologies is not relevant for retrieval inference services.

At a first glance, taking the existing scaling mechanisms for database systems off the shelf and applying them to the scalability problem of DL reasoners may seem like an appealing solution. In fact, distributed database systems or web servers successfully employ load balancing mechanisms at the present (e.g., [3], [9]). However, we believe that better solutions can be found if both scenario-specific requirements of ontology-based applications and peculiarities of reasoning systems are taken into account.

Answering expressive DL queries is known to be a more complex task than query answering in standard database systems. We argue that, in order to solve the scalability problem of DL systems, investing more time and computational resources in

[1] Henceforth called retrieval inference services only.

[2] Actually, retrieval inference services are not independent from other standard inference services. In this paper we take the availability of standard inference services for granted and focus on retrieval inference services.

the analysis of queries pays off, if the results of the analysis are exploited for finding better optimizations and load balancing strategies. Therefore, we present convenient optimization criteria (including a novel load balancing strategy) and investigate how they can be exploited. The contribution evaluates the results of practical experiments on a particular implementation, i.e., a middleware that mediates between multiple clients and reasoners.

2 Requirements for the Middleware

When building practical OWL-based applications, the efficient interaction between clients and DL-based systems for information retrieval is still a big challenge for modern reasoners. Although research on optimization techniques for standalone reasoning systems reported substantial performance improvements recently (e.g., [13]), optimization techniques for front-end systems managing multiple reasoners have not been developed. In the following, we present the requirements for a scalable and efficient middleware for retrieval inference services.

Handling of concurrent client requests Although highly-optimized reasoners can achieve significant performance improvements for a single client, this effect decreases in scenarios where multiple clients pose queries concurrently. In fact, a single reasoner instance cannot process several client requests in parallel (although they can have multiple open queries for which further tuples can be retrieved in a subsequent incremental query). Thus, as long as the reasoner is processing a query (which usually includes activities such as parsing the query, reading the corresponding knowledge base, classifying it, as well as finding requested number of answer tuples and returning them) all other clients have to wait in a queue.

Due to these observations, we claim that, from a practical point of view, one of the main requirements for the middleware is the ability to handle concurrent client requests. For that, the middleware has to act as a proxy that employs sophisticated dispatching algorithms in order to delegate queries to multiple back-end DL reasoners.

Support for standard query languages The middleware has to implement widely used and accepted standard components and protocols such as interfaces, query languages and communication protocols in order to enable easy integration with a wide range of applications. The OWL Query Language (OWL-QL) is a query language and communication protocol for instance retrieval [4]. OWL-QL defines a protocol for query-answering dialogs among agents using knowledge represented in OWL. It facilitates the communication between querying agents (clients) and answering agents (servers). Among features, OWL-QL allows clients to pose conjunctive queries and request partial sets of answers[3]. The retrieval of partial sets of answers, a.k.a. iterative query answering, is supported by modern reasoners (e.g., RacerPro [6]). For iterative query answering, the reasoner can be configured to compute the next tuples on demand (lazy mode) or in a concurrent way (proactive mode). Moreover, it can be instructed to return “cheap” (easily inferable from syntactic input) tuples first [7]. Consequently, besides supporting OWL-WL, the middleware has to provide for efficient iterative query answering by exploiting corresponding functionality and configuration options (such as cheap tuples first mode) offered by reasoners.

[3] Not that OWL-QL does not specify anything about server-side implementation details such as caching, number of reasoners used as back-end etc.

Scalability and efficiency Another key requirement for the middleware is the capability to scale up in order to support scenarios that cause high query traffic. The required scalability (and high-availability) can be achieved by increasing the number of reasoners managed by the middleware. In situations where a single middleware may itself become a bottleneck, one possible solution would be to replicate it and to set up a HTTP or hardware dispatcher in front of multiple middleware instances. This way the middleware allows for building modular and hence scalable retrieval inference services for ontology-based applications.

Additionally, the middleware is required to minimize the answering time of each query, which can only be achieved by operating as efficient as possible. It is obvious that for a single query the middleware can not accelerate the computation of query answers. However, if queries are posed concurrently, which regularly occurs in a practical application scenario, the middleware can enable the answering of queries in parallel and hence improve the overall efficiency. Furthermore, it can exploit several optimization criteria for reducing the number of inference service invocations and finding the most appropriate reasoner to answer a query.

3 Optimization Criteria for Parallel ontology-based Retrieval

The primary goal of the middleware is to facilitate scalability and high-availability of retrieval inference services such that the interaction time is minimized for each client, e.g., in scenarios where multiple clients pose queries concurrently. To do this, the middleware has to exploit several optimization criteria when searching for the best decision to answer a query as fast as possible. We start our discussion with existing optimization criteria that can be adapted for the retrieval inference problem and continue with optimization criteria that are specific to ontology-based applications and reasoning.

Cache Usage Cache usage is known to be a key requirement for system efficiency. All standard software systems that have to serve more than a single client, e.g., web servers, use some form of caching. DL reasoners are no exceptions in this matter. They, indeed, incorporate some efficient caching mechanism. However, if a new layer is involved in building up an inference infrastructure, namely a middleware that mediates between clients and reasoners, it is much more efficient to cache inferred knowledge at this layer. The utilization of caches in the middleware layer allows for reducing the communication overhead by avoiding unnecessary communication with reasoners. Furthermore, the middleware managing several reasoners (probably with different KBs loaded) can constitute more extensive caches and return more answers from the cache than a single reasoner.

Query Classification Compared with database systems, query answering can be considered as a complex and expensive task that requires considerable system resources and time. For this reason, the middleware should invest time in the analysis and classification of queries which pays off due to the possibility of selecting an optimal strategy for each query. With respect to ontology-based applications exploiting iterative query answering, queries can be categorized into four classes:

- *First query to an unknown KB*: These queries reference a KB, which is unknown to the middleware. An unknown KB is a KB that has not been loaded by any of the reasoners managed by the middleware.
- *New query to a known KB*: They reference a known KB and are posed for the first time.

- *Known query to a known KB*: Known queries also reference a known KB, but they are asked before (and thus cached by the middleware).
- *Continuation query*: Continuation queries are queries posed by clients to get more answers for a query they have posed previously. This type of queries only play a role if clients want to get additional chunks of answer tuples.

Query Subsumption Subsumption relations between queries can also be exploited to optimize the performance of the middleware for query answering. Given a new query (to a known KB), the computed subsumption hierarchy w.r.t. queries, which have been answered from the same KB previously, can be used to reduce the query answering time.

If the new query is subsumed by one of the previous queries, the search space for answers to the new query can be narrowed down from the whole domain to the answer set of the previous query. This means that the reasoning process can benefit from the subsumption hierarchy to reduce the computation effort and hence answer the query faster. Therefore, a reasoner that already processed a query subsuming the current query becomes the preferred candidate to answer the current query.

In the opposite case, namely if the new query subsumes a previous query, answers of the previous query are obviously also answers for the new query. Therefore, the middleware can first deliver answers from the cache of the previous query. In fact, if the new query requires answers incrementally and the number of requested answers doesn't exceed the cache size of the previous query, the middleware can answer the query without any communication with remote reasoners.

Knowledge Base Distribution The middleware presented in this paper is required to support application scenarios, where queries are allowed to reference any knowledge base for getting answers from. This requires the middleware to make decisions on how to distribute the new knowledge optimally. The distribution of knowledge effects the performance of the middleware. In fact, the performance of the middleware can be improved substantially if reasoners are prepared for query answering in advance, i.e. they have loaded the KBs, classified the terminologies and constructed the index structures for query answering.

For that, the middleware can distribute known KBs to idle reasoners that have not loaded these KBs yet. Following a naive approach the middleware could try to distribute every known KB to every reasoner. However, a reasoner has only a finite amount of computational resources at its disposal and, moreover, some KBs may be requested rarely. In order to optimize the KB distribution, the middleware has to consider the load a KB has produced so far and the states of each reasoner.

Load Balancing At present, load balancing strategies are widely used by web (or application) servers in order to scale web sites (or portals) for increasing requests. In several scenarios the round robin strategy is implemented successfully. Despite the proved success of existing load balancing strategies in present systems, better load balancing solutions can be found with a particular focus on the characteristics of ontology-based applications that require retrieval inference services. In the next section, we present a novel load balancing strategy that can be used to improve the scalability and high-availability of parallel ontology-based retrieval.

4 Load Balancing Strategy

The optimization criteria discussed in the previous section can be considered as separate optimization modules of the middleware. In case other optimization modules cannot propose an optimization, the load balancing module consults the middleware about the appropriate reasoner to dispatch a query to.

The primary goal of the load balancing module is to balance the load (inference tasks) as homogeneously as possible among reasoners in order to prepare the middleware optimally for future queries. For a particular, query if there are any reasoners that have already loaded the referenced KB the strategy of the load balancing module has to prefer these. Additionally, the load balancing strategy has to consider the effects of actions taken by other optimization modules on the load distribution.

Following these ideas, we developed a load balancing algorithm inspired by the Ant Colony Optimization (ACO, [12]) algorithm[4]. The ACO is a probabilistic technique for solving computational problems that can be reduced to finding shortest paths through graphs, which is based on studies on the behavior of ants navigating the landscape. On its way from its nest to the food and on the way back, an ant marks the way it took with so-called pheromones. The chemicals can be sensed by the following ants. In case there is more than one path to the food, a path with a higher pheromone concentration has a bigger probability to be chosen over the paths with a lower concentration. As shorter paths will have a higher concentration, due to more ants have went forth and back, it has the higher probability to be chosen by the following ants.

With the help of artificial pheromones this behavior is imitated by our load balancing strategy. The load balancing module maintains a table that monitors the absolute pheromone level of each reasoner. Whenever a reasoner answers a query, firstly its pheromone level is increased by adding a value that reflects the work associated with answering this query. The value used here is only a estimation, the real value can be measured later. The estimated value is obtained by calculating the average of real values that are caused by previous queries of the same classification type. In a second step all pheromone levels in the table are normalized. This simulates the natural evaporation of the pheromones. Contrary to the ants' behavior the load balancing module prefers the path with the lowest pheromone level, i.e., it instructs the reasoner with the lowest pheromone level to answer the current query. Due to space limitations, we present only the intuition; for details, please refer to [2].

The strategy provides three main benefits. First, the strategy allows to easily reflect all queries that are answered by a reasoner, both the queries that were distributed by the load balancer and those that were distributed by one of the other optimization modules. Second, the strategy grounds its decisions not only on the current load situation of the reasoners, but also partly on the past states, respectively the progression. By increasing the pheromones and normalizing them after, also former queries are represented in the current pheromone level. However, the latter queries have a much higher influence on the current pheromone level than the earlier ones, which underwent several more normalization. The strategy will prevent the load balancing module from overreacting on temporary fluctuations while still concentrating on recent queries. Third, the algorithm also offers the possibility to reflect the anticipated load the query will cause. Higher load queries, like those that involve the loading and indexing of a KB, increase the reasoner's pheromone level more. Thus the load balancing module has a more realistic picture of the load situation at any time.

5 Evaluation

In order to evaluate the optimization modules presented in this contribution, we implemented a software system. The system acts as a front-end for clients, which pose

[4] Henceforth called ACO-LB.

queries in OWL-QL[5] in parallel, by dispatching the queries to back-end reasoners. At the present, it can manage a configurable number of RacerPro instances.

The load balancing strategy and other optimization modules are evaluated with respect to query response times that are measured in various benchmarks. We utilized the Lehigh University Benchmark (LUBM, [11]) to define new benchmarks that are suitable for testing concurrent client requests. Each benchmark represents a scenario and is described by a test plan. Each test plan specifies the number of clients participating in the scenario, the number of OWL-QL queries each client sends, the number of answers required by each query, and the order in which these queries are send. In order to perform tests on different optimization modules more than one OWL ontology is needed[6]. The ontologies used in the tests are sample university ontologies, which are generated automatically using the tools provided by LUBM, and include a large number of individuals. Besides the URLs used to reference them, they are identical so that the measured times are comparable. The queries used in the tests are LUBM queries translated into OWL-QL[7]. Every query used in the tests references exactly one of the sample university ontologies.

The tests are performed with the open source load and performance measurement tool Apache jMeter[8] (JMeter). jMeter is used to execute the tests and measure the response times. The whole testing was performed on a Sun Solaris Server with 8 processors and 32GB memory. The chosen test platform with multiple processors allows to reduce the impact of the network latency on the measured times. During the test the middleware managed different number of RacerPro systems in the version 1.9.0. Due to space limitations, we present only a small extract of results here; for details, please refer to [2].

Evaluation of Load Balancing To demonstrate the advantages of load balancing for OWL-based applications, we compared a system without load balancing against another system with load balancing using two back-end reasoners. In these tests 10 clients, 20 queries and 2 KBs are involved. Starting at the same time, each client sends two mutually different queries one after another. Half of the clients pose queries to one KB and the other half to the other one. The queries of the clients referencing the same KB are different.

First, the tests are performed with our middleware that employs ACO-LB as the load balancing strategy. The results are shown in Figure 1. The queries arriving the system with no load balancing (1 reasoner) have to wait in a queue and hence block each other. The system with load balancing (2 reasoners) benefits from dispatching the KBs to different reasoners in the beginning and parallelizing the query answering later. Due to the parallel answering of the queries, the waiting times for the queries are substantially reduced in the system with load balancing compared with the single reasoner system (compare ACO columns for system with one and two back-end reasoners in Figure 2).

Both performances shown in Figure 1 are measured with the same middleware. To measure the performance of a single reasoner system, the middleware is configured to manage a single reasoner. Therefore Figure 1 also reveals that scaling the middleware by increasing the number of reasoners retains availability and improves the query answering performance.

Later, the tests are repeated, this time with a system that uses the round robin strategy for load balancing.

[5] Only conjunctive queries with distinguished variables are supported

[6] These KBs can be found at `http://www.sts.tu-harburg.de/~at.kaya/racerManager/univ-bench.zip`.

[7] The queries can be found at `http://www.sts.tu-harburg.de/~at.kaya/racerManager/lubm-owlql.zip`

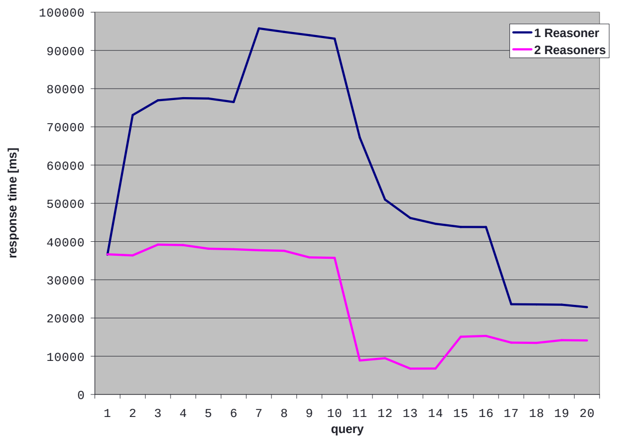

Fig. 1. The performance of the ACO-LB strategy with one and two reasoners

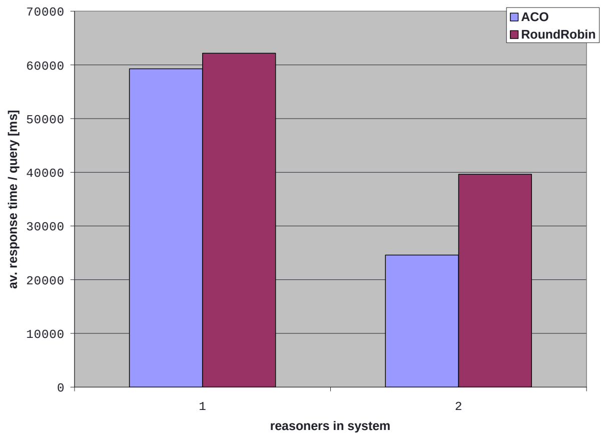

Fig. 2. Average response times of ACO and round robin systems with one and two reasoners

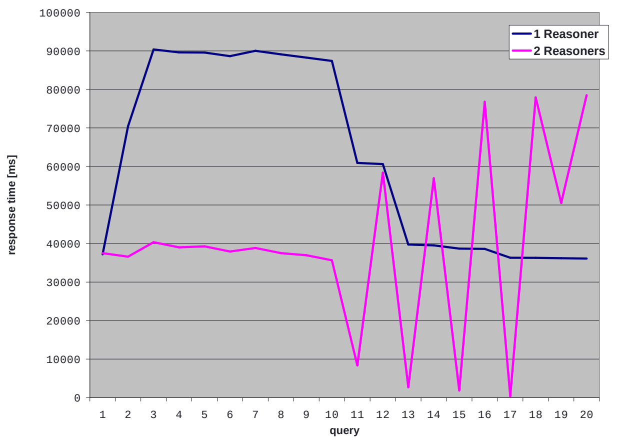

Fig. 3. The performance of the round robin strategy with one and two reasoners

In these tests, the ignorance of the round robin strategy leads to a very unpredictable behavior and higher response times (see Figure 3). The results also reveal that, if average response times are considered, the ACO-LB strategy performs significantly better than the round robin strategy, which is simple but ignorant (see Figure 2).

Evaluation of Middleware Caching In order to proof the claim that caching at the middleware layer are much more efficient than local caching at the back-end reasoners we defined a second benchmark. In the corresponding tests 6 clients, 18 queries and 2 KBs are involved. Starting together, each client sends 3 completely equal queries one after another[8]. Half of the clients pose queries to one KB and the other half to the other one. The queries of clients posing the same KB are different. In order to observe the effects of middleware caching only, both systems are initialized equally, i.e., they both manage two back-end reasoners, each one loading and indexing a KB before the arrival of the first query.

The test results in Figure 4 makes clear that a systems performance can be improved through caching in the middleware layer. The first segment of the graph shows that both systems behave very similar for the first 6 queries, which is when all queries are unknown to both systems. This shows that the introduction of a middleware cache has no negative impact on the answering time of queries that can not profit from it.

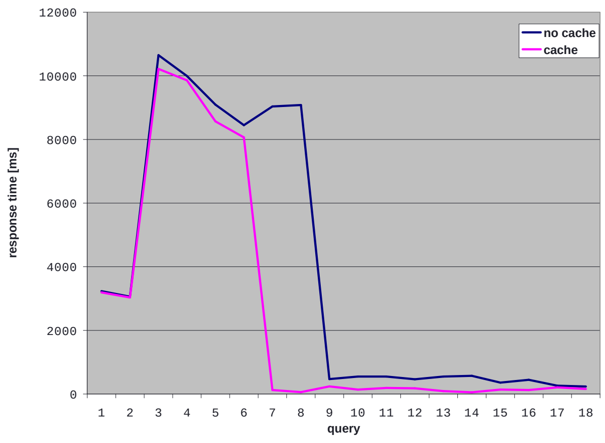

Fig. 4. The performance of the systems with and without a cache at the middleware layer

The next segment of the graph, namely, between the queries 7 and 12, exhibits the results if known queries arrive a for a second time. Queries in the un-cached system are forwarded to a reasoner and have to wait for the previous queries to be answered at the local reasoner queue, thus experience long response times. The middleware-cached system avoids these waiting times in the local reasoner queues by answering the queries immediately from the own cache. Our evaluations showed that the response times can be decreased by the factor 60 with the current implementation [2].

Moreover, the advantage of middleware caching is also visible in situations where both systems have empty queues. In the last segment of the graph the same set of

[8] At a first glance assuming a client to send the same query 3 times may seem unlikely. However, this is equal to the situation where 3 different clients send the the same query independently.

known queues arrive the middleware for the third time, where both systems are able to deliver all answer tuples from their caches and have no waiting times are piled up in the queues. Due to the communication overhead (between the middleware and the reasoners) and the additional processing time of the queries at the reasoners, the system without middleware caching is 3 times slower on average.

Note that both systems are tested on the same multiprocessor machine, where all system components are deployed on dedicated processors, and hence the communication costs are very low due to remarkably small network latency. Therefore, the difference observed in the performances of both systems will dramatically increase, if the reasoners are deployed on different physical machines than the middleware. In practice, most systems, especially those with a multi-layered architecture, have their components deployed on multiple servers, e.g., because of security restrictions or availability requirements.

6 Concluding remarks

In this paper we presented a scalable and efficient middleware that can utilize existing DL systems for retrieval inference services in the context of OWL-based applications. We have specified how several optimization criteria, in particular a load balancing strategy tailored towards the scenario-specific requirements of practical applications, can be exploited to achieve better scalability and high-availability.

Furthermore, we evaluated the results of practical experiments on a particular implementation of the middleware and showed that (in the context of ontology-based applications that require retrieval inference services) investing more time and computational resources in the analysis of queries pays off, if the results of the analysis are exploited for optimization. This insight is also the main contribution of this paper. Note that we argue that this is not only valid for RacerPro but also for other DL reasoners.

Experiments with the query subsumption optimization module, which we could not represent in this paper due to space limitations, showed that utilizing this module is not advantageous in every test setting. Besides a more detailed analysis of this module, our future work will investigate further optimizations for the middleware to meet the requirements of practical applications.

7 Acknowledgments

This work is partially supported by the EU-funded projects BOEMIE (Bootstrapping Ontology Evolution with Multimedia Information Extraction, IST-FP6-027538) and TONES (Thinking ONtologiES FET-FP6-7603). Tobias Berger has been supported by a DAAD (German Academic Exchange Service) scholarship.

References

1. 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.
2. Tobias Berger. Implementation, Test and Evaluation of Load Balancing Strategies for Multi-User Inference Systems, 2007. Available at `http://www.sts.tu-harburg.de/pw-and-m-theses/2007/berg07.pdf`.
3. Tony Bourke, editor. *Server Load Balancing.* O'Reilly, 2001.
4. R. Fikes, P. Hayes, and I. Horrocks. OWL-QL - A Language for Deductive Query Answering on the Semantic Web. Technical Report KSL-03-14, Knowledge Systems Lab, Stanford University, CA, USA, 2003.

5. John Grant and Jack Minker. A logic-based approach to data integration. *Theory Pract. Log. Program.*, 2(3):323–368, 2002.
6. V. Haarslev and R. Möller. RACER System Description. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning, IJ-CAR'2001, June 18-23, Siena, Italy*, pages 701–705. Springer-Verlag, 2001.
7. Haarslev Volker, Moeller Ralf, Wessel Michael. The New Racer Query Language-nRQL. 2005.
8. Apache JMeter Application, 2007. Available at `http://jakarta.apache.org/jmeter/`.
9. Chandra Kopparapu, editor. *Load Balancing Servers, Firewalls, and Caches.* Wiley, 2002.
10. Chen Li, Ramana Yerneni, Vasilis Vassalos, Hector Garcia-Molina, Yannis Papakonstantinou, Jeffrey Ullman, and Murty Valiveti. Capability based mediation in TSIMMIS. *SIGMOD Rec.*, 27(2):564–566, 1998.
11. LUBM: The Lehigh University Benchmark, 2006. Available at `http://swat.cse.lehigh.edu/projects/lubm/`.
12. Thomas Stuetzle Marco Dorigo, editor. *Ant Colony Optimization.* The MIT Press, 2004.
13. Ralf Möller, Volker Haarslev, and Michael Wessel. On the Scalability of Description Logic Instance Retrieval. In *Proc. of the 2006 International Workshop on Description Logics DL'06*, 2006.
14. Anni-Yasmin Turhan, Sean Bechhofer, Alissa Kaplunova, Thorsten Liebig, Marko Luther, Ralf Moeller, Olaf Noppens, Peter Patel-Schneider, Boontawee Suntisrivaraporn, and Timo Weithoener. DIG 2.0 – Towards a Flexible Interface for Description Logic Reasoners. In B. Coence Grau, P. Hitzler, C. Shankey, and E. Wallace, editors, *OWL: Experiences and Directions 2006*, 2006.