Abstract

The Semantic Web is not widespread as it has been expected by its founders. This is partially caused by lack of standard and working infrastructure for the Semantic Web. We have built a working, portable, stable, high-performance infrastructure for the Semantic Web. This paper is focused on tasks performed by the infrastructure.

1. Suffering of the Semantic Web

The main goal of the Semantic Web is to create a universal medium for the exchange of data. The Web can reach its full potential only if it becomes a place where data can be shared and processed by automated tools as well as by people. For the Web to scale, tomorrow’s programs must be able to share and process data even when these programs have been designed totally independently [4].

Unfortunately, it seems, this goal has not yet been reached, albeit years of research by numerous researchers and large number of published standards by several standardization organizations.

We believe the Semantic Web is not yet widespread due to three prohibiting facts: missing standard infrastructure for Semantic Web operation, lack of interest from significant number of commercial subjects, and the last but not least absence of usable interface for common users.

A nonexistence of a full-blown, working, high-performance Semantic Web infrastructure is one of the main disablers for Semantic Web common use. Whereas the ‘old web’ has clearly defined infrastructure with many production-ready infrastructure implementations (e.g. Apache [1], IIS [3]), the Semantic Web has only experimental fragments of infrastructure with catastrophic scalability (e.g. Sesame [7], Jena [2]).

We have tried, during our experimental research, to convince commercial subjects to make somehow their data accessible on the Internet (of course with some reasonable level of security), and they all refused to make external access to their data. Commercial subjects do not intend to participate willingly in the ideas of the Semantic Web, because for them it either means to share their business data openly or to invest a lot of time and money for securing access to them.

Current standards in the Semantic Web area do not allow it to be used by common users. Whereas any user of WWW can easily navigate using hyperlinks in an available, production-quality WWW client, a contingent Semantic Web user has only a choice from a set of complicated query languages (e.g. SPARQL [10], SeRQL [6]). These query languages are not intended for casual users, only small number of people is able to use them, which is one of the disablers of the Semantic Web spreading.

We have concentrated our effort to the elimination of the first problem and we will describe our implementation of the Semantic Web infrastructure. Some parts are more thoroughly described in our other paper [8].

Next chapters are organized as follows: after an overview of the infrastructure there is a description of the SemWeb repository in section 3. Section 4 describes import interfaces and their implementation. Section 5 deals with semantics querying and query evaluation. Final two sections contain performance comparison and conclusions.

2. Infrastructure overview

We have recognized and described the problem of a missing, standard infrastructure for the Semantic Web in our article [12], where we have proposed a general Semantic Web infrastructure. During the last year we have made a significant progress: we have implemented full-blown, working, fast, scalable infrastructure for the Semantic Web.

The figure 1 depicts the overall scheme of our infrastructure. In this picture rectangles represent active processes, diamonds are protocols and interfaces, and gray barrels represent data-storages. All solid-line shapes depicts implemented parts of our infrastructure, whereas all dashed-line shapes represent possible experimental processes implemented by researchers playing on our infrastructure.
2.1. SemWeb repository

The heart of our infrastructure is the SemWeb repository. It is responsible for storing incoming data, retrieving results for queries, and storing the used ontology.

One of the most important parts of the SemWeb repository is the SemWeb server and it is described in the chapter 3.

Import interface enables fast, parallel data storing, and hides details about background SemWeb data-storage import capabilities.

It will be described in more details in the chapter 4. The query interface has two difficult tasks: to be independent on a query language or environment and to be independent on the SemWeb data-storage query capabilities. More about this part of our infrastructure in the chapter 5.

The last part is the SemWeb data-storage, which is responsible for holding Semantic Web data in any format (e.g. any relational database, Kowari).

2.2. Import paths

We use two distinguishable sources of data. The simplest one is a data import through importers from external data-storages. The task of importers is mapping external data-storage data-scheme to the SemWeb repository ontology. The second source of data crawls the wild Web using the Egothor web crawler [9]. The crawled web pages are stored in the Web pages data-store, where they can be accessed in parallel by deductors, which can deduce data and their ontologies from web pages and map them to our ontology. More about this path in the next chapter.

2.3. Query environments

Query environments present outputs from SemWeb repository. They make queries using query interface of the SemWeb repository and present results of the queries to users in any feasible manner. We have implemented a SPARQL compiler as an example, which translates SPARQL queries to our query interface requests.

2.4. Data-storage access

We have designed and implemented an object oriented library in C++ for a data-storage access independent on a background data-storage implementation. This library is used for a low-level access to the data in all interfaces to data-storages. It allows us to change an underlying data-storage without modifying the code of our infrastructure.

2.5. Portability

Unlike other many research projects implemented usually in Java, we have decided to implemented nearly all parts (excluding Egothor implemented in Java independently on our project) in ISO/IEC 14882 C++. The main reasons are speed, more controlled computing environment (e.g. memory management), and, although it seems absurdly comparing to Java, stability.

When properly used, using ISO C++ brings full portability among different systems and compilers. Moreover, it allows us to implement bindings to other broadly used languages e.g. Java or C#.

3. SemWeb server

The main active part of the SemWeb repository is the SemWeb server. It is a background worker that does the inferencing, making data unifications, and fulfills the task of a reasoner as well. It utilizes for its operation import and query interfaces for data manipulation.
It should be noted, that SemWeb server is not a web server in a conventional meaning. It does not fulfill any HTTP requests.

3.1. The server’s role

We believe, we do not need to have all accurate data and inferences at the moment of importing data. Alike the real world, the world knowledge changes at each moment and we are not able to catch it in one snapshot. Therefore post-processing data in the background by the SemWeb server and computing some additional data in the background is acceptable and feasible.

The SemWeb server is only a framework offering unified connection, interface and task management for experimental plug-ins, as described in the next section.

3.2. Server’s plug-ins

SemWeb server’s plug-ins are independent modules, which perform simultaneously different operations on SemWeb storage in the background. Whereas import and query interfaces are only libraries enabling the access to the SemWeb storage, the SemWeb server allows active operations upon the storage.

Each plug-in must conform to an interface requested by the SemWeb server, whereas SemWeb server offers several classes of services for plug-ins.

3.3. Implementation

Plug-ins are implemented as dynamically loaded libraries. This is an important feature, which allows selective loading and unloading of any server’s plug-in without interrupting overall infrastructure operation.

Whereas we thoroughly use C++ in the whole project, the interface requested by the SemWeb server is in C-like style, because there is currently no possibility to make a portable C++ dynamic library interface.

4. Import API

This chapter gives a brief overview of data structures and functions used to import data into SemWeb data store.

4.1. Data Structures

The basic property of the import API is a definition of internal memory structure for data insertion (sometimes called RDF Graph) and functions which provide connection to data store. The internal memory structure can be filled by both RDF triples and reifications. The content of this structure is periodically saved into the data store. When you want to save data, it is necessary to define data store type and other parameters.

4.2. Implementation

As mentioned above, we have implemented the API in C++. The data is stored in an Oracle relational database.

Some components (e.g. SemWeb server) query the data and when they deduce some formerly unknown knowledge, they insert the information back into the data store. These information is typically represented by a small number of triples, because the quality of these information is more important then their quantity. Other components (e.g. Importers) insert large amount data into the data store. The goal for these components is to import data quickly.

Conclusion: The import API for any data store should support two modes:

- Insert immediate, where data are inserted immediately when InsertTriple function is called.
- Batch insert, inserts data into a temporary space and after the import is finished, the whole data is saved into the data store.

Our implementation supports both modes.

5. Query API

The API is based on simple graph matching and relational algebra. Simple graph matching allows only one type of query. It consists of a set of RDF triples that contain variables. Result of the query is a set of possible variable mappings. This set can easily be interpreted as a relation with variable names used as a schema for the relation.

We used a relational algebra known from SQL since we wanted to make querying as simple and clear as possible and relational algebra has been around quite some time and is well known and accepted among database developers.

So far, we decided to support only some of the common relational operations. Since the schema of elementary relations (results of basic graph patterns) consists of variable names and so it is defined in the query and not in the database schema, we use only natural joins. Variable names are used to determine which columns should the join operation operate on.

5.1. Query language

We decided not to create yet another SQL-like query language. Since the query interface is intended to be used not by people but rather software, the query interface is actually a set of classes (an API). A query is basically just a tree of instances of these classes. Had we created a query
language, our form of query would basically be a derivation tree of a query in that language.

5.2. Query evaluation

We did not want to limit ourselves to just one system for data storage. Since the beginning of development we have been using four different data storages with several other in mind. Each of the systems offered different query capabilities from just evaluating all stored RDF triples to sophisticated query languages.

If we placed strong requirements on query capabilities of those systems, we could use only a few of them. On the other hand, we wanted to provide a more sophisticated query interface than that provided by all of the systems. Furthermore we wanted to use as much of the systems’ query capabilities as possible.

The contrast between complex query interface we wanted to give to the user and only very basic query capabilities provided by data storage system made it obvious that our system must be capable of evaluating the queries itself.

By implementing all operations within our system, we have reduced the requirements for the data storage engine to just one. The engine has to be able to list all stored triples. Thus the system is capable to use extremely simple storage engine that does nothing but read RDF triples from a Turtle file [5].

One of the other storage engines is a Oracle database. It would be highly inefficient to use the database only to list all triples, since we want to make some relational algebra operations on the data and Oracle have spent years optimizing these operations.

As a result, our system is capable of evaluating any query itself, but tries to use any help the storage engine can provide. This makes adding a new storage engine very simple since all that has to be done is implement listing of all stored triples. The same goes for adding new features to the query interface. Once the feature is implemented in our system, it is immediately available with all storage engines. Of course, performance of query evaluation will probably be suboptimal.

5.3. Remote query

Creating a data interface between different programming languages is not an easy task. To allow queries from languages other than C++ we created a HTTP-based API that is language independent. The client issues a HTTP GET command that contains the query to be executed and receives a HTTP response that contains the results.

To make this possible, we had to create a way to send a query over HTTP. Since we do not use any textual query language, this required some work to be done. But due to a simple tree structure of the queries serializing a query to a character string was a relatively easy task. The name of the class is stored in the string followed by the data. The same serialization algorithm is recursively called to store them to the string).

An example of such encoded query could be this:

```
Query{0;BasicGraph{Triple{Variable{x}; URI{http://www.is.cuni.cz/stoh/schema/ ot_osoba#prijmeni};Literal{Dokulil;;}}} ;x}
```

Furthermore, the format of the results has to be decided. To make the server as fast as possible the in-memory data format used within the server is used. The format is suitable to be transferred over network to different platforms - it has no little/big endian or 32/64 bit compatibility issues and it only uses small amount of information other then the actual data. Only 5 bytes plus size of a URI (encoded as UTF-8) is required to transfer the URI, 5 bytes for untyped literal, 9 bytes for typed literal, ...

We have implemented and successfully tested a C# client library.
6. Performance tests

The test environment consists of two machines. The first one hosts an Oracle database server (2x CPU Xeon 3.06 GHz with hyper-threading, DB instance was assigned 1.0 GB RAM) and the second one is an application server (2x CPU Quad-Core Xeon 1.6 GHz, 8GB RAM).

All tests used relatively large data containing 2,365,479 triples (303 MB Turtle [5] file).

6.1. Data import

Implementation should show us some bottle-necks and it would help us to find some of new improvements. The first touchstone we focused on was the speed of the data load into the data store. If we compare data stores from [11] then our test implementation has very good results. Figure 3 shows that Sesame-DB has serious performance issues when loading huge data. The load time greatly increases with the size of the input data.

Our implementation has almost linear dependency on the size of the input data. When we load a huge data, we are faster then other well known data stores, because the load time curves has evidently worse complexity than O(N).

Data were loaded in 100k triples batches [Figure 4]. Whole load took 1 hour and 54 minutes, out of which 1 hour and 14 minutes were spent transferring data from source data file to temporary tables in the database and other 18 minutes were spent on cleanup actions.

6.2. Query performance

Although we have tried to implement the algorithms used in query evaluation in an efficient manner the algorithms themselves are only basic versions so the performance of the query evaluation leaves a lot of space for improvement.

We have tested three different storage engines:
- BerkeleyDB based storage that stores triples in a B-tree
- fully in-memory engine
- Oracle-based RDF storage

Since most other RDF storages are implemented in Java, using them as a storage engine in an efficient manner is not an easy task and will be addressed in the future.

We measured the performance of evaluation of a simple query. This should give us some idea about performance of individual engines as well as the whole evaluation systems. More complex evaluation will certainly be needed and is in our immediate goals.

The BerkeleyDB-based storage engine required 1.8 seconds to complete the query, while in-memory engine took only 0.7 seconds. The performance of Oracle-based engine was the worst, requiring 6.4 seconds.

We have expected these results. The current in-memory engine is read-only and is optimized for best performance in queries similar to the one we tested. On the other hand, we used the Oracle database only to provide us with plain RDF triples and performed the join operations in our system. But this is not the main reason for the bad performance. The problem is, that the Oracle database is placed on another server and network delays for each returned triple add together. Had we used the Oracle database to join and filter the results the performance would have been much better due to smaller network traffic and better optimization of joins in Oracle. Our measurements showed that time required to evaluate this query is around 0.2 seconds.
6.3. Oracle query performance

We have performed several performance tests over our Oracle-based RDF store. The queries were completely translated to SQL and then evaluated by the Oracle server. An example of a very basic query (basic graph pattern with one triple) looks like this:

```sql
SELECT x_l.lit_rec_type AS x_kind,
       x_l.lit_value AS x_value,
       (SELECT lng_value
        FROM adt_lang
        WHERE lng_id = x_l.lit_lang_id) AS x_lang,
       (SELECT dtp_value
        FROM adt_data_type
        WHERE dtp_id = x_l.lit_type_id) AS x_type
FROM (SELECT x
       FROM (SELECT tri_subject_lit_id AS x
              FROM dat_triple t, dat_literal s, dat_uri p, dat_literal o
              WHERE t.tri_subject_lit_id = s.lit_id
              AND t.tri_object_lit_id = o.lit_id
              AND t.tri_predicate_uri_id = p.uri_id
              AND t.tri_predicate_uri_id = (SELECT uri_id
                                             FROM dat_uri
                                             WHERE uri_value = :p1_predicate)
              AND tri_object_lit_id = (SELECT lit_id
                                        FROM dat_literal
                                        WHERE lit_value = :p1_object))
       LEFT JOIN dat_literal x_l ON q.x = x_l.lit_id)
```

In the following text, some queries are called to complete instantaneously. This means, that their evaluation time was comparable to the network latency (the database resides on a different server). The queries in the following text are written as triples, where ?x denotes variable x, <uri1> denotes a URI with a value 'uri' and "value" denotes literal with value 'value'. The actual values are not given, as they are rather long and would be meaningless to the reader.

The first query consists of basic graph pattern with one triple in the form ?x, <uri>, "literal". This query returned 10 rows and evaluated instantaneously.

The second query contained two triples: ?x <uri1> "literal1", ?x <uri2> "literal2". The query evaluated instantaneously and returned one row.

Next query was ?x ?y "literal". This query required 8 seconds to evaluate and returned 4 rows. On the other hand, the query <uri> ?x ?y evaluated instantaneously returning 28 rows.

A more complex query ?x <uri1> "literal1", ?y <uri2> ?x, ?y <uri3> ?z, ?y <uri4> "literal2", ?y <uri5> ?w, ?w <uri6> "literal3" that returned only one row took as much as 200 seconds to evaluate. With the knowledge about the structure of the data, one could easily come up with an evaluation plan that would evaluate (nearly) instantaneously. But due to the way that data are stored in the database, the statistics that the Oracle server utilizes are unable to provide this. Dealing with this problem will be one of the subjects of our future research.

All queries presented so far only returned small result sets. We also measured one query ?x <uri1> ?y1, ?x <uri2> ?y2, ?x <uri3> ?y3 that returned 88964 rows. This took 70 seconds.

Another ‘big’ query was ?x <uri1> ?y, ?x <uri2> ?z and produced 184179 rows in 66 seconds.

The result of this is, that queries like “give me first and last names of all people in the database” are much slower than what they would be if the data was stored in a traditional relational database. The fact, that each triple is stored separately and table join has to be performed is one obvious factor. Less obvious but just as important is the fact, that the statistics used by the Oracle optimizer to create query evaluation plans do not work well if the data is stored like this and the optimizer makes wrong assumptions.

7. Conclusion

We have implemented and thoroughly tested the infrastructure for gathering, storing and querying semantic data. We have focused our efforts on effectivity, extensibility, scalability and platform independency. Both our experiences and benchmarks show that this goal is feasible.

Our infrastructure is currently used as a platform for further semantic web research. We expect to enhance both interfaces and functionality to support these semantic experiments.

We have two long-term goals. The first one is an implementation of a Semantic Web-specialized data-storage, which can significantly improve the behavior and performance of the SemWeb repository.

As the second long-term goal, we plan to interconnect diverse semantic repositories, possibly with different implementation. Such interface-based loosely coupled network could become a nucleus of really usable semantic web, both for academic and practical purposes.
References