RDF Core: A component for effective management of RDF Models

FLORIANA ESPOSITO, LUIGI IANNONE, IGNAZIO PALMISANO AND GIOVANNI SEMERARO

Dipartimento di Informatica
Università degli Studi di Bari
Via Orabona, 4
Bari, 70125, ITALY
+39 080 544 2299
{esposito,iannone,semeraro}@di.uniba.it, ignazio_io@yahoo.it

Abstract.

In order to make Semantic Web effective, the first step was the development of languages that could support data portability, namely XML, metadata descriptions, namely RDF, and ontology management and inference, such as DAML+OIL, OWL etc. Those languages have to be manipulated by applications and many Application Programming Interfaces (APIs) have been developed in order to accomplish this task. Obviously, they differ in implementation details. Moreover, developers often would like to exploit more than an API at a time. Another issue is that a developer would be very advantaged if he could have a uniform support for some services across these frameworks (such as query languages), despite the lack of standards. In this paper, we present a component, called *RDFCore*, developed in order to overcome these problems. We will also illustrate the added value that our framework provides to RDF in order to exploit the full potentiality of the language and to employ it in research as well as in real world applications. Consequently we will provide some test results on the performances of the presented framework.

Introduction

World Wide Web Consortium (W3C), that is the main promoting committee involved in the evolution towards the Semantic Web[1], has been recently working on the development of technologies that could support this process. While some of these technologies are still in early phases, part of them can already be exploited in real world applications. This is the case of Resource Description Framework (RDF). It represents the basic support to write metadata on Web resources and to grant interoperability among heterogeneous applications when exchanging these metadata.

RDF describes resources in terms of primitives (classes, properties, resources, etc.) without taking into account the description structure itself. In fact, the description can be encoded in XML (but also in other different formats, see for instance [2]). This ensures its portability across the Web.

Moreover, RDF represents a suitable solution to implement the Semantic Web vision also because it presents three key features:

- **Extensibility.** Each user can add its own description extending pre-existing ones without any limit.
- **Interoperability.** RDF descriptions can rely on XML serialization every time they need to be exchanged among heterogeneous platforms
- **Scalability.** RDF descriptions can be viewed as sets of three field records (triples) (Subject, Predicate and Object). This makes them easy to fetch and manage even when a single description holds many triples in it.

Many Application Programming Interfaces (APIs) have been developed in order to support RDF-based applications. They offer a lot of useful features, ranging from efficient persistence and powerful query languages [8] to simple and well designed object models [4]. That is why we felt the need for a uniform framework (RDFCore) that will be presented in the following sections. The main aim of RDFCore is granting the widest compatibility with existing RDF APIs, exploiting their advantages in a transparent way for users and, where possible, enhancing traditional approaches to RDF-based development.

RDF Core

Overview

In the following section we will describe a framework named RDFCore and, besides its features, we will also point out how the problems related to RDF have been tackled.

RDFCore main components: Managers

The architecture sketched below (Figure 1) shows the main components of the RDFCore Framework.

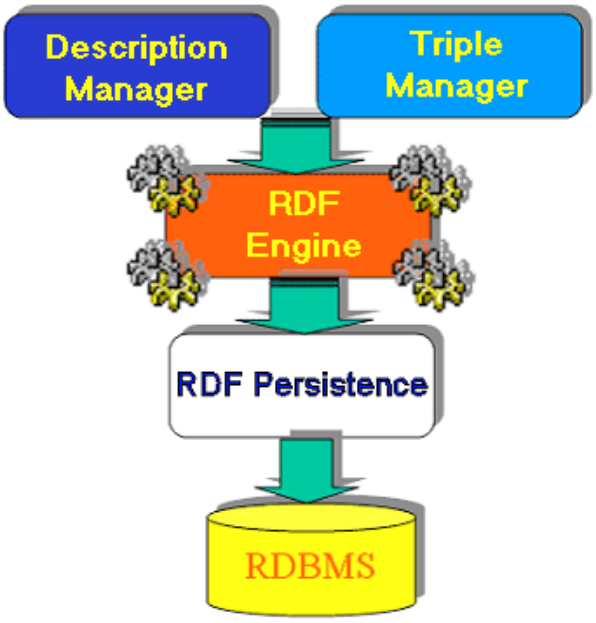

Figure 1 RDFCore Architecture

RDF Descriptions can be seen as sets of statements (typically called *Models*). Each statement is a triple compound by a subject, a predicate and an object. Therefore, users access RDF resources at two different levels of granularity – *Models* and *Statements*. That is why we developed two different entities, called *Description Manager* and *Triple Manager*, that deal with all the possible operations on

Descriptions and on *Triples*, respectively. Therefore, as far as *Descriptions* are concerned, users can:

- Add/Delete, Retrieve a *Description* to/from their own repository
- Update an entire *Description* with a new one
- Query a *Description* or a bunch of them.

while *Triple Manager* offers all the typical operations on single statements or on sets of statements (as subsets of a *Description*) like:

- Add
- Delete
- Update

All these operations would seem quite obvious. Indeed, all the most famous APIs currently available offer similar support to RDF users (see for instance Jena RDF Toolkit [3] or Stanford RDF APIs [4]). However, all these operations within our framework bring with themselves a slight advantage.

First of all, RDFCore has been devised as a multi-user environment. In fact, each user owns its own repository of RDF resources. Furthermore, users can be arranged in groups, can share resources with other members and there is the possibility of establishing policy rights on operations involving shared resources, such as granting/removing read/write access for a particular user or group of users. Other APIs do not offer a well-constructed persistence model like this one. The usefulness of such user management is strictly related to resource authoring. As a matter of fact, if the scenario is the WWW we could easily foresee communities of Web resource authors that generate, along with the actual web-resource, its description in RDF (no matter whether this generation will be automatic or not). Therefore, the need of having such an organisation of the RDF resources would soon arise.

RDF Engine and RDF Persistence

Description Manager and *Triple Manager* make up the sole user interface of RDFCore and they both rely on the RDF Engine module (see Figure 1).

In the RDFCore architecture, RDF Engine represents a specification rather than a concrete piece of software. In fact, it enumerates all the necessary operations for the

upper modules to properly carry out their functionalities. Actually, each call to the business functions of the proper *Manager* is translated into a combination of RDF Engine operations.

In the previous sections, we mentioned that there are many existing APIs to manage RDF and we also pointed out that it is strongly desirable that users can have the possibility to exploit features of any of them without switching architecture. That is why RDF Engine specifies which operations are required and nothing else. The responsibility of actual implementation of the services specified by RDF Engine is delegated to RDF Persistence level components.

In this way, a well-known best practice in Object-Oriented design, that is the *implementation* of abstract interfaces, can be exploited. In practice, RDF Engine is an interface whose implementation can vary depending on the requirements developers want to meet.

Therefore, many RDF Engine implementations can co-exist in a single instance of RDFCore. A typical scenario would be one in which different kinds of users have different implementations of the underlying RDF Engine. The advantage is that some users could need some requirements that are provided (for instance) within some specific persistence. The only effort in order to meet those requirements is to build up an implementation of the RDF Engine that acts as a bridge between that persistence and the upper level components (*Managers*). A more concrete example will be provided below in the description of the applications of our framework.

Actually, two implementations of RDF Engine have been produced, based on two different solutions for RDF Description storage/retrieval:

- An implementation based on RDF/XML serialization
- An implementation based on triple storage, built on Jena Toolkit API [3]

Both of them, as well as the upper components, comply to the well-known Stanford RDF API [4] as a standard for RDF object model, since it is the most widespread basic API for RDF Description management. This is accomplished by means of establishing that the input/output parameters in the modules interface have types taken from the RDF API object model (such as *Model*, *Statement*, *Resource* etc.)

Exploitation of RDFCore: COLLATE

One of the most complete exploitation case studies for our framework takes place in the EU research project COLLATE (IST-1999-20882) [5]. It belongs to the Fifth Framework Programme in scientific European Community research programme, under the Information Society Technology category, Key Action III: "Multimedia Content and Tools". The focus of this project is the development of a collaborative system for scientists involved in the study of the film production in Austria, Germany and Czech republics in the 30s. Three film archives have to be made electronically available (in order, above all, to preserve very fragile and intangible material) and scientists have to be allowed to index, catalogue and annotate such assets in order to build scientific discourses on their work among the scientific community endorsed with COLLATE [6], [7]).

This could be easily assimilated to the wider scenario foreseen by Semantic Web: a huge quantity of resources (documents, assets) with many relationships among them.

COLLATE requirements are:

- A uniform way of identifying resources (films, film related documents, cataloguing and indexing information, scientist annotations, scientific discourses)
- Distribution of information; in fact, archives still keep their resources in a decentralized architecture in order to avoid the moving of huge amount of data, both physically and electronically (for obvious reasons)
- Intelligent navigation through data and metadata, including navigation across scientific discourses on resources

For all these reasons RDF is a straightforward solution since it holds in itself the features we underlined in the introductory sections.

We go on examining which added value our framework provides to COLLATE. It is quite obvious that a huge collection of documents and metadata such as COLLATE heritage needs a careful devising of a scalable component in order to manage storage and retrieval of both resources and relationships among them. While the solution for the former problem is delegated to efficient RDBMS, as far as the latter we developed a suitable RDF Persistence for granting scalability to RDFCore framework. This module relies on Jena Toolkit storage model for RDF. It consists in exploiting a

relational representation of the RDF triples (subject, predicate, object) stored in a database. This approach takes advantage of the outstanding performance rates of the most famous RDBMS (such as Oracle, MySQL and PostgreSQL). One of the most immediate benefits is the fact that applications need not to load in-memory RDF *Models* (Descriptions) in order to deal with small portions of them (typically small sets of *Statements*), saving lots of memory and time for each operation.

Moreover, Jena Toolkit offers RDF Description Query Language (RDQL [8]) as language for querying RDF Descriptions. This support has been extended for querying multiple *Models*, that together with multi-user environment and scalability, proved to be a suitable solution for COLLATE requirement.

The query language, however, remains a weakness point of all RDF APIs available, including Jena. At the time of writing, still no standard query language specifications are available. This hampers the interoperability between components and, therefore, between different systems; in other words, two systems using different APIs to manage RDF can exchange data, but cannot easily exchange queries on these data.

To address this issue, RDFCore embeds a subcomponent, called Enhanced Query Engine, able to deal with different query languages. The design of this component exploits the Strategy pattern [10] (like other components in RDFCore architecture), enabling the use of a dynamic set of query languages. In order to add the support for a new query language, only the classes implementing the interfaces to wrap the parser of the language and the query engine are needed, allowing for easy update. This update, obviously, can be the standard query language the W3C (together with other organizations) is working on, as soon as it is available [1].

Empirical evaluation of performances

In this section we present some results from a preliminary empirical evaluation we carried out on the RDFCore software components. We mainly tried to investigate one of the key features that a framework devoted to Web (and Semantic Web) development should have: scalability. The notion of scalability is very well known in IT environment and it can be measured with respect to many variables. Being

[1] http://www.daml.org/dql/

basically a knowledge storage system, RDFCore needs to be scalable, firstly with respect to the amount of data that it has to manage. Therefore, tests that have been carried out had the purpose of investigating how smoothly RDFCore performances decreased as the data size increased. Particularly, our aim was to have a component showing linear scalability with data size, i.e. time doubles as data size doubles.

In the previous sections, while describing the design of RDF persistence architecture, we pointed out that our framework could provide simultaneously different strategies for the actual data storage thanks to the persistence architectural layer of abstraction. Indeed, as we mentioned in the previous section, we developed two different persistence mechanisms, respectively:

- Based on file system binary storage of RDF/XML resources, relying on a compressed XML storage format (namely PDOM [2])
- Based on RDBMS storage of RDF resources, relying on Jena API for RDF.

We prepared two different test sets, both devised in order to progressively scale up in data size but with slightly different strategies. The first one increases data in size but not in content, by simply repeating the basic RDF description *n* times in the same document. The second one has been created by adding new statements to the starting description without repeating any object, subject or properties. In this way all triples in the descriptions from the second test set are different from each other, while there is a lot of redundancy in the first test set. The reason for doing that is that in both RDF persistence implementations some mechanism to take advantage from redundancy has been devised (e.g.: indexing of URI). Therefore an RDF description with many repetitions should be processed in lesser time than a variegated description.

In all our tests, the descriptions named *NNNx_rdf* are redundant descriptions, where *NNN* is the number of times a particular triple is replicated in the description; on the other hand, the descriptions named *OutputNNNNN_rdf* are descriptions with no redundancy, and *NNNNN* is the number of triples in the particular model.

[2] http://www.infonyte.com/en/prod_pdom.html

Obtained results

In Table 1 and Table 2, divided for the sake of readability, we show the results of processing the first test set (highly redundant) with an RDFCore exploiting the file system-based persistence that we mentioned before, and with the JENA-based persistence, relying on the MySQL RDBMS. In Figure 2 and Figure 3 (for PDOM), and subsequently Figure 4 and Figure 5 (for JENA), we show the growth of required time to store descriptions compared with a theoretical linear function on data size (used as baseline). In these figures, as well as in the subsequently ones, the scale on the Y axis is logarithmic. Where not specified, the measuring unit for time is the millisecond. Table 3 reports the results obtained on the redundancy-free test set, while Figure 6 and Figure 7 provide a graphical representation of them. Notice that missing values (- in tables) were omitted because they have been considered irrelevant.

		PDOM Persistence			JENA Persistence		
File	File size (Kbytes)	Elapsed time (milliseconds)	Theoretical elapsed time	PDOM file size	Reading time	Storing time	Theoretical storing time
2x.rdf	173	3886	4000	-	203	9777	10000
3x.rdf	259	4016	6000	-	250	14772	15000
4x.rdf	342	4226	8000	-	313	19779	20000
5x.rdf	432	4446	10000	-	453	24767	25000
6x.rdf	518	4827	12000	-	485	29787	30000
7x.rdf	605	4547	14000	-	563	34787	35000
8x.rdf	691	5178	16000	-	640	39766	40000
9x.rdf	777	4757	18000	-	734	44822	45000
10x.rdf	864	5417	20000	-	875	49822	50000
11x.rdf	950	5117	22000	-	953	54762	55000
12x.rdf	1036	5168	24000	-	1125	59783	60000
13x.rdf	1122	5007	26000	-	1062	64747	65000
14x.rdf	1209	5488	28000	-	1250	69827	70000
15x.rdf	1295	5427	30000	-	1234	74727	75000
16x.rdf	1381	5948	32000	-	1312	79837	80000
17x.rdf	1468	5728	34000	-	1422	84737	85000
18x.rdf	1554	5808	36000	-	1547	89737	90000
19x.rdf	1640	5879	38000	-	1547	94687	95000
20x.rdf	1727	5929	40000	-	1656	99682	100000

Table 1 High redundancy test (a)

		PDOM Persistence			JENA Persistence		
File	File size (Kbytes)	Elapsed time (milliseconds)	Theoretical elapsed time	PDOM file size	Reading time	Storing time	Theoretical storing time
20x.rdf	1727	5929	40000	-	1656	99682	100000
30x.rdf	2590	7411	60000	261	2390	149573	150000
40x.rdf	3453	9043	80000	314	3250	199394	200000
50x.rdf	4316	10104	100000	370	4015	249230	250000
60x.rdf	5179	10905	120000	427	4781	299171	300000
70x.rdf	6042	11636	140000	482	5578	348987	350000
80x.rdf	6905	13930	160000	532	6453	398823	400000
90x.rdf	7768	13450	180000	584	7203	448658	450000
100x.rdf	8631	14130	200000	638	9437	498494	500000
110x.rdf	9494	15292	220000	691	9406	548403	550000
120x.rdf	10357	15793	240000	744	10343	598239	600000
130x.rdf	11220	18807	260000	797	11032	648011	650000
140x.rdf	12083	13450	280000	848	11688	697917	700000
150x.rdf	12946	22272	300000	900	12594	747847	750000
160x.rdf	13809	21802	320000	952	13469	797643	800000
170x.rdf	14672	23384	340000	1003	14469	847636	850000
180x.rdf	15535	24105	360000	1055	15469	897452	900000
190x.rdf	16398	25317	380000	1106	17438	947425	950000
200x.rdf	17261	26201	400000	1157	16891	997201	1000000

Table 2 High redundancy test (b)

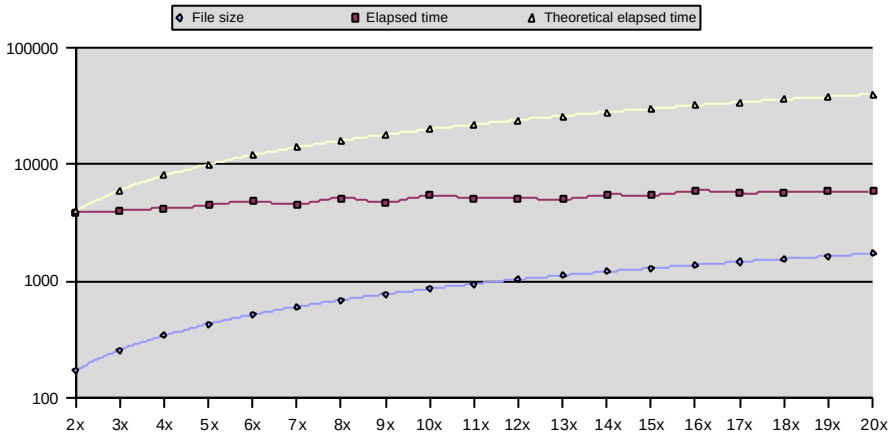

Figure 2 High redundancy test (PDOM) (a)

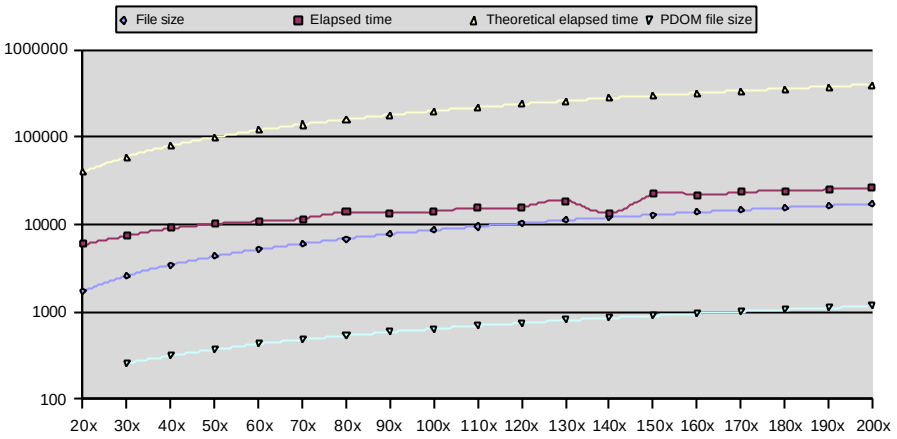

Figure 3 High redundancy test (PDOM) (b)

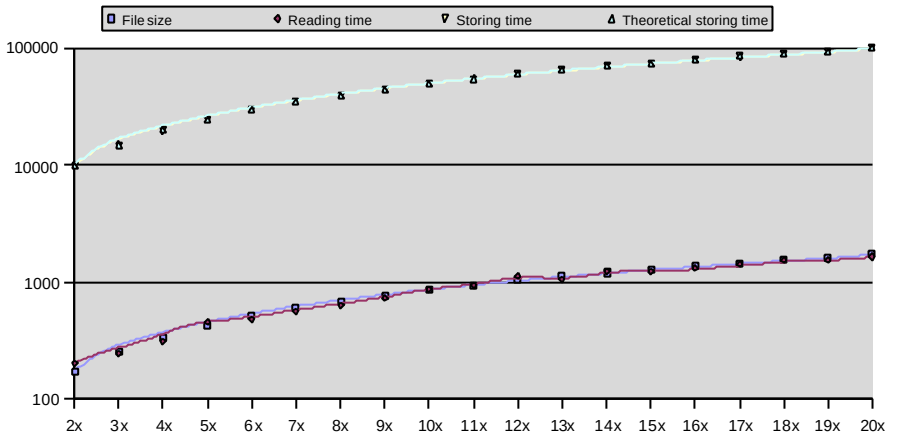

Figure 4 High redundancy test (JENA) (a)

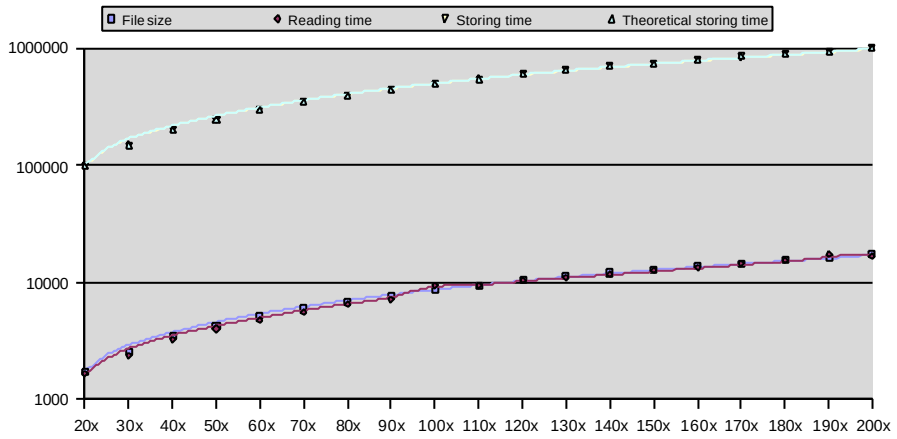

Figure 5 High redundancy test (JENA) (b)

		PDOM Persistence				JENA Persistence		
File	**File size**	**PDOM file size**	**Reading time**	**Storing time**	**Theoretical storing time**	**Reading time**	**Storing time**	**Theoretical storing time**
Output10000	1480	1210	6990	15382	15000	2219	83612	80000
Output20000	2990	2470	10404	26689	30000	3140	167201	160000
Output30000	4490	3700	15682	36823	45000	4797	250750	240000
Output40000	6000	4970	19999	48139	60000	6125	334200	320000
Output50000	7510	6210	26178	59776	75000	7828	418035	400000
Output60000	9000	7450	29893	76700	90000	9422	501715	480000
Output70000	10500	8700	34089	99152	105000	13281	585204	560000
Output80000	12000	9920	38305	145219	120000	15328	667835	640000
Output90000	13500	11100	45174	208650	135000	16531	752483	720000
Output100000	15000	12400	49812	308293	150000	18438	836212	800000

Table 3 No redundancy test

The X axis in Figure 6 and Figure 7 reports the number of triples in the files used for the test.

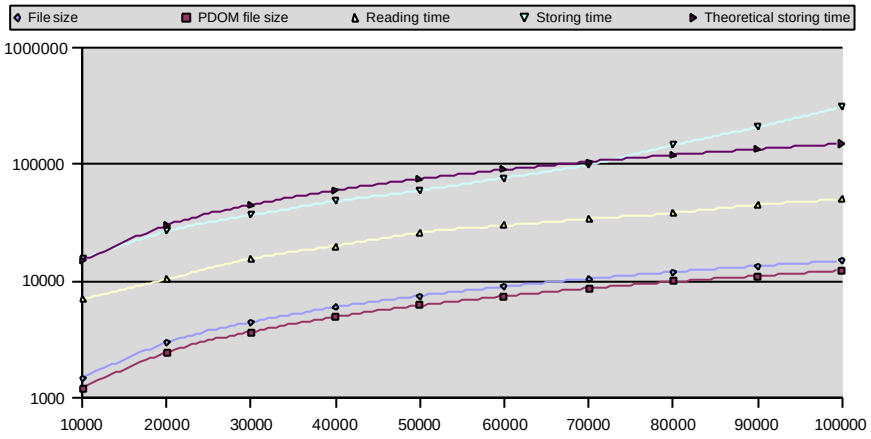

Figure 6 No redundancy test (PDOM)

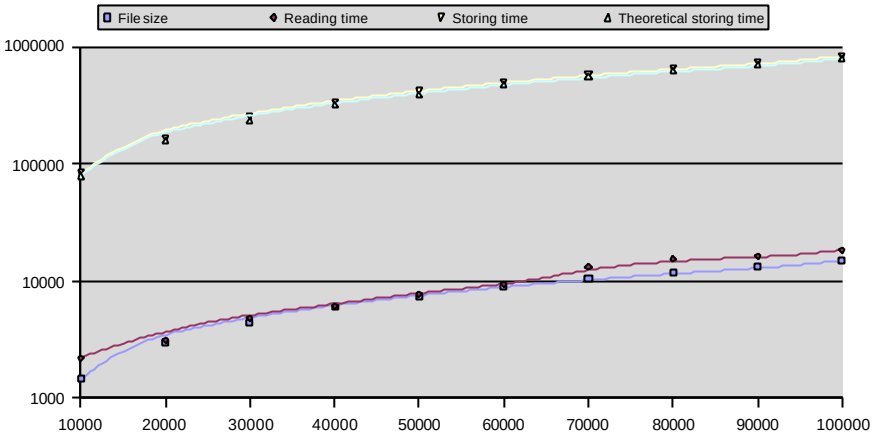

Figure 7 No redundancy test (JENA)

Table 4 reports RDFCore performances in adding a statement to very huge descriptions that have been already stored in the repository. Figure 8 and Figure 9 show the graphic trend of required time.

PDOM Persistence			**JENA Persistence**		
File	**Elapsed time**	**Theoretical elapsed time**	**File**	**File size**	**Elapsed time**
160x.rdf	9333	9333	**Output10000**	1480	358
170x.rdf	9564	9916	**Output20000**	2990	12
180x.rdf	10826	10500	**Output30000**	4490	25
190x.rdf	10756	11082	**Output40000**	6000	70
-	-	-	**Output50000**	7510	36
-	-	-	**Output60000**	9000	10
-	-	-	**Output70000**	10500	17
-	-	-	**Output80000**	12000	21
-	-	-	**Output90000**	13500	20
-	-	-	**Output100000**	15000	20

Table 4 Add triple test

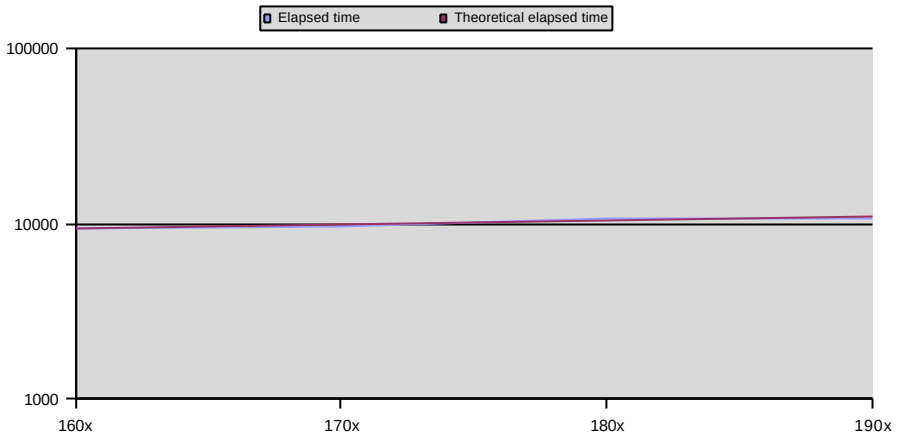

Figure 8 Add triple test (PDOM)

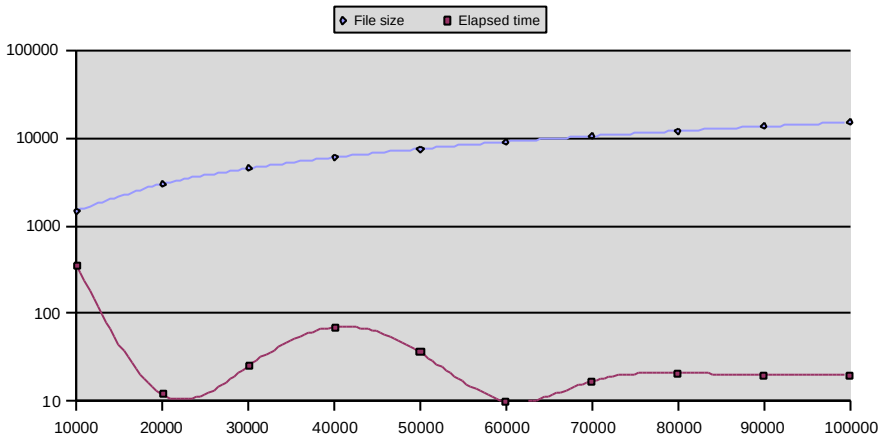

Figure 9 Add triple test (JENA)

Furthermore, we measured the time spent by RDFCore to retrieve a description from the repository and make it ready for manipulation by user (Table 5 and Figure 10 and Figure 11) and in querying a model for every triple it contains (Table 6 and Figure 12 and Figure 13).

	PDOM Persistence		JENA Persistence
Resource	**Elapsed time**	**Theoretical elapsed time**	**Elapsed time**
Output10000	13570	13000	484
Output20000	23804	26000	5
Output30000	34420	39000	15
Output40000	43573	52000	63
Output50000	59285	65000	31
Output60000	-	-	5
Output70000	-	-	7
Output80000	-	-	15
Output90000	-	-	16
Output100000	-	-	16

Table 5 Retrieve description test

As for Figure 6 and Figure 7, in Figure 8 and Figure 9 the X axis reports the number of triples in the files used for the test.

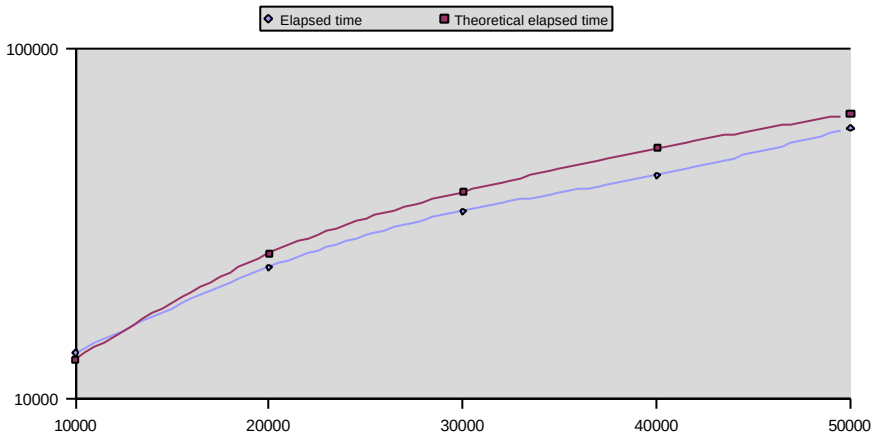

Figure 10 Retrieve description test (PDOM)

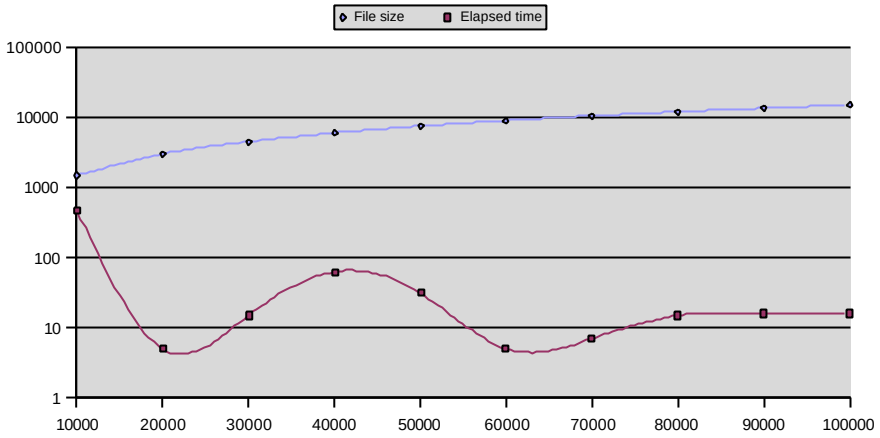

Figure 11 Retrieve Description(JENA)

		PDOM Persistence	JENA Persistence
Resource	**Triple number**	**Elapsed time**	**Elapsed time**
Output10000_rdf	10000	10505	453
Output20000_rdf	20000	15502	31
Output30000_rdf	30000	24075	16
Output40000_rdf	40000	32497	15
Output50000_rdf	50000	49361	16

Table 6 Querying persistence

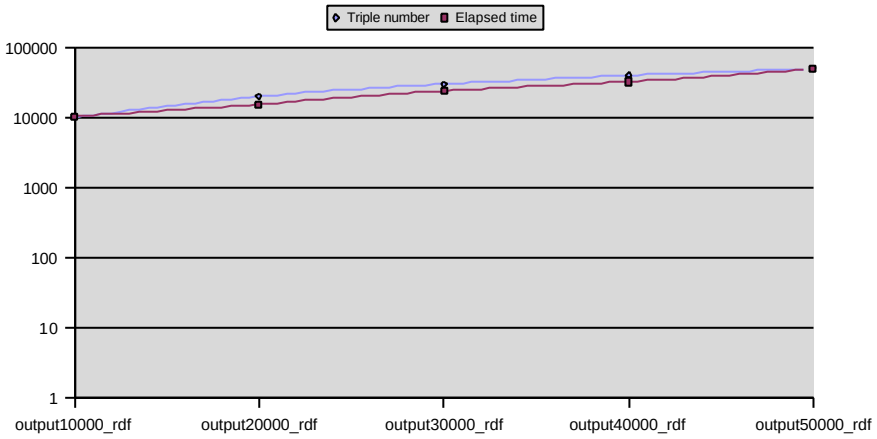

Figure 12 Querying persistence (PDOM)

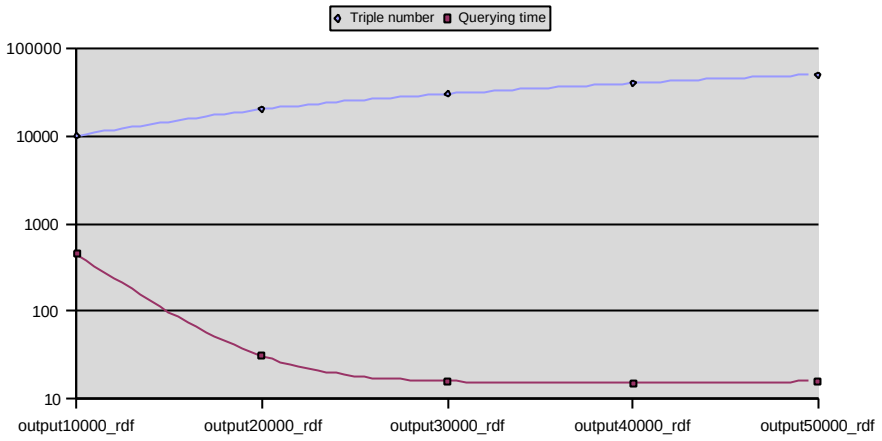

Figure 13 Query test (JENA)

Queries

The query test involves the use of the Enhanced Query Engine component of our architecture; specifically, the query used to stress the system (taking into account the size of the dataset and the size of results) was a very simple one: we asked the system to return every statement, describing a matching statement as a statement with a variable value for subject, predicate and object. This is done, in our system, creating a *Pattern* (a list of conditions on statements) and translating it into a query expressed in one of the query languages that are supported by the Enhanced Query Engine. In our test, we used RDQL as a query language; the translated query is

```
SELECT ?s, ?p, ?o WHERE (?s,?p,?o)
```

that returns every statement in the given model.

Result analysis

The obtained results show that the whole system does scale in a linear way with both persistence layers. It is noteworthy that JENA persistence absolute times, when adding a new model, are higher than those of the PDOM implementation. This depends on a JENA weakness due to the complexity of the internal database structure. The next version of JENA (JENA 2.0) promises substantial performance improvements, and this should tackle the resulting weakness of our system. On the

other hand, when doing retrieving and querying tests, where PDOM is still linear, JENA is very close to constant complexity, independently from the size of managed data. This result was expected because of the different approaches used by the two distinct layers: PDOM loads its data into in-memory representations, while Jena relies on its RDBMS persistence, obviously faster in these operations.

Conclusions

In this paper, we briefly described motivations and requirements for the brand new vision emerging on the Web: the Semantic Web. We pointed out, among others, the need of exploiting suitable technology for dealing with metadata, such as RDF. This technology has many benefits and, as we stated in the first sections of this paper, has to be integrated in frameworks that offer both scalability and standard support. Then, we presented our solution to tackle RDF related issues and we mentioned one specific application of RDFCore in a current ongoing EU research project (COLLATE). Finally, we presented an empirical evaluation from which we noticed that our designed architecture resulted in a scalable system (as shown by early tests on the prototype presented in this paper). Forthcoming research will have three main directions:

- Integration with RDF Schema Technology
- Moving to a standard RDF Query Language (when issued by responsible committee)
- Embedding Semantic Web upper level languages, such as DAML+OIL[9], in order to deal with ontologies and reasoning.

References

[1] T. Berners-Lee, J. Hendlers and O. Lassilla, The Semantic Web Scientific American, May 2001 http://www.scientificamerican.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21&catID=2

[2] D. Beckett N-Triples EBNF Grammar definition http://mail.ilrt.bris.ac.uk/~cmdjb/2001/06/ntriples/

[3] B. McBride, Jena: A Semantic Web Toolkit, IEEE Internet Computing, Vol. 6, N. 6, 55-59, Nov/Dec 2002.

[4] S. Melnik: "RDF API Draft", working document, Stanford University, 1999

[5] COLLATE – COLLATE - Collaboratory for Annotation, Indexing and Retrieval of Digitized Historical Archive Material http://www.collate.de/

[6] S. Ferilli, Management of Cultural Heritage Material: The COLLATE project. In: L. Bordoni, G. Semeraro (Eds.), Proceedings of the Workshop on Artificial Intelligence for Cultural Heritage and Digital Libraries, 7th Congress of the Italian Association for Artificial Intelligence (AI*IA '01), Bari, 25 September 2001, pp. 29-33.

[7] H. Brocks, U. Thiel, A. Stein & A. Dirsch-Weigand, Customizable Retrieval Functions Based on User Tasks in the Cultural Heritage Domain. In: Constantopoulos, P. & Sølvberg, I.T. (Eds.). Research and Advanced Technology for Digital Libraries. Proceedings of the 5th European Conference, ECDL 2001. Berlin: Springer, 2001, pp. 37-48.

[8] Jena RDF Query Language http://www.hpl.hp.com/semweb/rdql-grammar.html

[9] Horrocks, DAML+OIL: a Reason-able Web Ontology Language, in Jensen, C. S.; Jeffery, K. G.; Pokorny, J.; Saltenis, S.; Bertino, E.; Böhm, K.; Jarke, M. (Eds.), (2002) Advances in Database Technology - EDBT 2002, Lecture Notes in Computer Science 2287, 2-13, Springer:Berlin, 2002.

[10] E.Gamma, R.Helm, R.Johnson, J.Vlissides, Design Patterns Addison-Wesley Pub Co; 1st edition (1995) ISBN 0201633612, pp. 315-324