Web Information Systems

Assignment 2: Part 1: The Comparison of *SPARQL* with *SeRQL*

Radim Čebiš
September 2007

In the beginning I must point out that it was really difficult decision to select query language which would be nice to compare with *SPARQL* [1]. I briefly familiarized myself with *SPARQL* and recommended *SeRQL* [2] but these languages are very similar. I even found people being confused about asking graph structure with *SQL*-like syntax which is used in both of these languages. I looked for another interesting query languages which would be more different from *SPARQL*. I found for example *Versa* [3] very interesting with it's functional approach. Unfortunately none of the found languages were so widely used like *SPARQL* or *SeRQL*. The last revision of *Versa* is from 2002 which I consider too old. Sometimes I was able to find a certain functionality which was not included in *SeRQL* nor *SPARQL* but some other functionality was missing. For example recursive queries and aggregation are supported in *Versa* [4]. I will try not just compare *SPARQL* with *SeRQL* but I will also mention missing features in both of these languages. I used *SeRQL* revision 1.2 and *SPARQL* candidate recommendation from June 2007 as referencing versions.

Firstly I will write short background of *SeRQL* because it is older language. The language is considered a second generation language because it's design goal is to implement the best features from earlier languages such as *RQL*, *RDQL* and *N3*. It is developed in collaboration between open source community and the industry. For me it is hard to say if this is better approach then standardization in *W3C* process but obviously this cooperation with industry can result in viable and successful results.

SPARQL with it's *W3C* standardization process is contrary to the *SeRQL*. First working draft is from 2004. Newest version is candidate recommendation from June 2007. It is trying to adopt similar functionality as *SPARQL*. In the working group was a man who was working for company which supports *SeRQL*. Even though it is still "under construction" there exists a lot of software implementing this query language. I think standardization leads to more developers working together which is pretty important in the web area.

I do not want to jump into conclusions so firstly I will describe the differences between *SPARQL* and *SeRQL* and in the end I will think about their impacts. The languages are pretty similar. It almost looks that only difference is little bit of syntactic sugar. For example clause *WHERE* has shifted meaning. Both languages support syntax for advanced path expressions as branching and chaining. Both use clause *CONSTRUCT* to return *RDF* graph and they offer to structure the graph according to a template. Optional paths are supported in both of the languages because it is really essential. It allows partial match on the target graph similar to outer join in *SQL*. I would say that it almost looks like there are only different terms describing same functionality in these languages. For example the syntax used to constrain variables in optional path expressions is called "nested *WHERE* clauses" in *SeRQL* but in *SPARQL* pretty same concept is described as "constraints in optional pattern matching".

But when I looked more into the depth I could find number of distinct or missing functionality. I will start with set operations. They can be considered fully supported in *SeRQL*.

There is union (*UNION* keyword), intersection (*INTESECT* keyword), set difference (*MINUS* keyword). Someone could think of missing Cartesian product but here we are not in relational algebra and it does not make any sense when we are using triples (not tuples). We can even use *UNION ALL* which gives us duplicate answers. That can lead to faster computation. It is really different story in case of *SPARQL* which supports only union operation. The rest is missing.

Other useful operations are nested queries with support of operators as *IN, ANY, ALL*, *EXISTS*. These operators and nested queries are supported in SeRQL but there is not any support in *SeRQL*.

SeRQL has also built in semantic predicates as *directSubClassOf* which can help in reasoning. *SPARQL* treats *RDF* graph as pure data and in not aware of any inference which could be provided by *RDF* store. It is interesting that even though *SeRQL* support all the set operations it does not have *ORDER BY* clause. There is no possibility to order the results. *SPARQL* uses *ORDER BY* (*ASC*|*DESC*) clause similarly to *SQL* languages. I think this is not a big issue because we can always order the results later. I did not find support for regular expressions in *SeRQL*. Regex is commonly used operator in *FILTER* clause in *SPARQL*. *SeRQL* offers us only *LIKE* clause where we can use wildcards. *SPARQL* has interesting *DESCRIBE* form which returns *RDF* graph with data about requested resource. I think this can be very useful because sometimes we do not even know the structure of the requested data. The result is determined by *SPARQL* query processor.

Both of the languages are missing computed results like *SELECT (?x + ?y)* but I think this will not be hard to implement in other versions. Unfortunately they do not support aggregation and grouping which makes them unable to compute something like "semantic closeness". *SPARQL* did not identified this as a requirement in the requirements gathering and they this requirement is now postponed because it's difficulty in open world notation. I think it is a pity that both languages do not support returning graph properties as a path or distance in graph. There is interesting paper about it [5]. I was even writing about the use of computed semantic closeness in the web navigation in my first assignment.

In conclusion I think that *SPQRQL* and *SeRQL* are pretty similar languages. In my opinion *SeRQL* supports some more functionality such as set operations and nested queries with operators as *IN* or *ALL*. I suppose *SeRQL* is little bit more mature language. But I think that the *SPARQL* will be developed further and in some time it will probably catch up with *SeRQL*. The differences are not so difficult to adjust. Maybe it will be difficult to design nested queries but if it was possible to do in *SeRQL* it will be possible to add it to the *SPARQL*. I see one advantage in *SPARQL*. It will eventually become *W3C* standard and will be implemented by a lot of programs. Even now there are many programs supporting *SPARQL*. I feel that this interoperability is and will be useful. Unfortunately neither *SPARQL* nor *SeRQL* supports functions such as aggregation, recursive operations nor more advanced graph properties. I reason that this functionality would bring new fresh air into the field and it is a pity that we cannot see how interesting things could have appeared. I would rather add these functions than improve minor differences because I would like to see new opportunities which could be achieved by these functions.

References

[1] SPARQL Query Language for RDF, 14 June 2007
(http://www.w3.org/TR/2007/CR-rdf-sparql-query-20070614/)

[2] User Guide for Sesame. Chapter 6: The SeRQL query language (revision 1.2)
(http://www.openrdf.org/doc/sesame/users/ch06.html)

[3] Versa. Revision 0.4
(http://copia.ogbuji.net/files/Versa.html)

[4] Haase P, Broekstra J, Eberhart A, Volz Raphael. A Comparison of RDF Query Languages. In: Proceedings of the Third International Semantic Web Conference, Hiroshima, Japan, 2004.
(http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/rdfquery.pdf)

[5] Angles R, Gutierrez C, Hayes J. RDF Query Languages Need Support for Graph Properties. In: Technical Report TR/DCC-2004-3, Dept. of Computer Science, Univ. of Chile , June 2004.
(http://www.dcc.uchile.cl/~cgutierr/ftp/graphproperties.pdf)

Part 2: The Comparison of *Jena* with *Sesame*

Jena [1] and *Sesame* [2] are *Java* frameworks for building semantic web applications. I think developers must use some kind of framework because it is difficult to implement all standards and protocols from scratch. Frameworks include higher level *API* for parsing and writing files of semantic data. They offer the developer practical *API* implementing useful standards or functionality such as *RDF* or *SPARQL*. All these services would cost a lot of time to implement and integrate. When we need to use them, some kind of framework comes in handy. In this work I am comparing *Jena* and *Sesame* frameworks in theirs current versions which means *Jena* in version 2.5.4 and *Sesame* in version 1.2.7.

Both frameworks were created in cooperation with industry companies and are open source projects. *Jena* grew up from work of *HP* and *Sesame* is developed and maintained by *Aduna* with help of some other contributors. Frameworks are intended for *Java* which is used by most of the semantic web development tools [3].

Sesame supports local and remote repositories of *RDF* data. It also offers the user an *RDF* server which can be asked to provide data via *HTTP*. Repositories can be saved into memory for fastest access but usually some kind of *RDBMS* is used for persistence. Currently the layer responsible for storage (*SAIL*) supports *PostgreSQL*, *MySQL*, *Microsoft SQL Server* and *Oracle* databases. *Sesame* framework includes parsers for most widely used representations of semantic information such as *N3*, *N-Triple*, *XML/RDF* and *Turtle*. All these formats are also included in *Jena* framework. Supported query languages are *RDQL*, *RQL* and *SeRQL*. The last mentioned language is developed as a part of *Sesame* which means that the implementation is very well aligned with specification. Interface for manipulating the data in *RDF* store is separated in it own package. *Sesame* offers reasoning based on *RDFS*. Fortunately there are extensions for *Sesame* which for example offer *OWL DL* reasoning. External contributors also created extensions which allow us to use higher *API* for communication with *Sesame* server from languages as *Ruby*, *Pearl*, *PHP5* or *Delphi*.

Jena is similar to *Sesame* but in my opinion it gives the developer more robust *API*. I think it stress more on capability to integrate with other code. In memory persistence and *RDBMS* persistence are supported too but *Jena* is prepared to cooperate with more database systems (*HSQLDB*, *MySQL*, *PostgreSQL*, *Derby*, *Oracle*, *Microsoft SQL Server*). I feel that missing support of *HSQLDB* and *Derby* in *Sesame* is not important because these are "small" databases and usually we would choose other ones. But maybe it shows that *Jena* tries to encourage interoperability with more software. Remote server can be also used. It is called *Joseki* [4] and is developed as a sub-project of *Jena*. It provides *HTTP* interface to *RDF* data and fully supports *SPARQL*. Unfortunately it still lacks update protocol.

As I mentioned before, *Jena* includes parsers for same kinds of serialization as in *Sesame*. As a query language the *SPARQL* is used. The query engine is implemented as a module which can mean that it will be possible to offer other query languages even though *SPARQL* should be sufficient in most cases. Of course *Jena* provides an *API* for manipulating *RDF* graphs and other models such as interference model. Graphs are represented as abstract "models" which can be populated with data from files, databases, URLs or a combination of these. *Jena* also provides a set of abstractions and convenience classes for accessing and manipulating ontologies represented in *RDF*.

Unfortunately *Jena* and *Sesame API* for manipulating *RDF* data are not compatible. This results in programs that are tightly coupled with used framework. This situation happened because there is no *RDF API* specification accepted by *Java* community as for example *DOM API* (*org.w3c.dom*). Luckily there are developers who needed to make *Jena* and *Sesame* interoperable so *Jena Sesame Model* [5] and *Sesame Jena Adapter* [6] emerged.

I see the biggest strength of *Jena* in it's reasoning capabilities and ontology *API* which tries to provide a language-neutral consistent interface. The framework has it's own various internal reasoners such as transitive, *RDFS* rule and incomplete *OWL* reasoners. But the most important thing is that *Jena* provides support for external reasoners through *DIG* (*Description Logic Interface*) [7]. This subsystem is allowing a range of inference engines to be plugged into *Jena*. In addition the *Pellet* (*OWL DL*) [8] reasoner can be plugged in even without using *DIG* interface. This enables us to use variety of reasoners. For example *Racer* [9] and *FaCT++* [10] are *DIG* compliant reasoners.

Finally my opinion is not so clear. *Jena* provides really robust *API* and it's support of reasoning system can be very helpful in some applications. On the other hand I like the user manual of *Sesame* which is more solid than in *Jena*. I consider *Jena* more opened as it supports *SPARQL* and it seems as it is trying to stress more on interoperability and connectivity. In the end the selection of framework will always depend on the specific application which makes comparison really difficult because only option how to compare is to write the same application twice with help of diverse frameworks. That is maybe why there are no comparisons of these frameworks. It is hard to see usability of *API* just by reading *JavaDoc* and no one is so interested to try to write same application with use of different frameworks which is understandable. I thought about it but of course there is not enough time and I hope that I will get to actual use of one of these frameworks in the final assignment. I would probably use *Jena* framework for big complicated applications which could profit from it's robust *API* and great reasoning support. For smaller applications the *Sesame* framework might be more suitable because I think the time spent by learning it's properties could be shorter which can make a big difference in smaller projects.

There are more attributes which should be taken into account before decision about the framework. Fortunately the *RDBMS* is not the case because the support of them is equivalent but for example we should think about which query language is more appropriate for our application because when we select Sesame we will not be able to use *SPARQL* and otherwise when we select *Jena* we will not be able to use *SeRQL*. This really depends on concrete application because from the previous part concerning comparison of query languages we can deduce that *SeRQL* is in some cases more powerful. Also in some cases we want to implement some kind of lightweight client, possibly web based. Than the extensions for *Sesame* providing support in scripting languages as *PHP5* and *Pearl* can come in handy.

As mentioned before the selection depends on the application. In advance I would choose *Jena* framework because I think I could learn a lot from it's architecture and because it supports *SPARQL* which will become standard and I hope the experience with this language could be used somewhere else in the future.

References

[1] Jena Documentation
(http://jena.sourceforge.net/documentation.html)

[2] Home of Sesame
(http://www.openrdf.org/)

[3] Developers Guide to Semantic Web Toolkits for different Programming Languages
(http://sites.wiwiss.fu-berlin.de/suhl/bizer/toolkits/)

[4] Joseki Documentation
(http://www.joseki.org/documentation.html)

[5] Java Sesame Model Homepage
(http://users.ecs.soton.ac.uk/wf/jsm.htm)

[6] Sesame – Jena Adapter Homepage
(http://sjadapter.sourceforge.net/)

[7] DIG Interface Homepage
(http://dig.sourceforge.net/)

[8] Pellet Homepage
(http://pellet.owldl.com/)

[9] Racer Homepage
(http://www.racer-systems.com/)

[10] FaCT++ Homepage
(http://owl.man.ac.uk/factplusplus/)

Also *JavaDoc* included in *Sesame* and *JavaDoc* included in *Jena* were used.