Performance of Semantic Search

John Li, Adam Pease, Christopher Barbee
Teknowledge Corporation
Palo Alto, CA
{jli, apease, cbarbee} @teknowledge.com

Introduction

The DARPA Agent Markup Language (DAML) and its most recent version, DAML+OIL, have provided a promising web standard and a draft language to express ontologies on the web [1]. With the advent of this language, semantic search will become feasible and the vision of a semantic web will finally be realized.

A new semantic markup language is not enough without tools that support it. With the availability of semantic markup, we can do better than simple keyword search. True semantic search becomes possible. The Agent Semantic Communications Service (ASCS) allows users to perform high-quality semantic search in this DAML annotated web environment. ASCS consists of two main components. A Semantic Search Agent (SSA) allows agents to find entities based on the ontologies they share, and a Semantic Translation Service (STS) provides a basis for communication between agents that employ different ontologies. In addition, the architecture of the ASCS allows any text-based search agent to work together with it and have their joint results presented to the users.

For example, an application can use an SSA to query very detailed information annotated in DAML somewhere on the web, such as a telephone's extension number. The result is very accurate since the semantic contents of the numbers are clearly marked with the underlying DAML ontology. Other pages may employ a different ontology to mark up the same information in DAML. The STS does mapping of the two ontologies, and enables the SSA to locate that piece of information. This is called the "semantic interoperability". Current text-based search tools very often return nothing or a bundle of irrelevant pages only human can sort out.

We believe that in a mixed environment annotated with both DAML and non-DAML (as the web we currently have), an ASCS combined with any text-based search agent can perform better than the latter alone. This claim needs to be validated by experiments some of that have been completed and are described in this paper.

One challenge to our claims is whether there is a cost to the higher precision and recall provided by a DAML-enabled search agent. We do see that precision and recall in a text-based search environment usually come at the cost of other performance measurements, such as the response time. To investigate the cost of higher precision and recall by using ASCS semantic search agents, to find out where we need to improve our design and implementation of ASCS, and to guarantee the scalability of our approach, we recently had conducted a set of experiments. The purpose of this paper is to report these experimental results.

The rest of the paper is organized as follows: we will first introduce the architecture and implementation of the ASCS, then describe the experimental design in terms of the variables to be measured in the tests. The results will be presented and a discussion about their meaning and our future work will follow.

ASCS architecture and implementation

Our current experiments with ASCS only cover search on static repositories. We use XSB [3] servers as the static repositories for our SSAs. We invoke a crawler that parses DAML-encoded pages and creates indexes for the content. DAML markup is stored in these servers as Prolog assertions. An SSA (implemented in Java) receives a query in DAML/RDF format and converts it to a Prolog query. The XSB server then searches its repository and returns to the SSA the bindings for the variables in the query (see Figure 1: An example of ASCS architecture). We will discuss the need for a DAML query language later.

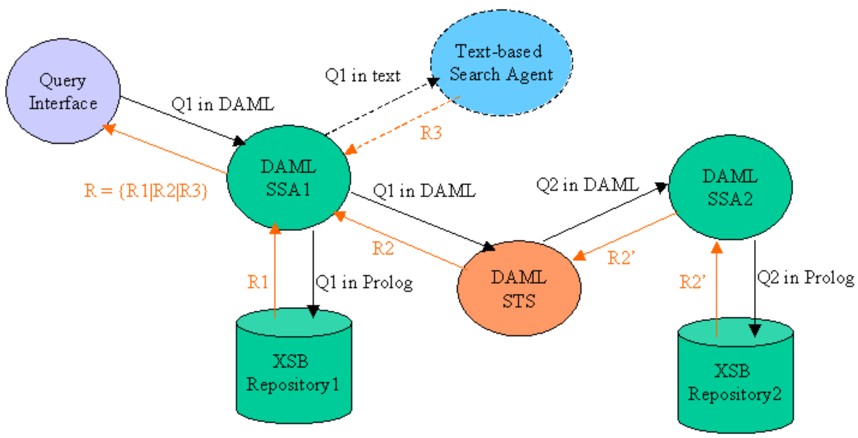

Figure 1. An example of ASCS architecture.
Q1 and Q2 are queries. R1, R2, R2' and R3 are responses.
The STS reformulates Q1 into Q2 that SSA2 can understand and translates the response R2' into R2 that SSA1 can understand. R is the response that merges R1, R2 and R3.

When an SSA finds there is an STS available to its service, it also sends the query to the STS, in order to expand the opportunity of finding matches in other repositories annotated with different ontologies. The STS is a bridge between two ontologies: the starting ontology and the destination ontology. We implemented the STS as an XSB translator program with a Java wrapper. The wrapper receives and converts the query in DAML/RDF format into a specific Prolog format in that variables and constants can be easily distinguished and translated. The XSB translator program then reformulates the

Prolog query according to the destination ontology and returns the new query to its wrapper. The wrapper converts the returned query into DAML/RDF format again and sends it to an SSA known to perform search on the destination ontology. When the response is returned to the second SSA, the STS translates it back into the starting ontology by finding the bindings for the variables in the original query. Finally, the responses from these two paths (one from a search by a single SSA, and the other from a search via an STS and another SSA) and possibly a third response from some text-based search are merged and displayed to the user.

The number of SSA's and STS's can be multiplied in real applications. To avoid duplications of search and possible looping, mechanisms such as timeout limits, hop list and counts have been implemented. To most accurately reflect the capability of our ASCS agents, we did not employ any text-based search agent in our experiments.

Experiment setting and variables

We now detail the experiment design. We wanted to measure performance variation in terms of query response time. To do this, we used the following variables:

(1) The size of the repositories (integer metric): We first built a small ontology for the WordNet [2] lexicon. Then we used the ontology to mark up, in DAML, most English words in WordNet. This resulted in about 70,000 web pages. Each page contains the synonyms, antonyms, hypernyms, hyponyms and troponyms of an English word with DAML markup that gives some semantic for those links. Then these pages were crawled and parsed. A repository of about 1 million records was generated. To differentiate the performance of the ASCS in repositories of different sizes, we extracted four subsets from this one, with size of 1,007, 10,044, 100,033 and 1,000,004 records, respectively. Each repository is a subset of a larger one. This guarantees a query can be answered in all repositories if it can be answered in the smallest subset.

(2) The use of translation service (binary metric): We expected the usage of ASCS translation service, STS, would require more response time than the straightforward SSA search. We wanted to measure just the overhead incurred by the translation architecture. Translation itself may be quite costly and will be a focus of future experiments. To construct the second set of repositories for SSA2, we added a suffix "_Foo" to each constant in the WordNet repositories mentioned above. For this experiment we conducted two variants. In one, we had a query answered only by one SSA without the help of any STS. Then the same query was sent to an SSA1-STS-SSA2 architecture as shown in Figure 1 in which two SSAs, SSA1 and SSA2, perform searches on their respective repositories, annotated with two different ontologies. The query was first sent to SSA1 and then STS. SSA1 could answer the query directly with its repository while SSA2 could not. The STS reformulates the query into that of SSA2's and translates the response from SSA2 back into the ontology that SSA1 uses. The results from SSA1 and SSA2 were merged.

(3) The complexity of the query and the number of answers (integer metric). Some queries may take longer to answer simply because they are more complicated and have more answers. Therefore, we prepared two types of queries in the experiments: simple look-ups with single answer and conjuncts of the queries with multiple answers. For example, we asked a single-line straightforward look-up which just matches one record in the repository ('to find a sense of the word'). The more complex type of queries returns multiple answers to a conjunct of queries ('to find the synonyms of the word via senses').

(4) The number of machines (integer metric): The ASCS software allows multiple SSAs to be installed on a single machine or in a distributed environment to pass the query to each other or to STSs within a certain timeout limit and hop limit. We wanted to have an estimation of the costs involved in the distributed mode. Previous SSA1-STS-SSA2 architecture was split into two parts with SSA1 installed on one machine and SSA2 plus STS in another. The response time in this case was compared with the case when all these agents residing on a single machine.

Indexing methods on the repositories are usually a factor in the search performance. Here we used the standard XSB indexing method for all the repositories. Therefore, it is not a measurement in our experiments.

Results

We got all the responses to our queries as accurate as we expected. We run the simple query ('find the senses of word X') and the complex query ('find the synonyms of the word X via its senses') in a single SSA setting, a single machine SSA1-STS-SSA2 setting, and a two machine SSA1-STS-SSA2 settings. For both the simple and the complex queries, we got exactly the expected sets of answers for the queried words in a single SSA setting, and duplicated sets of answers when the STS was involved.

The timing data for all the tests are summarized in the following table. Each piece of data represents an average response time for at least ten test queries in the same setting. As we mentioned before, each query contains one new word. Therefore, at least ten different words were used in each setting for the same query. Almost the same set of words was used for every setting. The unit of time in the table is ms. Note that the absolute timing of our results are machine dependent and, therefore, it is less meaningful to look at their absolute values than to compare their relative magnitudes. The machines we had conducted our experiments on are relatively low powered Linux machines. The Linux environment allowed the loading of our largest repository into the XSB server.

In the following table and chart we will use S1, S2 and S3 to denote three settings: a single SSA, an SSA1-STS-SSA2 set on a single machine and an SSA1-STS-SSA2 set on two machines, respectively. The suffix –s and –l represent simple queries and complex queries, respectively. For example, S1-s in the chart represents the data in the second column of the table while S2-l represents that in the sixth column of the table.

	Simple Queries (in ms)			Complex Queries (in ms)		
Size of DB	**S1**	**S2**	**S3**	**S1**	**S2**	**S3**
1,007 records	176	565	691	205	706	1,019
10,044 records	189	595	655	296	802	1,138
100,033 records	592	924	1,069	709	1,726	2,000
1,000,004 records	1,680	3,564	3,467	5,148	10,743	10,080

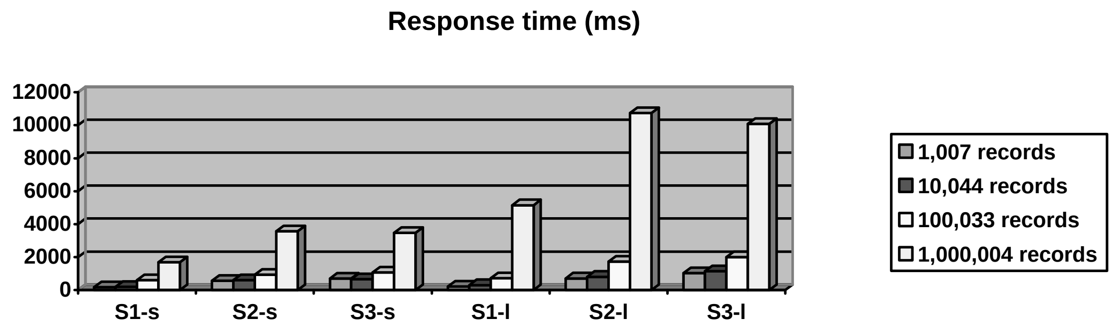

Table 1: Test data (in ms)

From Table 1, we have some inconclusive observations:

(1) There was not much difference in performances between the two smallest repositories (1,007 and 10,044), for all test settings and query types. That probably means their response time shown for each setting and query type is minimal and irreducible with regard to the repository size. In the future tests, we probably can eliminate one and only use 10,044 as the baseline.

(2) In general, the simple queries took shorter time to answer than the complex ones (ratio is 1:1.2~3.1). Furthermore, the response time difference between the two types grew larger as the repositories became larger. For example, for S2, 706 is about 1.3 times big as 565 in 1,007 records while 10,743 is about three times of 3,564 in 1,000,004 records.

	Simple Queries			Complex Queries		
Size of DB	**S1(in ms)**	**S2 (in ms)**	**Ratio**	**S1(in ms)**	**S2 (in ms)**	**Ratio**
1,007 records	176	565	3.2	205	706	3.4
10,044 records	189	595	3.1	296	802	2.7
100,033 records	592	924	1.6	709	1,726	2.4
1,000,004 records	1,680	3,564	2.1	5,148	10,743	2.1

Table 2: Ratios of response time when additional SSA and STS are used (all data in ms except the ratios).

(3) There is a cost for the translation of the queries and the additional responses (See Table 2). The response time for the single machine SSA1-STS-SSA2 settings is about two to three times of that for the single SSA setting (ratio is 1:1.6~3.4). An interesting fact is the ratio decreases as the repository size increases. That probably

means the cost of translation becomes less significant when the repository becomes large. It would also be interesting to know whether this overhead time would increase linearly with the number of STSs and SSAs involved. Further experiments are needed.

(4) The two-machine setting in general needed a tiny bit more response time than the single machine setting. We expected that the network overhead time might fluctuate based on the traffic load on the net and an accurate measurement of it is difficult. However, when the repository became too large (over 1million records), the opposite situation happened. We suspect this abnormality was caused by the hardware constraints of the test machine.

Simple Queries			**Complex Queries**		
S1	**S2**	**S3**	**S1**	**S2**	**S3**
2.8	3.9	3.2	7.3	6.2	5.0

Table 3: Ratios of response time from 100,033 records to 1,000,004 records

(5) When the size of the repository increased from 100,033 to 1,000,004 records, there was an across-the-board increase in the response time (ratio is 1:2.8~7.3. See Table 3). However, the response time for the complex queries grew more rapidly than the simple ones in all settings. The former has an increase ratio of 5.0 ~ 7.3 while that of the latter ranges from 2.8 to 3.9. The worst ratio (7.3) happened in the single SSA setting. This indicates that the degradation of performance might not be caused by the additional SSA or STS but might be due to the way the XSB server handles the complex queries or due to our test hardware constraints. Further investigation will help us to clarify it.

(6) Overall, the response time data showed much smaller differences than the magnitude differences of sizes among the test repositories. With relatively lower-powered test machines and the big repositories, the worst response time is about 10 seconds. We are cautiously optimistic about the scalability of the ASCS.

Discussion and Future work

(1) The need for a DAML query language

We believe there is a need for a universal query language to support distributed agents. We investigated the current implementations of RDF query languages, such as Guha's[4], Karvounarakis's RQL [5] and the others mentioned in [6]. They are mostly well-designed SQL type languages suited for queries on internal RDF repositories, but not for communications among distributed agents. One promising solution for a query language among agents is DAML itself. Agents should be able to employ the RDF parser to parse the variables and constants in namespaces, name symbols, or constraints and translate them into any internal formats they want to use. The results of the query should also be translated into DAML. The same principle applies to the Query Interface (QI): it has a

DAML interface with other agents and its own desired interface to the user. For the purpose of experiments, we invented a temporary convention for the SSAs and STSs to pass the queries. It is not an ideal solution but is sufficient for the experiments. For example, our software can recognize the namespace variables and the name symbol variables in the following query that is asking the synonyms of the word “capacity” via the senses of the word.

```
(DAML Query:
<rdf:Description
about="http://projects.teknowledge.com/DAML/Variable?URL=U&amp;NS=Us1#capacity">
<hasSense rdf:resource="http://projects.teknowledge.com/DAML/Variable?NS=Uo1&amp;S=O1"/>
</rdf:Description>
<rdf:Description
about="http://projects.teknowledge.com/DAML/Variable?URL=U2&amp;NS=Uo1&amp;S=O1">
<wordHasTheSense
rdf:resource="http://projects.teknowledge.com/DAML/Variable?NS=Uo2&amp;S=O2"/>
</rdf:Description>)
```

In Prolog its translation is as follows:

d(U, Us1, capacity, ‘http://projects.teknowledge.com/DAML/DynamicOntology1.DAML’, hasSense, Uo1, O1),
d(U2, Uo1, O1, ‘http://projects.teknowledge.com/DAML/DynamicOntology1.DAML’, wordHasTheSense, Uo2, O2).

where O1, O2 are variables for the name symbols, a synset and a word in the synset. Uo1 and Uo2 are variables for their namespaces. Us1 is the variable for the namespace of constant ‘capacity’. U and U2 are the URL variables for the RDF triples, (capacity, hasSense, O1) and (O1, wordHasTheSense, O2).

(2) The need for a cluster of servers to serve the repository:

We strongly believe that, in order to reduce the response time for complex queries on a large repository, there is a need to split the repository into several pieces and have a distributed server farm for storing them. One caveat is that each server may have only partial answer to the query. Thus, we also need something like a query coordinator that can perform query in the distributed mode.

In summary, in this short paper we reported the experiments we have done with ASCS semantic search. With the advent of DAML, we got the high-quality responses for our queries. We also observed some possible overhead costs in using the translation services. Further investigation is needed to make more certain conclusions. The future development of the ASCS will include the expansion of the query format to cover more than the conjuncts, the *dynamic* search/query-and-answering with DAML, and the search combined with text-based agents.

References:

[1]: James Hendler: Agents and the Semantic Web (to appear in IEEE Intelligent Systems). http://www.cs.umd.edu/users/hendler/AgentWeb.html

[2]: WordNet - a Lexical Database for English.
http://www.cogsci.princeton.edu/~wn/

[3]: XSB research group.
http://xsb.sourceforge.net/

[4] R.V.Guha. rdfDB : rdfDB Query Language.
http://web1.guha.com/rdfdb/query.html

[5] Greg Karvounarakis. The RDF Query Language (RQL).
http://139.91.183.30:9090/RDF/RQL/index.html

[6] Sergey Melnik. Storing RDF in a relational database
http://WWW-DB.Stanford.EDU/~melnik/rdf/db.html