Benchmarking ontology tools. A case study for the WebODE platform

Oscar Corcho, Raúl García-Castro, Asunción Gómez-Pérez

Ontology Group. Departamento de Inteligencia Artificial, Facultad de Informática,
Universidad Politécnica de Madrid, Spain
{ocorcho@fi.upm.es, rgarcia@fi.upm.es, asun@fi.upm.es}

Abstract

As the Semantic Web grows the number of tools that support it increases, and a new need arises: the assessment of these tools in order to analyse whether they can deal with actual and future performance requirements. In order to evaluate ontology tools' performance, the development and use of benchmark suites for these tools is needed. In this paper we describe the design and execution of a benchmark suite for assessing the performance of the WebODE ontology engineering workbench.

Introduction

In recent years, much effort has been made to develop ontology editors and ontology tools for creating and maintaining ontologies with different knowledge models and with different underlying knowledge representation paradigms (Gómez-Pérez et al., 2003). Ontology tools are now used in a wide range of applications and manage large upper level and general ontologies, natural language resources like thesauri, etc. Hence, ontology tools should be evaluated thoroughly in order to analyze whether they can deal with actual and future performance requirements.

For the time being, little effort has been put on creating benchmark suites and carrying out performance studies for ontology technology, while in other fields there is much work already done. For this reason, we think that there is a need to construct a benchmark suite for ontology tools. This benchmark suite will allow evaluating the ontology management services that these tools provide, possibly detecting aspects that need optimization in order to allow for a better performance and integration of this technology into other information systems.

To carry out this benchmarking study, we have focused on the WebODE ontology engineering workbench (Arpírez et al., 2003), analyzing the performance of the public methods of its API. These methods allow managing the ontology components defined in the WebODE knowledge model (concepts, relations, instances, axioms, constants, bibliographic references and imported terms). Since these methods are similar in most of the ontology tools, this study can in the future be easily extended to other ontology tools, such as Protégé-2000, OntoEdit, KAON, etc.

In this paper we describe, step by step, how we have designed and executed a structured benchmark suite in order to analyze the performance of these methods.

Goals of the benchmarking

The long term goal of benchmarking the WebODE ontology engineering workbench is to achieve a continuous improvement in the platform's quality. There are other short term goals as:

- Assess the platform's performance, so as to be able to detect anomalies in it.
- Monitor the platform, so as to be able to observe the performance of critical elements and the effects in performance when making changes in the platform.
- Diagnose future problems of the platform.

Design of the benchmark suite

We have selected the 72 ontology management methods from WebODE's API in order to be able to make a complete analysis. As every service and application supported by WebODE manages ontologies through its API methods, knowing these methods' performance will let us know the performance of the services and applications.

As we are interested in the platform's temporal performance, the **metric** will be the **execution time** of the methods.

To carry out the study, the data needed will be obtained from four different scenarios, where the methods will be executed:

- **Over a high load state**. In order to be able to detect performance anomalies.
- **Repeatedly over the same load state**. To be able to check the method's stability.
- **Over incremental load states**. To know the load-performance relationship.
- **With different input parameters**. To check if changing the method's input parameters affects its performance.

Definition of the benchmarks

In order to have a representative and interpretable set of tests (Williams et al., 2003), the API methods have been classified into five groups according to the kind of operation they do: Inserts, Updates, Removes, Selects and Non basics. This last group consists of methods that use other methods from the API.

For each selected method, one or more benchmarks have been defined according to the variation of its input parameters. So, from the 72 API methods we get 128 different benchmarks.

Each benchmark executes its correspondig method with the selected input parameters and stores the execution time of the method.

If the evaluation of the platform is to be effective, the benchmarks must be characterized accurately (Dongarra et al., 1987). So, the definition of the benchmarks has been completed with two execution parameters and an initial load state.

Definition of the execution parameters

As the benchmarks must be robust and scalable, allowing variable and unpredictable input rates and behavior (Bull et al., 1999; Shirazi et al., 1999), they have been parameterized according to two parameters:

- **Load factor (X)**. Sets the load factor for each benchmark's initial state.
- **Number of iterations (N)**. Sets the number of consecutive executions of the method in a single benchmark.

Definition of the load state

Every benchmark must be compared according to the same situation. So, a common initial load state has been defined for each benchmark group defined previously.

The initial load state of each individual benchmark has been defined as the ontology components that must exist in the platform in order to execute the benchmark with no errors.

The initial state of each benchmark group is the union of the initial states of each benchmark in the group. Table 1 shows the initial load state of the Updates group.

Updates group		
1 ontology with	X term references	
	1 concept with	X class attributes
		X instance attributes
		X synonyms
	X concepts with	1 class attribute
		1 instance attribute
		1 synonym
	X constants	
	X formulas	
	X groups	
X ontologies with	1 term reference	
	1 concept	
	1 constant	
	1 formula	
	1 group	

Table 1: Initial load state of the Updates group

Execution of the benchmark suite

The benchmarks have been implemented with Java, using only standard libraries and with no graphical components, in order to have a portable benchmark suite.

Once defined and implemented, each benchmark has been run several times with different number of iterations (N=10, 50, 75, 100, 200, 300, 500, 1000, 2000, 3000, 4000, and 5000) and with increasing load factors (X=10, 50, 75, 100, 200, 300, 500, 1000, 2000, 3000, 4000, and 5000). As with a load factor of 5000 we have obtained enough data to be able to differentiate the methods' performance and their behavior, the benchmarks haven't been executed with higher load factors.

The execution results have been stored in a hierarchical measurement data library, in order to be able to access them easily.

Analysis of the results

First of all, we have to bear in mind that the conclusions obtained after analyzing the results are usually temporary limited (Gray, 1993). As the methods in the API will undergo changes, these results just inform us about WebODE's current performance, not its future one.

The data obtained after running the benchmarks is the measurement of the execution times of the methods. As this data can't be used directly, it must be transformed to obtain analyzable data.

From the raw results we can obtain:

- **Graphs** that show the behavior through time of the methods.
- **Statistical values** worked out from the execution times of the methods.

Statistical values obtained are central tendency measures (mean, median and mode), variation measures (variance and standard deviation), and Pearson's correlation which will show us the linear strength of the determinations.

Also, we have calculated the percentage of measurements out of interval and estimated the function determined by the execution times through simple regression.

In the graphs, we can see periodically peaks representing high execution times, due to tasks from the systems that run under the platform, like Java or Oracle. That's why we have worked with a "smoothed" version of the graphs, so they are easier to analyze. In order to smooth the graphs, only the medians of the values from each pixel interval have been drawn.

The original graph from benchmark1_1_14 (which adds values to class attributes using the method *addValueToClassAttribute*) can be seen in Figure 1 and its smoothed graph appears in Figure 2.

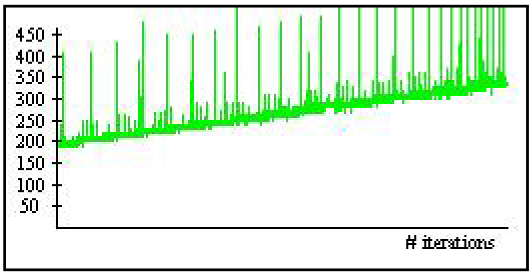

Figure 1: Original graph from benchmark1_1_14

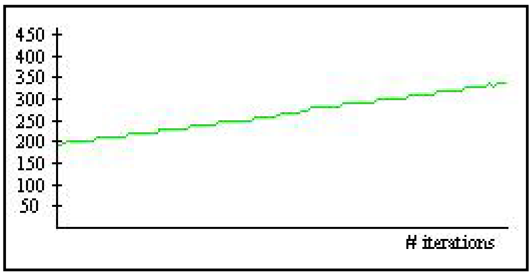

Figure 2: Smoothed graph from benchmark1_1_14

Let us see now the analysis for each scenario proposed above:

Running over a high load state

In order to compare the performance of the different methods, we have analyzed the median of its execution time when running the benchmarks over the maximum load state (X=5000), with the maximum number of iterations (N=5000).

Table 2 shows the median of the execution times of the Updates' group of benchmarks.

Benchmark	Method	Median
benchmark1_2_09	updateInstanceAttribute	201.0 ms.
benchmark1_2_10	updateInstanceAttribute	191.0 ms.
benchmark1_2_11	updateSynonym	120.0 ms.
benchmark1_2_12	updateSynonym	110.0 ms.
benchmark1_2_14	updateConstant	110.0 ms.
benchmark1_2_17	updateGroup	100.0 ms.
benchmark1_2_18	updateGroup	100.0 ms.
benchmark1_2_07	updateClassAttribute	80.0 ms.
benchmark1_2_08	updateClassAttribute	80.0 ms.
benchmark1_2_13	updateConstant	60.0 ms.

Table 2: Execution times of the Update´s group of benchmarks

Only 14 methods (19%) of the whole benchmark suite have an execution time greater than 250 ms.

In every case, the percentage of values out of interval is very low (less than 2%), which is an acceptable value.

Pearson's correlation coefficient shows what can be seen to the naked eye, that there is little linear strength between the values, because of the numerous peaks in the execution times.

Running repeatedly over the same load state

In order to compare whether there is a variation in the performance when running a method repeatedly, the key factor is the behavior of the functions defined by the execution times when running the benchmarks over the maximum load state (X=5000), with the maximum number of iterations (N=5000).

In most cases, execution times remain constant through time (like in Figure 3), in some cases the execution time diminishes (like in Figure 4), although this is not worrying because performance increases. But, there are 4 methods (5%) whose execution time increases through time with a slope greater than 0.02 (like in Figure 5).

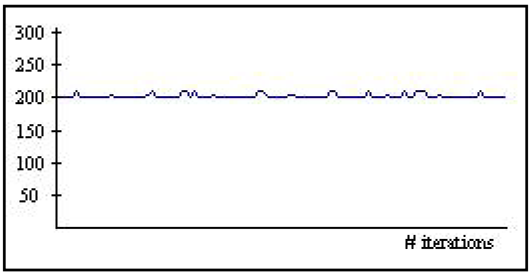

Figure 3: Graph from benchmark1_2_09 (the execution time remains constant through iterations)

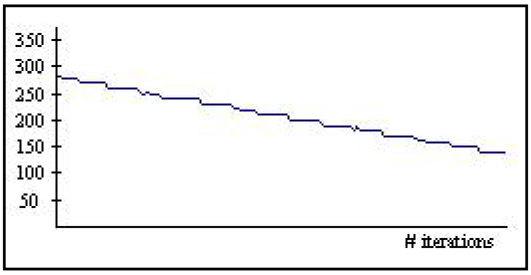

Figure 4: Graph from benchmark1_3_10 (the execution time decreases through iterations)

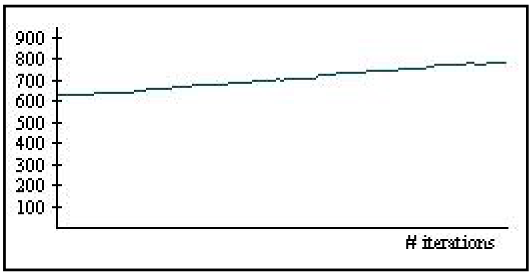

Figure 5: Graph from benchmark1_1_18 (the execution time increases through iterations)

Running over incremental load states

In order to compare the performance of the different methods when increasing the load, we have analyzed the function defined by the medians of the execution times of each benchmark from a minimal initial state (X=10) to a maximum one (X=5000), with the maximum number of iterations (N=5000).

Although some of the estimated functions are constant, like the one shown in Figure 6, most of them have a positive slope as can be seen in Figure 7. Only in 11 methods (15%), this slope is greater than 0.02.

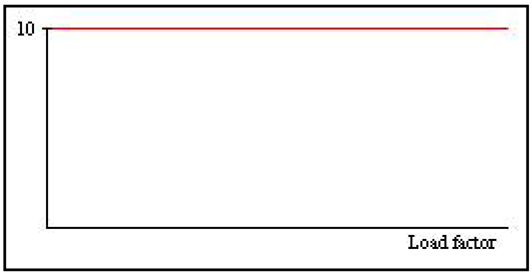

Figure 6: Graph from the medians of benchmark1_2_04 (the estimated function is constant)

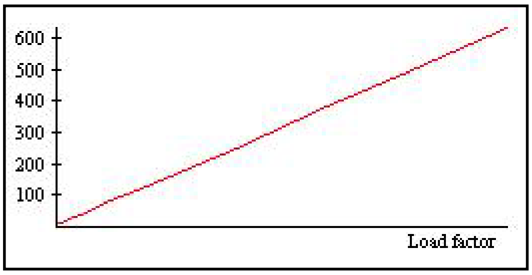

Figure 7: Graph from the medians of benchmark1_4_10 (the estimated function's slope is positive)

Besides, Pearson's correlation coefficient values show a high linear relation, meaning that the evolution through time of the execution times behaves linearly.

Running with different input parameters

In order to compare the performance of the different methods when changing its input parameters, we have analyzed the behavior of the benchmarks that use the same method when running over the maximum load state (X=5000), with the maximum number of iterations (N=5000).

In general, the method's performance is not sensible to parameter changes. The only exceptions are 12 methods

(16%) whose execution times differ significantly when changing input parameters.

There are two kinds of variations: the execution times behave differently over time, like in Figure 8 where the benchmarks that execute the method *addValueToClassAttribute* are shown, or their values are very different, as the benchmarks that use the method *getClassAttribute* which appear in Figure 9.

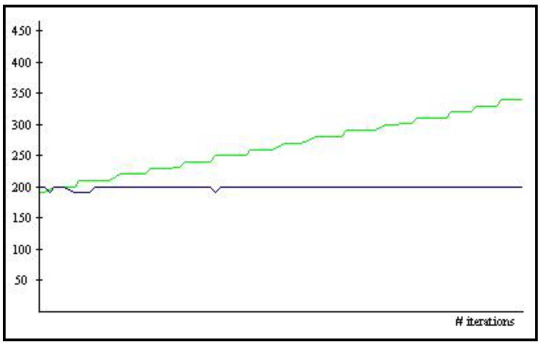

Figure 8: Graphs from benchmark1_1_14 and benchmark1_1_15 (execution times behave differently)

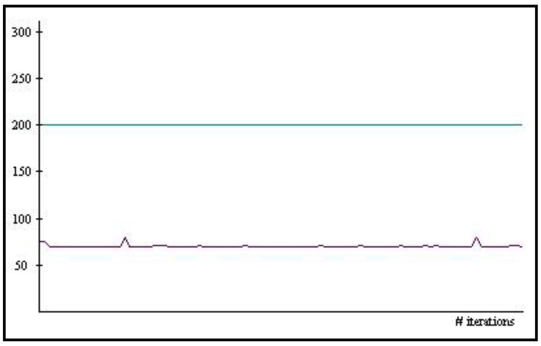

Figure 9: Graphs from benchmark1_4_11 and benchmark1_4_12 (execution times are very different)

Development of improvement recommendations

Once the data has been analyzed, the next step is the development of improvement recommendations. These recommendations include those methods whose execution times:

- Are higher than 250 ms.
- Increase over time with slope greater than 0.02.
- Increase when augmenting load with slope greater than 0.02.
- Vary when modifying its input parameters.

So, improvement recommendations include 21 of the 72 WebODE's API methods (29%).

Adapt the system

From the improvement recommendations obtained after the analysis of the results, the WebODE's development team has identified the changes that must be implemented in the platform in order to improve its performance.

After rerunning the benchmark suite, we have checked the decrease of the execution times in the improved methods.

For example, the study showed that the methods that manage instance attributes (*addValueToInstanceAttribute, removeValueFromInstanceAttribute, getInstanceAttribute and getInstanceAttributes*) were among the slowest. After optimizing a SQL query from an internal piece of code used by these methods, their overall performance improved as can be seen in Table 3.

Benchmark	Before	After	Improvement
benchmark1_1_18	600 ms.	461 ms.	23%
benchmark1_1_19	471 ms.	371 ms.	21%
benchmark1_3_12	390 ms.	331 ms.	15%
benchmark1_3_13	281 ms.	240 ms.	14%
benchmark1_4_15	280 ms.	230 ms.	17%
benchmark1_4_16	300 ms.	250 ms.	16%

Table 3: Execution times before and after improving instance attribute management

Conclusions and future work

After benchmarking the WebODE ontology engineering workbench:

- We have identified the slowest methods, the bottlenecks and the performance anomalies of the platform.
- We have determined precisely the platform's performance.

In the future, we plan to:

- Extend benchmarking to other ontology tools (OntoEdit, Protégé-2000, KAON, etc.).
- Include other metrics to measure properties like correctness, stability, etc.
- Carry out a synthetic study about the performance of services and applications that use WebODE.

Acknowledgements

This work is partially supported by the IST project KnowledgeWeb (IST-2004-507482) and by the IST project Esperonto (IST-2001-34373).

References

Arpírez J.C., Corcho O., Fernández-López M., Gómez-Pérez A. (2003). WebODE in a nutshell. AI Magazine. 24(3) (pp. 37--47). Fall 2003.

Bull J.M., Smith L.A., Westhead M.D., Henty D.S., Davey R.A. (1999) A Methodology for Benchmarking Java Grande Applications. EPCC. June 1999.

Dongarra J., Martin J.L., Worlton J. (1987) Computer benchmarking: paths and pitfalls. IEEE Spectrum, Vol. 24, N. 7 (pp. 38--43). July 1987.

Gómez-Pérez A., Fernández-López M., Corcho O. (2003). Ontological Engineering. Springer-Verlag. November 2003.

Gray J. (1993). The Benchmark Handbook for Database and Transaction Systems (2nd Edition). Morgan Kaufmann.

Shirazi B., Welch L., Ravindran B., Cavanaugh C., Yanamula B., Brucks R., Huh E. (1999). DynBench: A Dynamic Benchmark Suite for Distributed Real-Time Systems. IPDPS 1999 Workshop on Embedded HPC Systems and Applications, San Juan, Puerto Rico, April 1999.

Williams L.G., Smith C.U. (2002). Five Steps to Solving Software Performance Problems. http://www.perfeng.com.