Informal introduction to SPARQL

Data depicted by RDF can be viewed as a graph database i.e. what users want to query about are the graph elements, nodes or edges(both are RDF terms). SPARQL is a query language for RDF, of which most forms contain a set of triple patterns called a basic graph pattern to specify the desirable RDF terms.

Since the RDF terms to query must be unknown, they are represented by varibles in SPARQL, with identifiers starting with '?' or '$'. Note that the '?' or '$' is not part of the variable name and thus not shown in the result.

Querying certain RDF terms means these terms must involve at least one of the triples(if there are not, then the wanted RDF term has no relationships with any other RDF terms in the graph, i.e. the RDF term is meaningless in this RDF graph). Therefore, we use a group of triples, or a graph pattern, to specify the wanted terms.

we shall see this by an example:

```
PREFIX ex:  <http://example.org/>
SELECT ?title ?author
WHERE  {
?book ex:publishedBy  <http://crc-press.com/uri> .
?book ex:title        ?title .
?book ex:author       ?author.}
```

Generally speaking, a SPARQL query can be divided into 3 parts:

1. the first part defines the namespace using the keyword PREFIX
2. the second part determines the result format(SELECT, CONSTRUCT, ASK, DESCRIBE)
3. the thrid part specifies the graph pattern using WHERE clause

Note that IRIs in SPARQOL are surrounded by angle brackets <>.

The WHERE clause specifies what relationships the variables should have with others. Variables can appear in a triple as the subject, object and the predicate. The format in WHERE clauses are very similar to Turtle: the semicolon ';' and the comma ',' can be used in the same meaning.

Consider the RDF graph

```
@prefix  ex:   <http://example.org/> .
```

```
@prefix  book: <http://semantic-web-book.org/uri/> .
ex:SemanticWeb ex:publishedBy  <http://crc-press.com/
uri>;
               ex:title "Foundations of Semantic Web
Technologies" ;
               ex:author book:Hitzler, book:Krötzsch,
book:Rudolph .
```

The result of the previous SELECT query example may look like:

title	author
"Foundations of..."	http://semantic-web-book.org/uri/Hitzler
"Foundations of..."	http://semantic-web-book.org/uri/Krötzsch
"Foundations of..."	http://semantic-web-book.org/uri/Rudolph

where each row correspond to a **solution**, an assignment for variables which match the graph pattern listed in WHERE clause(though it may be not complete; i.e. some variables may have not assignments).
If an RDF term r is assigned to a variable v in a row, then we say the variable v is **bound** in r, and there is a **binding** (v , r) in this row.

To list all the variables in the result, we can use the wildcard '*' instead of specifying all the variable identifiers in the SELECT clause.

Group Patterns, Options and Alternatives

Sometimes it's desirable to query terms satisfying at least one of the specified graph patterns, or see values of certain properties if the term has. To accomplish this, there must be a way to separate the relationships which must be meet and the rest. In SPARQL, we can group and separate triples by curly braces {}:

```
PREFIX ex:  <http://example.org/>
SELECT ?title ?author
WHERE  { { ?book ex:publishedBy  <http://crc-press.com/
uri> .
?book ex:title       ?title }
{}
?book ex:author       ?author
}
```

here there are 3 basic graph patterns, which together form a **complex graph pattern**. In this example, the WHERE clause is just the same as

```
{
?book ex:publishedBy  <http://crc-press.com/uri>;
ex:title       ?title;
ex:author       ?author.
}
```

we can use the keyword OPTIONAL to allow information to be added to the solution where the information is available:

```
pattern1 OPTIONAL { pattern2 }
```

The OPTIONAL matching provides this facility: if the optional part does not match, it creates no bindings but does not eliminate the solution. Typically there are some variables in pattern2 which also appear in the SELECT clause.

e.g. with RDF graph input

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .
@prefix rdf:        <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

_:a  rdf:type        foaf:Person .
_:a  foaf:name       "Alice" .
_:a  foaf:mbox       <mailto:alice@example.com> .
_:a  foaf:mbox       <mailto:alice@work.example> .

_:b  rdf:type        foaf:Person .
_:b  foaf:name       "Bob" .
```

and the SPARQL query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x  foaf:mbox  ?mbox }
       }
```

the result is

name	mbox
"Alice"	<mailto:alice@example.com>
"Alice"	<mailto:alice@work.example>
"Bob"	

The keyword OPTIONAL is **left-associative.** A query containing multiple OPTIONAL patterns are checked independently.

For alternative graph patterns, the keyword UNION is used :
`pattern1 UNION pattern2`
Any assignment satisfy at least one of the pattern would be a solution in the result set. For SELECT query, the result set is the combination of pattern1 and pattern2. For example, for RDF graph input

```
@prefix dc10:  <http://purl.org/dc/elements/1.0/> .
@prefix dc11:  <http://purl.org/dc/elements/1.1/> .

_:a  dc10:title     "SPARQL Query Language Tutorial" .
_:a  dc10:creator   "Alice" .

_:b  dc11:title     "SPARQL Protocol Tutorial" .
_:b  dc11:creator   "Bob" .

_:c  dc10:title     "SPARQL" .
_:c  dc11:title     "SPARQL (updated)" .
```

and the query

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc11:  <http://purl.org/dc/elements/1.1/>

SELECT ?x ?y
WHERE  { { ?book dc10:title ?x } UNION { ?book dc11:title  ?y } }
```

The result set is

x	**y**
	"SPARQL (updated)"
	"SPARQL Protocol Tutorial"
"SPARQL"	
"SPARQL Query Language Tutorial"	

UNION is also left-associative.
The priority of UNION is the same as OPTIONAL. So
`{ {s1 p1 o1} OPTIONAL {s2 p2 o2} UNION {s3 p3 o3}}`
is equivalent to
`{ {{s1 p1 o1 } OPTIONAL {s2 p2 o2 }} UNION {s3 p3 o3}}`

Blank nodes in WHERE clause

blank nodes in SPARQL graph patterns act as non-distinguished variables, not as reference to specific blank nodes in the data graph being queried.
They are indicated by either the label form like “_:a” or the abbreviated form “[p o]”, where p is a predicate and o is an object.

In the query side,
since the design of blank nodes are to show the existence of certain resources in a triple, they are somehow treated like variables, i.e., they can be replaced by arbitrary elements of the given RDF graph. But blank nodes can not appear in the SELECT clause, and one blank node identifier can not appear in two basic graph patterns.

create a blank node in the WHERE clause:
```
_:b57 p1 o1 .
```
then the blank node _:b57 can be used in that group(basic graph pattern), like
```
s  p2  _:b57 .
```
If a temporary identifier is also unnecessary, we can just use
```
s p2 [p1 o1] .
```

In the result side,
query variables might be instantiated by blank nodes, so blank nodes may appear in the result. In the result set, same blank node identifiers do refer to the same blank node, but the blank node identifiers in the result set are irrelevant to that in the original RDF graph.
e.g. the following two result sets are equivalent

subj	value
_:a	"for"
_:b	"example"

subj	value
_:y	"for"
_:g	"example"

Queries with Data values

Consider an example data input:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex:  <http://example.org/> .
ex:s1  ex:p  "test" .
ex:s2  ex:p  "test"^^xsd:string .
ex:s3  ex:p  "test"@en .
ex:s4  ex:p  "42"^^xsd:integer .
ex:s5  ex:p  "test"^^<http://example.org/datatype1>" .
```

and consider the query pattern

```
{?subject <http://example.org/p> “test” .}
```

SPARQL rigorously distinguish among typed and untyped literals, and this query would only return ex:s1 as the result.

Most of the SPARQL implementations support the datatypes in the namespace http://www.w3.org/2001/XMLSchema# , and the match for values of these type is down to the value level. For example,

```
{ ?subject  <http://example.org/p>
"042"^^xsd:integer . }
```

will return ex:s4 as the result despite the difference of the lexical form.
For unsupported datatype however, both the lexical form and the datatype IRI are compared with the one in the query, and a match will be found only if both agree exactly.

SPARQL also provides syntactic abbreviations for some particularly common datatypes, like xsd:integer, xsd:decimal, xsd:boolean (true and false).

Filters

Sometimes we might add additional, more detailed requirements for the wanted RDF terms, like a range specification. All the features of SPARQL are not enough to provide such functions, and this is what the FILTER clause does:
SPARQL `FILTERs` restrict the solutions of a graph pattern match according to a given expression. They eliminate any solutions that, when substituted into the expression, either result in an effective boolean value of `false`(see below) or produce an error.

Let’s start from an easy example:

```
PREFIX ex: <http://example.org/>
SELECT ?book WHERE
  { ?book    ex:publishedBy  <http://crc-press.com/uri> .
    ?book    ex:price        ?price
    FILTER (?price < 100)
}
```

This query retrieves all books that were published by CRC Press, and for which the property ex:price has been assigned a value that is smaller than 100(here 100 is an abbreviation for "100"^^xsd:integer). The content in FILTER() is an expression defined by https://www.w3.org/TR/rdf-sparql-query/#rExpression .

The effective boolean value(EBV) is defined as follow:

- The EBV of any literal whose type is `xsd:boolean` or numeric is false if the lexical form is not valid for that datatype (e.g. "abc"^^xsd:integer).
- If the argument is a typed literal with a datatype of `xsd:boolean`, the EBV is the value of that argument.
- If the argument is a plain literal or a typed literal with a datatype of `xsd:string`, the EBV is false if the operand value has zero length; otherwise the EBV is true.
- If the argument is a numeric type or a typed literal with a datatype derived from a numeric type, the EBV is false if the operand value is NaN or is numerically equal to zero; otherwise the EBV is true.
- All other arguments, including unbound arguments, produce a type error.

The scope of a FILTER

Filter conditions always refer to the whole group graph pattern(including OPTIONAL and UNION graph patterns) within which they appear. The position of a filter in a group doesn't matter.

Operators in FILTER

As opposed to graph patterns, filters are not strictly based on the RDF data model, but may contain any requirement that can be checked computationally. A series of operators can be used in the FILTER expression, listed in the following link: https://www.w3.org/TR/rdf-sparql-query/#OperatorMapping

Some operators deserve further explanation.

1. = and sameTerm

The expression `term 1= term2` returns true iff they are term-equal. It produces an error if there's a unsupported datatype or two arguments are not term-equal literals[1]. In the rest cases, it returns false. Note that "!=" is implemented by taking the negation of "=".

[1] https://www.w3.org/TR/rdf-sparql-query/#func-RDFterm-equal-foot1

`sameTerm(term1, term2)` returns true iff two terms are term-equal. Otherwise, it returens false. That is, `sameTerm` can function for unsupported type.

2. `lang` and `langMatches`
`lang()` returns an empty string if the argument has no language tag.
`langMatches(language-tag, language-range)`
language settings may have hierarchical structure, combination of LANG() and = may be not convenient enough to check a language;
e.g. `lang("Test"@en-GB) = "en"` returns false, but the alternative condition `langMatches(lang("Test"@en-GB), "en")` returns true. Moreover, langMatches can used to detect the existence of language tags by using '*':
`langMatches(lang(?title), "*")`

erros in FILTER
see
https://www.w3.org/TR/rdf-sparql-query/#evaluation

Other result format: CONSTRUCT, ASK, DESCRIBE

CONSTRUCT : return an RDF document
this keyword needs a template for the generated result. For example,

```
PREFIX ex: <http://example.org/>
CONSTRUCT { ?person  ex:mailbox     ?email .
            ?person  ex:telephone   ?phone . }
WHERE { ?person  ex:email   ?email .
        ?person  ex:phone   ?phone . }
```

The returned RDF document would consist of 0 or more templates with each instantiated by a solultion. If any such instantiation produces a triple containing an unbound variable or an illegal RDF construct, such as a literal in subject or predicate position, then that triple is not included in the output RDF graph.

The graph template can contain triples with no variables (known as ground or explicit triples), and these also appear in the output RDF graph returned by the CONSTRUCT query form.

Blank nodes in templates play a special role when constructing RDF results. They do not represent single blank nodes in the final result, but are replaced by new blank nodes for each of the result's variable assignments – i.e. for each table row in SELECT format. This can be illustrated by the next query:

```
PREFIX ex: <http://example.org/>
CONSTRUCT { _:id1  ex:email   ?email .
```

```
            _:id1  ex:phone   ?phone .
            _:id1  ex:person  ?person . }
WHERE { ?person  ex:email   ?email .
        ?person  ex:phone   ?phone . }
```

a result may be

```
_:a ex:email "alice@example.org" ;
   ex:phone "123456789" ;
   ex:person ex:Alice .
_:b ex:email "alice@example.org" ;
   ex:phone "987654321" ;
   ex:person ex:Alice .
_:c ex:email "a_miller@example.org" ;
   ex:phone "123456789" ;
   ex:person ex:Alice .
_:d ex:email "a_miller@example.org" ;
   ex:phone "987654321" ;
   ex:person ex:Alice .
```

Note that the blank node identifiers in the result is independent of those in RDF input.

```
PREFIX ex: <http://example.org/>
CONSTRUCT { ex:Query   ex:hasResults  "Yes" . }
WHERE { ?person  ex:email   ?email .
        ?person  ex:phone   ?phone . }
```

This query returns a predefined RDF graph with a single triple whenever the query has any solutions, and an empty document otherwise.

ASK : return yes or no
To simply find out whether a query has any results(solutions), however, SPARQL offers a simpler solution. Namely, queries with the key word ASK only return “true” or “false,” depending on whether or not some result matches the query conditions.
For example, consider the data input

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice" .
_:a  foaf:homepage   <http://work.example.org/alice/> .

_:b  foaf:name       "Bob" .
_:b  foaf:mbox       <mailto:bob@work.example> .
```

and the query:

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
ASK  { ?x foaf:name  "Alice" }
```

then the result is

```
yes
```

DESCRIBE: returns an RDF graph containing RDF data about resources
deciding which information is “relevant” for describing a particular object is of course strongly application dependent, and thus SPARQL does not provide a normative specifiaction of the required output.

an example:

```
DESCRIBE <http://example.org/>
```

a WHERE clause can be optionally used to select a collection of unknown resources to be described —— which further properties of the selected objects are delivered then is left to the concrete implementation.

```
PREFIX ex: <http://example.org/>
DESCRIBE <http://www.example.org/Alice> ?person
WHERE { ?person  ex:email   _a . }
```

This query requests a description for all URIs that match the query pattern for ?person, as well as for the fixed URI http://www.example.org/Alice. `DESCRIBE *` can also be used to mean all the variables listed in WHERE clause should be described.

Modifiers

modifiers are used to change the order of the result or pick only a part of the result. modifiers include(note that the following order is also the order of execution)

- Order modifier: put the solutions in order; works only for SELECT format
- Projection modifier: choose certain variables (in SELECT clause)
- Distinct modifier: ensure solutions in the sequence are unique (SELECT DISTINCT)
- Reduced modifier: permit elimination of some non-unique solutions(SELECT REDUCED)
- Offset modifier: control where the solutions start from in the overall sequence of solutions
- Limit modifier: restrict the number of solutions

Order
an example:

```
SELECT ?book, ?price
WHERE { ?book  <http://example.org/price>  ?price . }
ORDER BY ASC(?price)
```

for descending order, we can use DESC(?price). The default order is ASC().

To still be able to return a well- defined result in cases which elements of different types are compared, SPARQL specifies an order among them. Whenever two elements of different kinds are compared, the following order is applied (smallest elements first):
1. no value (the variable by which results are ordered is not bound)
2. blank nodes
3. URIs
4. RDF literals

In general, SPARQL does not specify how literals of different datatypes are to be compared. Only for the case where a literal of type xsd:string is compared to an untyped literal that has the same lexical value, the untyped literal is defined to be smaller. The order among two unbound variables, two blank nodes, or two literals of unknown datatype is also not defined, and can be different in individual implementations, which might possibly support additional datatypes for which SPARQL does not prescribe any order.

The result can have **multiple** ORDER BYs (for different variables). In this case, only if all of the previous variables have the same value, the current variable will be used to sorting.
For example, in `ORDER BY DESC(?price) ASC(?title)` which realizes a descending order by price and an (ascending) alphabetic order by title for items with the same price.

Besides, note that CONSTRUCT is implemented via SELECT, so ORDER BY can also be used in CONSTRUCT query.

LIMIT and OFFSET
These 2 key words allow us to select a result segment containing at most as many results as specified by LIMIT, and starting with the result at the position given in OFFSET.
e.g.

```
SELECT*WHERE{?s ?p ?o.} ORDER BY?s LIMIT 5 OFFSET 25
```

This query shows the five triples starting with triple number 25, ac- cording to the order of subject elements.

DISTINCT and REDUCED
DISTINCT eliminate duplicate solutions, while REDUCED may have duplicate solutions, but the times of duplicity is guaranteed to be no larger than that in the result set without DISTINCT or REDUCED.