Towards a Scalable and Efficient Middleware for Instance Retrieval Inference Services

Tobias Berger, Alissa Kaplunova, Atila Kaya, Ralf Möller

Hamburg University of Technology
{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 [2]. Experiences show that many practical applications require reasoning, especially instance retrieval inference services[1], on large knowledge bases (KBs).[2] Furthermore, applications request a reasonable instance retrieval performance even if the number of retrieval requests increases. In [14] 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, many 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 scalability 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 [12] and others [7]. 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., [4], [11]). 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 DL queries requires reasoning and therefore is known to be a more complex task than query answering in conventional database systems. We argue that, in order to solve the scalability problem of DL systems, investing more time and computational resources in the analysis of queries pays off, if the results of the analysis

[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.

are exploited for finding better optimizations. 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, and presents results.

2 Requirements for the Middleware

When building practical OWL-based applications, the efficient interaction between clients and DL systems for information retrieval is still a big challenge for modern reasoners (e.g., [15], [9], [1], [8]). Although research on optimization techniques for standalone reasoning systems reported substantial performance improvements recently (e.g., [14]), 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 Highly-optimized reasoners can achieve significant performance improvements for a single client. For scenarios incorporating multiple clients posing queries concurrently, reasoners can be enhanced to use multithreading for processing client requests in parallel. This solution requires for each request a dedicated thread that has a temporary storage for processing the request, an adequate locking strategy to share the data between the threads and hence adds an amount of complexity to the reasoner implementation, which can lead to significant performance decreases when handling concurrent client requests. Alternatively, DL reasoners can be integrated with existing software tools, which are successfully used in a wide range of applications for concurrency and scalability. In an integrated system, these tools can be used to constitute a middleware, i.e., an additional application layer on top of reasoner(s), that employs sophisticated dispatching algorithms to delegate requests to multiple back-end DL reasoners. Consequently, through the utilization of established standard software tools, the reasoner implementation can be prevented from getting more complicated, which is the main advantage of a middleware-based solution.

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 [6]. 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 other features, OWL-QL allows clients to pose conjunctive queries and request partial sets of answers.[3] Modern reasoners (e.g., RacerPro [8]) support the retrieval of partial sets of answers, a.k.a. iterative query answering. 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 [8]. Consequently, besides supporting OWL-QL, 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] Note 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 degree of availability) can be achieved by increasing the number of reasoners managed by the middleware. In situations where the middleware may itself become a bottleneck, a possible solution is to replicate it and 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.

It is obvious that in scenarios where queries are only send sequentially, the middleware proposed here, which is allowed to analyze but not manipulate the query, cannot accelerate query answering. However, if queries are posed concurrently, which is common practice in current applications such as multimedia content management systems, 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 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 technique for system efficiency. All standard software systems that 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, we claim that it is 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 interactions with reasoners. Consequently, back-end reasoners are less often blocked and can be instructed to answer other queries (higher availability). 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 in DL systems is 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 probability of selecting an appropriate 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 have been asked before (and thus are 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.

Depending on the classification of a query, the middleware can decide whether to dispatch it directly to a particular reasoner or to analyze it further. For example, the middleware can dispatch a *continuation query* to the reasoner that already returned some answer tuples for that query. If a query is classified as a *new query to a known KB* the middleware can analyze it further by checking for query subsumption.

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 necessary 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 [4]. Despite

the proved success of existing load balancing strategies in present systems [11], 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 availability of the middleware for parallel ontology-based retrieval.

4 Load Balancing Strategy

The optimization criteria discussed in the previous section can be considered (and implemented) as different optimization modules of the middleware. The query classification module analyzes every incoming query. Depending on the classification, the middleware can consult further optimization modules. Every module can analyze a query w.r.t. an optimization criteria and is aimed at proposing the most appropriate reasoner to answer the query. However, it is not always possible to determine a unique candidate for query answering. In several situations an optimization module may propose several reasoners as equally good candidates or it may fail to propose any reasoner, because a comparison is impossible. In such cases the middleware can consult the load balancing module 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 the middleware without consulting the load balancing module. Therefore any action effecting the load of a back-end reasoner has to be registered to enable monitoring of load distribution at any time.[4]

Following these ideas, we developed a load balancing algorithm and named it ACO-LB, which is inspired by the Ant Colony Optimization algorithm (ACO, [5]). 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 back, an ant marks the way it took with so-called pheromones. The chemicals can be sensed by the following ants. In case there are several paths to the food, the shortest path to the food will soon have the highest concentration, due to the fact that the ants that chose the shortest path will return faster so that less pheromones will evaporate in this path.

With the help of artificial pheromones this behavior is imitated by our load balancing strategy (ACO-LB). 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 an estimation, the real value can be measured later. The estimated value is obtained by calculating the average of real values that have been measured for 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 least engaged reasoner to answer the current query. Due to space limitations, we present only the intuition; for details, please refer to [3].

The strategy provides three main benefits. First, it allows for easy reflection of all queries that are answered by a reasoner, both the queries that were distributed by the

[4] Note that this information is also essential for the KB distribution module.

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 pheromones and normalizing afterwards, 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 queries in OWL-QL[5] in parallel, by dispatching the queries to back-end reasoners. It can manage a configurable number of reasoner (RacerPro) instances. Furthermore, optimization modules can be activated (or deactivated) as necessary in order to evaluate each optimization in isolation. Due to space limitations we can not present the design and architecture of the system in detail here.

The load balancing strategy and other optimization modules are evaluated w.r.t. query response times that are measured in various benchmarks. Each benchmark represents a scenario, which is described by a test plan, containing the number of clients participating in the scenario, the OWL-QL queries each client sends, the number of answers required by each query, the order in which these queries are sent, and the ontologies that are addressed.[6] In order to define sample ontologies with a large number of individuals we utilized the tools provided by the Lehigh University Benchmark (LUBM, [13]). Besides the URLs used to reference them, all sample ontologies are identical so that the times measured are comparable. The queries used in the tests are LUBM queries translated into OWL-QL, each of which references exactly one of of the sample university ontologies.[7]

All tests are performed on a Sun Solaris server with 8 processors and 32 GB shared memory in order to minimize the impact of network latency on measurements. The open source load and performance measurement tool Apache JMeter [10] is used to execute test plans and measure query response times. During the tests the middleware is configured to manage different number of RacerPro systems in the version 1.9.0. Additionally, the middleware incorporates a RacerPro system that is dedicated to query subsumption checks. Note that RacerPro supports (incomplete) query subsumption checks [8]. Due to space limitations, we present only a small extract of results here; for details, please refer to [3].

Evaluation of Load Balancing To evaluate load balancing, we compared three different systems against each other: the first one exploiting the round robin strategy for load balancing, the second one exploiting ACO-LB and finally a system without any

[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`

load balancing strategy.[8] To simulate a system without any load balancing strategy, we configured our middleware to manage a single back-end reasoner. 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.

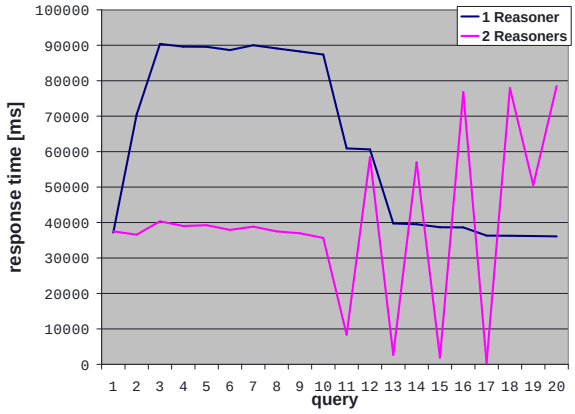

Fig. 1. Response times for the round robin strategy with one and two reasoners.

First, we compared a load balancing system that exploits the round robin strategy (and manages two back-end reasoners) with a system without any load balancing strategy (1 reasoner). Figure 1 shows the development of query response times for both systems.[9] At first sight it may seem surprising that for some queries the round robin strategy performs worse than a system without load balancing. This behavior is the result of the nature of the standard round robin strategy: it dispatches queries very fast but without analysis. Figure 1 reveals that the standard round robin strategy shows an unpredictable behavior and leads to higher response times than a system with no load balancing for some queries.

Later, we compared a system that manages two back-end reasoners and exploits ACO-LB with a system without any load balancing strategy (1 reasoner) using the same tests. Figure 2 shows the development of query response times for both systems. Queries are again sequenced in the same order in which they arrive the systems. We can see that for every query the system with the ACO-LB strategy achieves better response times than the system without any load balancing. It benefits from dispatching the KBs to different reasoners in the beginning and parallelizing query answering later. Furthermore, Figure 2 also reveals that if the number of back-end reasoners is increased, our middleware retains its availability and improves its query answering performance, and hence, can be used to build a scalable system.

Finally, average response times of all different systems are illustrated in Figure 3. It shows that the system with the ACO-LB strategy outperforms not only the system with a single reasoner, i.e., with no load balancing, but also the system with the round robin strategy.

[8] Note that cache, query subsumption and KB distribution modules are deactivated in these tests.

[9] Queries are sequenced in the same order in which they arrive the systems.

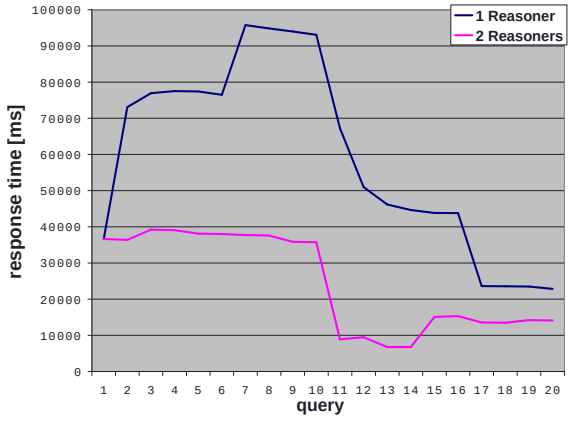

Fig. 2. Response times for the ACO-LB strategy with one and two reasoners.

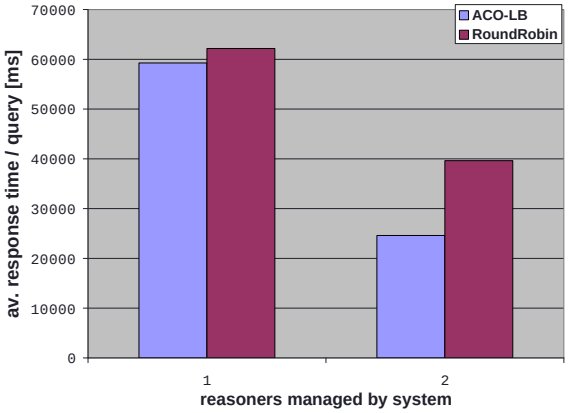

Fig. 3. Average response times of systems with different load balancing strategies.

Evaluation of Middleware Caching In order to verify the claim that caching at middleware layer is more efficient than local caching at back-end reasoners we defined a second benchmark.[10] 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.[11] Half of the clients pose queries to one KB and the other half to the other one. The queries of clients referencing the same KB are different. One of the systems exploits caching at middleware layer and the other one caching at back-end reasoners only. In this benchmark both systems are provided with enough space for caching. 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 results shown in Figure 4 make clear that a systems performance can be improved through caching in the middleware layer. Note that the graph shows the

[10] Note that query subsumption, KB distribution and load balancing modules are deactivated in these tests.

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

development of response times in the same order in which the queries are answered by the systems. The first segment of the graph (queries 0-6) shows that both systems behave very similarly for the first 6 queries, i.e., if unknown queries arrive. This shows that the introduction of a middleware cache has no negative impact on the answering time of queries that can not benefit from caching.

The next segment of the graph (queries 6-12) exhibits the results if known queries arrive for a second time. In the system without middleware caching the queries 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].

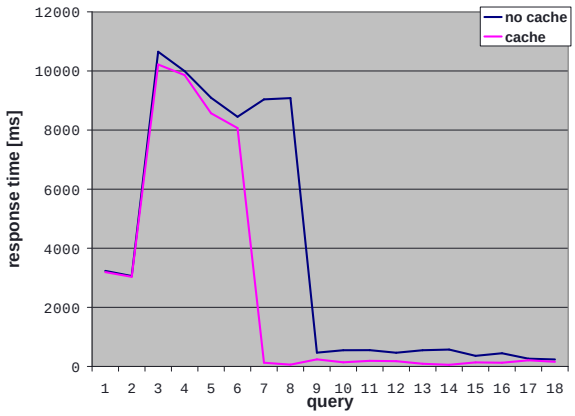

Fig. 4. Response times of systems with and without caching at middleware layer.

Moreover, the advantage of middleware caching is also visible in situations where both systems have empty queues. In the last segment of the graph (queries 12-18) the same set of known queries arrives both systems for the third time. In this case both systems are able to deliver all answer tuples from their caches and both have empty 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 with a large shared memory, where the communication costs between system components are very low due to remarkably small network latency. Therefore, the difference observed in the performances of both systems will dramatically increase, if the back-end 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 discussed several optimization criteria, in particular a load balancing strategy tailored towards scenario-specific requirements of practical applications, that can be exploited to achieve better scalability and 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 time and computational resources in the analysis of queries pays off, if results of the analysis are exploited for optimization later. This insight is also the main contribution of this paper.

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 advantageous only in some test settings. 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.

Acknowledgments

This work has been 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. A. Acciarri, D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, M. Palmieri, and R. Rosati. QuOnto: QUerying ONTOlogies. In *Proc. of the 20th Nat. Conf. on Artificial Intelligence (AAAI 2005)*, pages 1670–1671, 2005.
2. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
3. T. 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`.
4. T. Bourke, editor. *Server Load Balancing*. O'Reilly, 2001.
5. M. Dorigo and T. Stuetzle, editors. *Ant Colony Optimization*. The MIT Press, 2004.
6. 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.
7. J. Grant and J. Minker. A logic-based approach to data integration. *Theory Pract. Log. Program.*, 2(3):323–368, 2002.
8. V. Haarslev, R. Möller, and M. Wessel. RacerPro User's Guide and Reference Manual Version 1.9.0. April 2007.
9. U. Hustadt, B. Motik, and U. Sattler. Reducing $\mathcal{SHIQ}$-Description Logic to Disjunctive Datalog Programs. In *Proc. of the 9th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2004)*, pages 152–162, 2004.
10. Apache JMeter Application, 2007. Available at `http://jakarta.apache.org/jmeter/`.
11. C. Kopparapu, editor. *Load Balancing Servers, Firewalls, and Caches*. Wiley, 2002.
12. C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. Ullman, and M. Valiveti. Capability based mediation in TSIMMIS. *SIGMOD Rec.*, 27(2), 1998.
13. LUBM: The Lehigh University Benchmark, 2006. Available at `http://swat.cse.lehigh.edu/projects/lubm/`.
14. R. Möller, V. Haarslev, and M. Wessel. On the Scalability of Description Logic Instance Retrieval. In *Proc. of the 2006 Intern. Workshop on Description Logics DL'06*, 2006.
15. E. Sirin and B. Parsia. Pellet System Description. In *Proc. of the Int. Workshop on Description Logics, DL '06*, 2006.