Incremental Query Answering for Implementing Document Retrieval Services

Volker Haarslev[†] and Ralf Möller[‡]

Concordia University, Montreal[†]
[‡]University of Applied Sciences, Wedel

Abstract

Agent systems that search the Semantic Web are seen as killer applications for description logic (DL) inference engines. The guiding examples for the Semantic Web involve information and document retrieval tasks. The instance retrieval inference service of description logic inference engines can be used as a basic machinery for implementing agent-based retrieval systems. However, since information is permanently added to information sources, usually agents need to return to previously visited servers in order to get updates for their queries over time.

In this paper we present a software architecture that allows agents to register instance retrieval queries at a certain inference server. We will see how agents are notified when the result set of registered queries grows over time. The paper describes new optimization techniques for incrementally computing answers for sets of registered instance retrieval queries and reports on first experiences with an implementation as part of the Racer system.

1 Motivation

Agent systems that search the Semantic Web are seen as killer applications for description logic (DL) inference engines. The guiding examples for the Semantic Web involve information and document retrieval tasks. Agents hopping from node to node try to gather information relevant for some topic of interest. This paper considers an application scenario where documents are represented by A-box individuals associated with certain roles and attributes for describing the documents. Then, the instance retrieval inference service of description logic inference engines can be used as a basic machinery for implementing agent-based document retrieval systems. Queries for documents are simply specified by concepts. Assuming an agent gets a set of document names as a result, it records their attributes (and attribute values) accordingly and then leaves the node (server) for seeking additional areas of the Semantic Web. However, since information is permanently added to information sources, usually agents need to return to previously visited servers in order to get updates for their queries over time. Experiences with existing systems indicate that permanent polling operations use more resources than appropriate. In addition, if queries, possibly from

different applications, are gathered in a query set, an inference server can derive an optimal query answering strategy. Query registration is the basis for applying these techniques.

In this paper we present a software architecture that allows agents to register instance retrieval queries at a certain inference server. We will see how agents are notified when the result set to registered queries grows over time. The paper describes new optimization techniques for incrementally computing answers to sets of registered instance retrieval queries and reports on first experiences with an implementation as part of the Racer system [3].

2 The Racer Server and Racer Proxy

The Server is an application program which can read knowledge bases either from local files or from remote Web servers (i.e., a Racer Server is also an HTTP client). In turn, other client programs that need inference services can communicate with a Racer Server via TCP-based protocols. OilEd can be seen as a specific client that uses the DIG protocol [1] for communicating with a Racer Server, whereas RICE is another client that uses a more low-level TCP protocol.

In the context of multiple graphical interfaces and multiple client programs (e.g., agents), a description logic inference server such as Racer will be used by more than one thread. Thus, a serious server has to coordinate multi-user access in a similar way as database systems do. For dealing with multiple clients, the Racer Server includes a subsystem called the Racer Proxy. The Racer Proxy is started as a front-end to an associated Racer Server. It is configured to use a port number for external client access in the same way as a Racer Server. The port number of the associated Racer Server must be specified at proxy startup time. Then, a Racer Proxy is accessed just like a Racer Server, and it just forwards requests to a corresponding Racer Server.

The task of the Racer Proxy is to provide locks for inference services of the Racer Server that it "manages" such that instructions and queries of multiple clients are properly coordinated. For instance, queries for concept parents by one client are delayed until the T-box is entirely classified in order to answer a query of another client etc. A Racer Proxy can also be configured to use multiple Racer Servers for load balancing (for details see the Racer Manual [6, Version 1.7.7 and later]). Racer Proxy is written entirely in Java and is provided with source code for research purposes. The Racer Proxy is also responsible for implementing the Racer publish and subscribe interface, which is described in the next section.

3 A Document Retrieval Use Case

We assume that documents, for instance in PDF format, provide containers for meta information in RDF format [9]. RDF structures representing meta information about documents can be submitted to a Racer Server. Usually, Racer is only told the URL of those documents. Given the URL, the Racer Server fetches RDF documents from Web servers if appropriate. In particular, it is also possible to refer to a certain on-

tology in DAML [11] or OWL [10] format. Racer reads these ontologies from Web Servers if required and represents ontology information as a T-box. Racer can read DAML+OIL and OWL documents. Racer accepts the so-called OWL DL subset [10] with some additional restrictions (such as approximated reasoning for nominals (see [6] for details) and unique name assumption). DAML+OIL documents are interpreted with the same restrictions as manifested in OWL DL (the sets of classes and instances are disjoint, no reified statements, no treatment of class metaobjects etc.). Descriptions in RDF documents (with OWL DL restrictions) are represented as A-boxes by the Racer System (for details see the Racer User's Guide [6]). For readability reasons, however, in this paper we use the standard description logic syntax for examples using the description logic $\mathcal{SHIQ}(\mathcal{D}_n)^-$.

For the document retrieval use cases to be discussed in this section we assume an ontology (T-box) *Document_ontology* containing the following axioms (an attribute is a function whose range can be chosen from a concrete domain):

$attribute(isbn, integer)$, $attribute(n_copies_sold, integer)$,
$Book \sqsubseteq Document$,
$Article \sqsubseteq Document$,
$Computer_Science_Document \sqsubseteq Document$,
$Computer_Science_Book \sqsubseteq Book \sqcap Computer_Science_Document$,
$Compiler_Construction_Book \sqsubseteq Computer_Science_Book$,
$min(n_copies_sold, 3000) \sqcap Computer_Science_Document \sqsubseteq$
$\quad Computer_Science_Best_Seller$

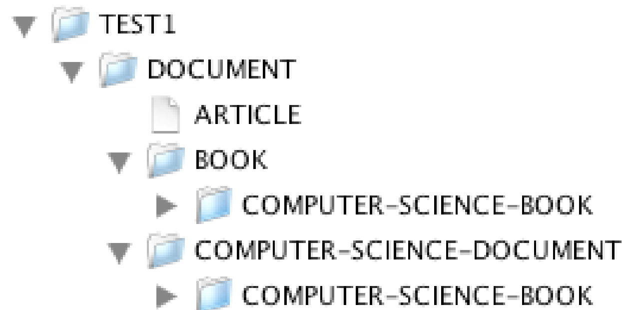

Figure 1: Graphical display of the T-box taxonomy shown by RICE.

In Figure 1 a graphical representation of the taxonomy of the T-box is shown. Now we consider an A-box providing information about particular documents. Stemming from RDF meta information about documents in PDF format, document descriptions are gathered in an A-box *Current_documents*. We assume a state in which the following assertions are given[1]:

[1]Note that in Racer strings for authors names, say, could be represented using concrete domains.

$document_1 : Article$,
$(document_1, |John\ Smith|) : has_author$

$document_2 : Book$,
$(document_2, |John\ Doe|) : has_author$,
$(document_2, isbn_2) : has_isbn_number$, $equal(isbn_2, 2234567)$,

$document_3 : Book$,
$(document_3, |Otto\ Normalverbraucher|) : has_author$,
$(document_3, isbn_3) : has_isbn_number$, $equal(isbn_3, 3234567)$,

In the document retrieval scenario discussed in this paper, queries for documents of some kind are specified using concepts. The retrieval service is implemented using the instance retrieval inference service offered by a Racer Server. Queries are answered with respect to the A-box *Current_documents* and T-box *Document_ontology* (see above).

It is obvious that in the document retrieval scenario introduced above, only some individuals are expected to be in the result set of an instance retrieval query. These "root" individuals are the documents in our scenario, e.g. *document_1*. For other individuals, e.g. for authors such as $|John\ Smith|$, it is known in advance that they are only accessed as role fillers of root individuals. Therefore, it is possible to explicitly indicate so-called "public" individuals. This is called publishing an individual in Racer terminology. For the example A-box we assume that the following statement is executed: $publish(\{document_1, document_2, document_3\})$

Published individuals can be returned in the result sets of specific instance retrieval queries (for other queries, e.g. role filler retrieval, publication is not relevant). Using the Racer system, clients can subscribe to a "channel" on which individuals are announced that are instances of a given query concept. As an example we consider an agent subscription named q_1 with query concept *Book* and server `"mo.fh-wedel.de"` running at port 8080. The A-box is called *Current_documents*, and it implicitly refers to the ontology *Document_ontology*.

The following code fragment demonstrates how to interact with a Racer Server from a Java application. The aim of the example is to demonstrate the relative ease of use that such an API provides. In our scenario, we assume that the agent instructs the Racer system to direct the channel to computer `"mo.fh-wedel.de"` at port `8080`. Before the subscription is sent to a Racer Server, the agent should make sure that at `"mo.fh-wedel.de"`, the assumed agent base station, a so-called listener process is started at port 8080. This can be easily accomplished:

```
public class Listener {
   public static void main(String[] argv) {
      try {
        ServerSocket socketServer = new ServerSocket(8080);
```

However, in the context of this paper, we use individuals for authors. Individuals can be surrounded by bars if, for instance, the names include spaces.

```
        while (true) {
           Socket client = socketServer.accept();
           DataInputStream in =
              new DataInputStream(
                 new BufferedInputStream(client.getInputStream()));
           byte b[] = new byte[1024];
           int num = in.read(b);
           String result = new String(b);
           ...
          }
      } catch (IOException e) {
          ...
      }
   }
}
```

If a message comes in over the input stream, the variable `result` is bound accordingly. Then, the message can be processed as suitable to the application. We do not discuss details here. The subscription to the channel, i.e., the registration of the query, can also be easily done using the JRacer interface as indicated with the following code fragment (we assume Racer runs at node `"racer.fh-wedel.de"` on port 8088).

```
public class Subscription {
   public static void main(String[] argv) {
      RacerClient client = new RacerClient("racer.fh-wedel.de", 8088);
      try {
         client.openConnection();
         try {
            String result =
               client.send
                     ("(subscribe q_1 Book" +
                      " Current_documents" +
                      " (:notification-method tcp mo.fh-wedel.de 8080))");
            }
         catch (RacerException e) {
            ...
            }
         }
         client.closeConnection();
      } catch (IOException e) {
         ...
      }
   }
}
```

The connection to the Racer server is represented with a client object (of class `RacerClient`). The client object is used to send messages to the associated Racer server (using the message `send`). Control flow stops until Racer acknowledges the subscription.

In our example we consider the query q_1. After the query q_1 is registered, Racer running at node "racer.fh-wedel.de" sends the following message string to "mo.fh-wedel.de" listening on port 8080: "((q_1 *document_2*) (q_1 *document_3*))". The message is stored in the variable `result`. Of course, the client is responsible for interpreting the result appropriately. In our agent scenario we assume that two documents, *document_2* and *document_3*, are recorded as possible hits to the query (possibly together with retrieved values for the ISBN number etc.), but we do not go into details here.

Next, we assume that the document information repository represented by the A-box *Current_documents* is subsequently filled with information about new documents.

$document_4 : Computer_Science_Document,$
$(document_4, isbn_4) : has_isbn_number,\ equal(isbn_4, 2234567)$

Much less is known about *document_4*, and after publishing it, nobody will be notified. Although there is a subscription to a channel for *Book*, it cannot be proven that *document_4* is an instance of this concept.

Now we assume there are two additional subscriptions q_2 and q_3 to the concepts *Computer_Science_Document* and *Computer_Science_Best_Seller*, respectively. For query q_2 Racer immediately generates a message ((q_2 *document_4*)) and redirects it to the channel specified with the subscription statement. However, for q_3 no message can be generated at subscription time. As time evolves, it might be the case that the number of copies sold for *document_4* become known. The A-box *Current_documents* is extended with the following assertions:

$(document_4, n_copies_4) : n_copies_sold,\ equal(n_copies_4, 4000)$

Since the information in the A-box now implies that *document_4* is an instance of the query concept in subscription q_3, the client is notified accordingly using the same techniques as discussed above.

The example sketches how description logics in general, and the publish and subscribe interface of Racer in particular, can be used to implement a document retrieval system (for additional examples and details on the publish and subscribe interface see the Racer User's Guide [6]). In the next section we describe optimization techniques for incrementally answering instance retrieval queries such that the vision of the above-mentioned application scenario can become reality with description logic inference technology.

4 Optimization Techniques

In description logic systems, usually all individuals mentioned in a A-box are subject to be results of instance retrieval queries. This is perfectly sanctioned by the logical semantics, However, in practical contexts in many cases only a set of strategic "root" objects are interesting for clients to retrieve. Given a root object, other individuals

are usually accessed by retrieving fillers for certain roles. In our document retrieval scenario, the root objects are documents, and other individuals (e.g., authors) are only retrieved as filler of roles for documents. The publish and subscribe interface of Racer allows for marking the root objects as published objects. In particular, computing an index for document retrieval (know as realizing the A-box) can be limited to published individuals only. Although being a pragmatic solution, the publication mechanism allows for a tremendous reduction of workload for a description logic inference system.

Indeed, effective candidate reduction with few resources is mandatory in practice to achieve adequate performance. Incremental query answering can also help here. Once a set of instance retrieval queries is registered, old results can be stored. Thus, for checking whether there are new elements in the result sets of registered queries when an A-box is extended, previous individuals can be easily discarded from the initial set of candidates. This is possible because of the incremental way of answering queries as built into the Racer Server. It would be much harder to achieve if agents were forced to continuously poll the information system because the Racer Server has to guess which query results should be cached.

Since Racer can operate on sets of registered queries, queries can be ordered with respect to subsumption. Given the partial order induced by the subsumption relations, an optimal execution sequence for answering multiple queries can be generated with a topological sorting algorithm. The more general queries are processed first, yielding a (possibly reduced) set of candidates for more specific queries as a by-product. In [5] we have introduced Dependency-based Binary Partitioning Search as a strategy for answering instance retrieval queries (i.e., instance retrieval with and without A-box realization). In addition, A-box optimization techniques such as individual model merging are introduced in [7]. The publish and subscribe interface of Racer exploits these query answering mechanisms.

A short case study demonstrates the importance of ordering queries w.r.t. the subsumption relation. Given the T-box defined above we consider an A-box with the following assertions (for n we use different settings):

$doc_1 : Article,$
$doc_2 : Article,$
$\ldots$
$doc_n : Article,$
$doc_n + 1 : Book,$
$doc_n + 2 : Book,$
$\ldots$
$doc_n + n : Book,$
$doc_n + n + 1 : Computer_Science_Book,$
$doc_n + n + 2 : Computer_Science_Book,$
$\ldots$
$doc_n + n + n : Computer_Science_Book$

All individuals mentioned above are assumed to be published, and Racer is instructed to answer instance retrieval queries by exploiting results gained from T-box classifica-

tion (subsumption-based query answering).

For demonstrating the importance of ordering queries, we consider the following query set: $\{retrieve(Book), retrieve(Computer_Science_Book)\}$. There are two strategies, either all instances of *Computer_Science_Book* are retrieved first (Strategy 1) or all instances of *Book* are retrieved first (Strategy 2).

Table 1: Runtimes (in secs) of instance retrieval query sets with different strategies.

n	Gen. Time	ASAT	Strategy 1	Strategy 2
10000	1	6	7	5
20000	3	10	29	19
30000	22	15	79	42
40000	34	23	164	115
50000	54	34	320	200
60000	80	42	904	552

The runtimes of the query sets under different strategies are indicated in Table 1. In the first column the number n is specified (note that the A-box contains three times as many individuals), in the second column the time to generate the problem is given (i.e., the time to "fill" the A-box), in the third column the time for the initial A-box consistency test is displayed, and in the last two columns the runtimes for the different strategies are indicated. All tests were performed on a 1GHz-512MB Powerbook.

Table 1 reveals that for larger values of n, Strategy 2, i.e., to first retrieve all instances of the superconcept *Book*, is approximately twice as fast as Strategy 1. The reason is that with Strategy 2 the set of candidates for the second instance retrieval query can be considerably reduced. The publish and subscribe facility provides the basis for building query sets that can be answered using the above-mentioned optimization strategies.

5 Realizing Local Closed World Assumptions

Feedback from many users of the Racer system indicates that, for example, instance retrieval queries could profit from possibilities to "close" a knowledge base in one way or another. Due to the non-monotonic nature of the closed-world assumption and the ambiguities about what closing should actually mean, in description logic inference systems usually there is no support for the closed-world assumption. Theoretically, some approaches for this problem are already known (see e.g. [2]). However, from a practical point of view, there are no implemented systems available that users can rely on. With the publish and subscribe interface of Racer, users can achieve a similar effect. Consider, for instance, a query for a book which does not have an author. Because of the open-world assumption, subscribing to a channel for $Book \sqcap (\leq 0\, has_author)$ does not make much sense. Nevertheless the agent can subscribe to a channel for *Book* and a channel for $(\geq 1\, has_author)$. It can accumulate the results returned by

Racer into two variables S_1 and S_2, respectively, and, in order to compute the set of books for which there does not exist an author, it can consider the complement of S_2 wrt. S_1. We see this strategy as an implementation of a local closed-world (LCW) assumption.

However, as time evolves, authors for documents determined by the above-mentioned query indeed might become known. In others words, the set S_2 will probably be extended. In this case, the agent is responsible for implementing appropriate backtracking strategies, of course.

Obviously, a client application could also frequently poll a Racer Server for the different sets S_1 and S_2. With the publish and subscribe interface unnecessary queries can be avoided, and the Racer Server can optimize the computation of the result sets for different queries. The LCW example demonstrates that the Racer publish and subscribe interface is a very general mechanism, which can also be used to solve other problems in applications involving knowledge representation.

6 Conclusion

Integrated into Semantic Web and distributed systems software infrastructure, Racer offers a first experimental implementation of the publish and subscribe interface described in this paper. With Racer, first results on optimization techniques for query answering have been investigated such that distributed document retrieval systems can be built on top of description logic inference technology. Due to our experiences, current technology allows for several thousands of published documents to be handled in an A-box that refers to a T-box with axioms for ten thousands of concept names [4].

As a summary one can say that for almost all types of A-box instance retrieval queries, computing the taxonomy induced by the T-box enables important optimization techniques. $\mathcal{SHIQ}(\mathcal{D}_n)^-$ has the nice property that computations on T-box information need not to be repeated if an A-box is changed. With the advent of nominals in description logics such as $\mathcal{SHOQ}$ [8], future DL systems will have a hard time to figure out what can be reused from those data structures computed from T-boxes since concept classification depends on A-box information as part of the knowledge base.

Acknowledgments

The idea to the publish and subscribe interface of Racer was born in a discussion with Mike Ushold at the 2002 Description Logic Workshop in Toulouse. Furthermore, in a discussion with Ragnhild van der Straeten, the exploitation of the publish and subscribe interface for realizing local closed world assumptions became clear. The Racer Proxy was implemented by Christian Finckler (University of Applied Sciences, Wedel). Thanks to all of them. All shortcomings of this paper are due to our own faults, of course.

References

[1] S. Bechhofer, R. Möller, and P. Crowther. The DIG description interface. In *Proc. International Workshop on Description Logics – DL'03*, 2003.

[2] F.M. Donini, D. Nardi, and R. Rosati. Description logics of minimal knowledge and negation as failure. *ACM Transactions on Computational Logic*, 3, April 2002.

[3] V. Haarslev and R. Möller. Racer system description. In *International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, 2001, Siena, Italy.*, 2001.

[4] V. Haarslev and R. Möller. High performance reasoning with very large knowledge bases: A practical case study. In *Seventeenth International Joint Conference on Artificial Intelligence, IJCAI-01, August 4-10, 2001, Seattle, Washington, USA.*, 2002.

[5] V. Haarslev and R. Möller. Optimization stategies for instance retrieval. In *Proc. International Workshop on Description Logics – DL'02*, 2002.

[6] V. Haarslev and R. Möller. The Racer user's guide and reference manual, 2003.

[7] V. Haarslev, R. Möller, and A.Y. Turhan. Exploiting pseudo models for TBox and ABox reasoning in expressive description logics. In *International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, 2001, Siena, Italy.* Springer-Verlag, 2001.

[8] I. Horrocks and U. Sattler. Ontology reasoning in the SHOQ(D) description logic. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001.

[9] Adobe Systems Inc. Embedding XMP metadata in application files, 2002.

[10] F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference, 2003.

[11] F. van Harmelen, P.F. Patel-Schneider, and I. Horrocks (Editors). Reference description of the DAML+OIL (march 2001) ontology markup language, 2001.