RDF* and SPARQL* Usability Study

Paul Warren, paul.warren@open.ac.uk; *Paul Mulholland*, paul.mulholland@open.ac.uk

N.B. this handout is to be read prior to undertaking the study, and to be retained for reference at any time during the study. The final page contains a summary of all you need to know.

1 Introduction

The purpose of this study is to understand the difficulties people experience using a data modelling format known as RDF* and the corresponding query language, SPARQL*. These are extensions of the W3C standards, RDF and SPARQL[1]. No prior knowledge or experience is required for the study; all the necessary features are explained in this handout. The object is to understand how people manipulate the data model and language constructs mentally, so please do not use pen and paper.

The structure of this handout is as follows. Section 2 describes RDF, and Section 3 describes how RDF has been extended to create RDF*. Section 4 then provides an overview of SPARQL. Sections 5 and 6 describe some specific features of SPARQL which are used in this study. Section 7 describes SPARQL*, the extension of SPARQL to query RDF*. Section 8 gives an overview of how the study is organised. Finally, there is a one-page summary which provides all the necessary information in compact form.

2 RDF

This section describes a simplified subset of the RDF data model, as used in the study questions. The basic component of the data model is a triple, consisting of *subject*, *predicate* and *object*. The subject and predicate will be resource identifiers, composed of a colon followed by any combination of letters and digits, e.g. *:Stephen*, *:marriedTo*, *:1801*, *:year1801*. In all subsequent discussion these will be referred to simply as 'identifiers'. The object may also be an identifier. An example of such a triple is:

:Stephen :marriedTo :Mary .

As an alternative, the object may be a number or a character string. For the purposes of this study, the former will be written as an integer in the normal way; the latter will be enclosed in single quotes, e.g.

:Stephen :hasAge 32 .
:Mary :role 'lawyer' .

Note that in the example given in the first paragraph of this section, the predicate (*:marriedTo*) connects two identifiers (*:Stephen* and *:Mary*). On the other hand, in the examples in the second paragraph, the predicates are assigning attributes (age and job role) to the two identifiers; the attributes themselves are either numbers or character strings. Note that numbers can also be used to represent dates (e.g. *1801*). Note also the difference between *1801*, which is a number; and *:1801*, which is an identifier.

[1] Hartig, Olaf, and Bryan Thompson. 'Foundations of an Alternative Approach to Reification in RDF'. ArXiv Preprint ArXiv:1406.3399, 2014.

A database consists of a set of such triples. In our study, triples are each written on a separate line, with a period (full stop) at the end of each line. The same identifier can occur in more than one triple, e.g. a simple database might consist of the following triples:

:Stephen :marriedTo :Mary .
:Stephen :hasAge 32 .
:Mary :role 'lawyer' .
:John :brotherOf :Mary .
:John :role 'accountant' .
:Mary :worksFor :BigCo .
:John :worksFor :BigCo .

A database such as the one above can be visualized as a graphical structure. The identifiers (*:Stephen, :Mary, :John, :BigCo*) can be regarded as nodes. The predicates *:marriedTo* and *:brotherOf* can be regarded as edges joining nodes. The predicates *:hasAge* and *:role* associate attributes with nodes.

3 RDF*

RDF* is an extension to RDF which permits metadata to be associated with a triple. For example, we may wish to say that Mary has worked for BigCo from 2000. We do this by writing the triple associating Mary and BigCo, enclosed within double angled brackets, within another triple:

<<:Mary :worksFor :BigCo>> :from 2000 .

In this example, the object of the inner triple is an identifier whereas the object of the outer triple is a number. However, either object could have been an identifier, number or character string. Another example would be:

<<:Mary :worksAs 'lawyer'>> :for :BigCo .

In principle, triples can be nested to any level of depth. In this study, triples are nested to a maximum of two levels, e.g.:

<<<<:Mary :worksFor :BigCo>> :role 'lawyer'>> :from 2000 .

This is an example of the use of all three types of object, i.e. identifier (*:Mary*), character string (*'lawyer'*), and number (here representing a date, *2000*). The example is indicating that Mary has had the role of a lawyer with BigCo from 2000.

4 Overview of SPARQL

SPARQL is a language for querying RDF databases. Each query begins with the SELECT keyword, followed by the list of variables which we wish to output, e.g.

SELECT ?person1 ?person2

This is followed by a WHERE clause, i.e. the WHERE keyword and a set of 'triple patterns' enclosed in curly brackets. In the simplest case, there is only one triple pattern e.g.

WHERE { ?person1 :brotherOf ?person 2}

In the database in section 2, the solution to this query will be:

?person1 = :John; ?person2 = :Mary

Note that, as with triples in the database, a triple pattern in a query consists of a subject, predicate and object. These may be variables, e.g. *?person*, or identifiers in the database, e.g. *:Stephen*, *:brotherOf*. An object may also be a number or character string, e.g.

SELECT ?person
WHERE {?person :hasAge 32}

The query will output *?person = :Stephen*.

The solution to a variable can also represent a number or character string, e.g.

SELECT ?age
WHERE {:Stephen :hasAge ?age}

Here, the solution is:

?age = 32

Sometimes we have two or more triples which share variables, and we need all triples to be valid. Returning to our example database, assume we want to know who is Stephen's wife and what company she works for:

SELECT ?wife ?company
WHERE { :Stephen :marriedTo ?wife . ?wife :worksFor ?company }

The dot between the triple patterns indicates that both must be satisfied; so that the same identifier (represented by *?wife*) has to be the object of both the triples. The solution is:

?wife = :Mary, ?company = :BigCo

We could also write the two triple patterns the other way around:

WHERE {?wife :worksFor ?company . :Stephen :marriedTo ?wife}

The variables written after the SELECT keyword must appear in the WHERE clause. However, there may be variables in the WHERE clause which do not appear after the SELECT keyword. These are variables which are needed to create the query, but which we do not wish to output. In the previous example, we may not be interested in Stephen's wife, only her company. We can write:

SELECT ?company
WHERE {:Stephen :marriedTo ?wife . ?wife :worksFor ?company }

This query will output *?company = :BigCo.*

There may be more than one solution for a variable, or for a combination of variables. For example, we may wish to know what information the database contains about people and their roles:

SELECT ?person ?role
WHERE {?person :role ?role}

Using the database described in Section 2, there are two sets of solutions:

?person = :Mary; ?role = 'lawyer'
?person = :John; ?role = 'accountant'

5 More about predicates in SPARQL

This section is about three operators which can be used with predicates in SPARQL. Firstly, we can use the concatenation operator, represented by the slash symbol (/) to concatenate

predicates. In the previous section we showed a query which outputted the company for which Stephen's wife works. It used the variable *?wife*, which we did not wish to output and was not in the SELECT clause. We can avoid using this variable, and create a more compact query by using the concatenation operator:

SELECT ?company
WHERE {:Stephen :marriedTo / :worksFor ?company}

Here *:Stephen* is linked to *?company* by a chain of predicates consisting of *:marriedTo* and *:worksFor*.

Another operator we use is the reverse operator, represented by a hat symbol (^). This reverses the directionality of a predicate. For example, we may wish to know who John is the brother of. We can do this with the following query:

SELECT ?person
WHERE {:John :brotherOf ?person}

However, we can achieve the same thing with the query:

SELECT ?person
WHERE {?person ^:brotherOf :John}

Here, the directionality of *:brotherOf* is reversed. Of course, in this example there is no need to introduce the extra complication of the reverse operator; one might as well use the first query. However, the ability to reverse the directionality of a predicate comes into its own when we use the reverse operator within a chain of concatenated predicates. For example, we may wish to know who is the brother-in-law of Stephen. We can do this without using the reverse or concatenation operators:

SELECT ?brotherInLaw
WHERE {:Stephen :marriedTo ?person . ?brotherInLaw :brotherOf ?person}

Here, there are two triple patterns connected by a dot, so they must both hold for a valid solution. However, we can avoid using the variable *?person* and create a more compact query, by using the reverse and concatenation operators together:

SELECT ?brotherInLaw
WHERE {:Stephen :marriedTo / ^:brotherOf ?brotherInLaw}

This query would return *?brotherInLaw = :John*.

A final feature relating to predicates, which we use in our study, is the ability to repeat a given predicate any number of times. We do this by suffixing a predicate with a plus symbol (+). We can illustrate this with a new database illustrating the relationship between animal groups, with just two triples:

:Monkey :subGroupOf :Primate .
:Primate :subGroupOf :Mammal .

We wish to know all the subgroups, and sub-subgroups of *:Mammal*. We can write the query:

SELECT ?subgroup
WHERE {?subgroup :subGroupOf+ :Mammal}

The plus symbol means that the predicate can appear once, or be concatenated with itself an unlimited number of times, i.e.

:subGroupOf
subGroupOf / subGroupOf
subGroupOf / subGroupOf / subGroupOf
…

Hence, the query will output:
?subgroup = :Primate; ?subgroup = :Monkey

Here, *:Primate* is a subgroup of *:Mammal*, and *:Monkey* is a subgroup of *:Primate*, and hence a sub-subgroup of *:Mammal*. If the database contained a subgroup of *:Monkey*, that would also be returned, and so on indefinitely.

6 Three functions: STR(), CONTAINS(), and FILTER()

We have already made a distinction between an identifier, e.g. *:London* and a character string, e.g. *':London'*. We may wish, however, to make comparisons between identifiers and character strings. This requires the use of three functions.

The first of these functions is *STR()*, which we use to extract the character string of which the identifier is composed. Thus *STR(:London*) returns the character string *':London'*.

We may also wish to check whether a particular character string is contained within another character string. We can do this using the *CONTAINS()* function. This function has two string arguments, and returns a logical true if the second argument is contained within the first, and a logical false otherwise. For example, the following will return a logical true:
CONTAINS(':London', 'London')

The following will return logical false:
CONTAINS(':London', 'Edinburgh')

Note that one or both of the arguments could be variables (i.e. beginning with a question mark) representing character strings.

The function FILTER takes one argument which is either true or false. It has the effect of removing any solution for which the argument evaluates to false.

As an example of how these three functions can work together, consider the database:
:Joe :livesIn 'London' .
:Fred :livesIn 'Edinburgh' .

Then consider the following query:
SELECT ?person
WHERE {?person :livesIn ?city . FILTER(CONTAINS(STR(:London), ?city))}

The first triple pattern in the WHERE clause permits two possibilities: *?person = :Joe*; and *?person = :Fred*. However, of the two possibilities for *?city*, only the string '*London'* is contained within the string *':London'*. Hence, only the solution *?person = :Joe* is valid, because only Joe lives in London.

In summary, with the use of these three functions, we can compare identifiers (*e.g. :London*) to character strings (e.g. *'London'*).

7 SPARQL*

In section 2 we introduced RDF*, an extension of RDF which permits triples to be embedded within other triples. In order to fully query RDF*, we need to analogously extend SPARQL to form SPARQL*.

Imagine we have a database containing:

<<:Mary :worksFor :BigCo>> :role 'lawyer' .

We may wish simply to know for whom Mary works. This we can do with a SPARQL query:

SELECT ?company
WHERE {:Mary :worksFor ?company}

This will give the solution *?company = :BigCo*. This is because any embedded triple in RDF* is regarded as valid. Put another way, the triple above can be regarded as two triples:

<<:Mary :worksFor :BigCo>> :role 'lawyer' .
<<:Mary :worksFor :BigCo>> .

In the same way as RDF* can embed triples within triples, using double angled brackets, so SPARQL* can embed triple patterns within triple patterns using double angled brackets. For example, if we wish to know both the company Mary works for and her role in that company, we need the SPARQL* query:

SELECT ?company ?role
WHERE {<<:Mary :worksFor ?company>> :role ?role}

Just as RDF* can embed triples to any level of nesting, so SPARQL* can embed triple patterns to any level of nesting. Imagine we have a database containing:

<<<<:Mary :worksFor :BigCo>> :role 'lawyer'>> :from 2000 .
<<<<:Mary :worksFor :BigCo>> :role 'manager'>> :from 2010 .

We may wish to know for whom Mary works, what roles she has had, and when she began each role. An appropriate query would be:

SELECT ?company ?role ?startdate
WHERE {<<<<:Mary :worksFor ?company>> :role ?role>> :from ?startdate}

This would output two solutions:

?company =:BigCo, ?role = 'lawyer', ?startdate = 2000
?company =:BigCo, ?role = 'manager', ?startdate = 2010

Note that the order in which the predicates appear in the WHERE clause must reflect the order in the predicates in the RDF*. The following query would return no solutions:

SELECT ?company ?role ?startdate
WHERE {<<<<:Mary :worksFor ?company>> :from ?startdate>> :role ?role}

8 Overview of study

The study consists of 15 questions, the first two of which are practice questions to get you used to the format. Your responses to these will not be used in the subsequent analysis. There are two types of questions: modelling questions and querying questions.

In the modelling questions you are presented with some information in English; one or more queries, in English, that one might want to ask; and some alternative models in RDF*. You are asked to indicate which of these models are correct. Here, 'correct' means that the RDF* model accurately represents the information and that SPARQL* queries can be constructed to represent the English queries. You are also asked to rank the models. You can use whatever criteria you wish to perform this ranking, and it is possible to give two or more models the same rank. Optionally, you can add free format textual comment for each model.

In the querying questions you are provided with a model in RDF*; a query in English; and some alternative queries in SPARQL*. You are asked to indicate which of the SPARQL* queries correctly represent the English query, when applied to the RDF* model.

For each question, there may be any number of correct responses. None may be correct; or they may all be correct; or some may be correct and some may be incorrect. For each proposed response, you will need to click to indicate 'correct' or 'incorrect'. You can change your response, by clicking on the alternative button, at any time until you click on 'Submit and continue' at the bottom right. When you do click on 'Submit and continue', you move on to the next page. It is not possible to return to a previous page. Please do not attempt to do so with the browser 'back' button. Finally, please note that this project has been reviewed by, and received a favourable opinion from, The Open University Human Research Ethics Committee, reference HREC/3568.

THE FOLLOWING PAGE SUMMARISES EVERYTHING YOU NEED TO KNOW

Summary

Conventions used for writing RDF* and SPARQL* in this study

- Identifiers are composed of a colon, followed by any combination of letters and digits, e.g. *:John, :brotherOf, :1801, :year1801*.
- Character strings are enclosed in quotes, e.g. *'lawyer'*.
- Numbers, including years, are written in the normal way as integers, e.g. *2010*.
- SPARQL* keywords and functions are written in capitals, e.g. *SELECT* and *STR()*.
- Variables begin with a question mark, e.g. *?name*. Note that variables can represent identifiers, character strings or numbers.

RDF triples

- RDF triples consist of subject, predicate and object. Subjects and predicates are identifiers. In this study, objects are identifiers, character strings or integers.

Overview of SPARQL

- *SELECT* clause indicates which variables are to be output, e.g. *SELECT ?person*
- *WHERE* clause indicates conditions to be satisfied, e.g.
  WHERE {:Mary :worksFor ?company}
- A dot between triple patterns in a WHERE clause means that both must be valid, e.g.
  WHERE { :Stephen :marriedTo ?wife . ?brotherInLaw :brotherOf ?wife }

Predicate operations

- / concatenates predicates in a *WHERE* clause, e.g.
  WHERE {:Stephen :marriedTo / :worksFor ?company}
- ^ reverses the directionality of a predicate, e.g.
  ?person ^:brotherOf :John is equivalent to *:John :brotherOf ?person*
- + repeats a predicate an unlimited number of times
  ?subgroup :subGroupOf+ :Mammal means that *?subgroup* can be a subgroup of :Mammal, or a sub-subgroup, or a sub-sub-subgroup etc.

SPARQL functions

- *STR()* converts an identifier into a character string, e.g.
  STR(:London) returns the string *':London'*
- *CONTAINS* returns a logical true if the first argument contains the second, false otherwise.
- *FILTER()* means that the query will only output values which make its argument true, e.g.
  FILTER(CONTAINS(STR(:London), ?city)) means that *?city* must represent a character string contained within the string *':London'*.

Embedded triples and triple patterns

- Embedded triples in RDF* are placed within double angled brackets, e.g.
  <<:Mary :worksFor :BigCo>> :role 'lawyer' .
  N.B. this is considered as two triples, i.e. the inner triple also holds.
- Triples can be embedded to any level of nesting, e.g.
  <<<<:Mary :worksFor :BigCo>> :role 'lawyer'>> :from 2000 .
- Embedded triple patterns in SPARQL* are also placed within double angled brackets, e.g.
  WHERE {<<:Mary :worksFor ?company>> :role ?role}
- SPARQL triple patterns can also be embedded to any level of nesting, e.g.
  WHERE {<<<<:Mary :worksFor ?company>> :role ?role>> :from ?startdate}