SciSPARQL User's Manual

Andrej Andrejev, 2012-06-07

Uppsala DataBase Laboratory
Department of Information Technology
Uppsala University
Sweden

Summary

Acknowledgements

1. Getting started

SciSPARQL is implemented with SSDM, technically - an extension of Amos II system. It is being distributed together with Amos II executables, headers and documentation.

The following files belong to SSDM extension proper:

```
bin/ssdm.dll
bin/ssdm.dmp
bin/ssdm.cmd
ssdm/*.*
```

To run SSDM, use `bin/ssdm.cmd` batch file. Directory `bin/` should be current or listed in system PATH.

The SciSPARQL toploop

When started, the system enters an Amos II top loop where it reads SciSPARQL statements, (including queries, function calls and system directives), executes them, and prints their results. The prompter in the SciSPARQL top loop is:

```
SciSPARQL n>
```

where n is a generation number. The generation number is increased every time a SciSPARQL statement is executed.

Typically you start by loading data. `LOAD` directive allows loading local or remote Turtle or NTriples files containing RDF datasets. Unqualified filename is processed relative to the current directory:

```
LOAD("talk.ttl");
```

The files are loaded into the 'default graph'. Multiple files can be loaded and any RDF blank nodes used in the different datasets become renamed to keep them lexically distinct. To empty the default graph before loading use `LOAD` with `true` as a second argument:

```
LOAD("talk.ttl",true);
```

Current dataset (i.e. default graph) can be written to a local NTriples file using `DUMP` directive:

```
DUMP("current.nt");
```

You can also load and execute SciSPARQL scripts, typically containing function definitions. `SOURCE` directive does exactly that:

```
SOURCE("talk.sparql");
```

At any point you can switch to Lisp interpreter using `LISP` directive, and return to SciSPARQL toploop by evaluating `language-sparql` symbol:

```
SPARQL 1> lisp;
Lisp 1> language-sparql
SPARQL 1>
```

To exit the toploop, use `EXIT` directctive.

```
SPARQL 1> exit;
```

2. SPARQL language basics

SciSPARQL i a superset of SPARQL 1.1 query language, which is defined by W3C Recommendations:

http://www.w3.org/TR/sparql11-query/

SciSPARQL complies with lexical specifications for the basic terms like URIs, Literals, Variables and Bank Nodes. The following chapters of the document can be used as a documentation for SciSPARQL language:

4-8, 10.1, 11, 15.3.1. 16.1

Chapters 17-19 give a more formal definition of the language syntax and semantics.

2.1. Lexical definitions

A ***token*** in SciSPARQL is either

- a syntactic **delimiter**, one of `( ) [ ] { } , ; . : ^^`

- an **operator**, one of `+ - * / = != < <= > >= || &&`

- a **keyword**, followed by a whitespace or a delimiter.

- a **numeric literal**, like
`2 3.14 -4.3e10`
folowed by a whitespace, a delimiter, or an operator.

- a **date & time literal**, like

- a **string literal**, enclosed in single or double quotes, like
```
"one" '1'
```
Quotes of the same type appearing inside the string, and the backslash character should be escaped with backslash '\':
```
"This string contains double quote \" and backslash \\."
```
No other escapes are supported in SciSPARQL strings, instead, tab and newline characters can be included directly into a string:
```
'          This string
is a paragraph.'
```

- a **language and locale tag**, starting with '@', like
```
@'En-Us'
```
Technically, any string is valid as a tag in SciSPARQL, and it is stored as part of a string literal that it syntactically follows.

- a Uniform Resource Identifier, **URI**, enclosed in angular brackets < >, like
```
<http://example.org>
```
The first character following the opening bracket should be a letter. Angular bracket, percent '%', space, tab and newline characters inside URI should be %-escaped, i.e. replaced with '%' followed by hexadecimal ASCII code of that character, for example:
```
<http://example.org/do.bml?param=%3Cempty%3E>
```

- an **abbreviated URI**, consisting of a prefix (see below) followed by URI local part like
```
rdf:type
```
Any non-alphanumeric characters in the URI local part should be %-escaped, as in URI. The only exceptions are '-' and '_' characters anywhere in the local part, and dot '.' if not on the last position.

- A **function call**, consisting of a URI, an abbreviated URI, or an identifier (see below), directly followed by an opening parenthesis '('.

- A **variable** (see below), followed by a whitespace, a delimiter, or an operator.

- A **labelled blank node** (see below), followed by a whitespace, a delimiter, or an operator.

An ***identifier*** in SciSPARQL is a sequence of letters, digits and numbers, where the first character is a letter. Identifiers are used to form variables, blank node labels, prefixes and function names.

Variables in W3C SPARQL and SciSPARQL start with variable marker '?' or '$' followed by an identifier. The variable marker is not a part of its name, so `?abc` and `$abc` are different occurrences of the same variable.

Labelled blank nodes in W3C SPARQL and SciSPARQL start with '_:' followed by identifier. There is also syntax for implicit blank nodes, like '[]', where unique labels are generated by the parser.

Prefixes in W3C SPARQL and SciSPARQL consist of an identifier followed by ':', or just ':', which corresponds to an empty identifier. Prefixes are declared in the beginning of a query or in separate SciSPARQL statemens, and are used to abbreviate URIs.

All W3C SPARQL and SciSPARQL identifiers and keywords are defined as case-insensitive, except for keyword a, which can be only used in lowercase.

2.2. Comments and whitespaces.

Comments start with '#', and follow to the end of line anywhere inside or outside a query.

In all cases end-of-line character is used as a whitespace delimiter, equivalent to space and tab, so the comment also becomes a whitespace. There is no other syntactic role associated with newline, so the query can be split into lines in any arbitrary way, as long as the tokens remain in one piece. This means that a comment can be inserted after any token.

2.3. Queries

A query is a syntactically valid sequence of tokens, optionally followed by ';' delimiter. When queries are processed in toploop or stored in `.sparql` file, ';' delimeter is required to separate the queries.

W3C SPARQL defines 4 types of queries - SELECT, CONSTRUCT, DESCRIBE and ASK, each starting with a corresponding keyword, and optionally preceded by PREFIX and/or BASE clauses.

Currently SciSPARQL supports only SELECT (Chapter 3) and CONSTRUCT (Chapter 4) queries, and PREFIX clause. It also adds DEFINE (Chapter 5), PREFIX, LISP, and EXIT directives, and any function call can be used as a query.

3. SELECT: querying for values

The most simple query will have to bind a value to a resulting variable:

```
SELECT (1 as ?res)
```

Will return

?res
1

In the most general form, a SciSPARQL select query has the following syntax:

(`PREFIX` *<prefix>*`:`*<URI>)**
`SELECT DISTINCT`*? <select-spec>+*
(`FROM` *<graph-id>)**
(`WHERE`*? <block>)?*
(`GROUP BY` *<var>+)?*
(`HAVING` *<expr>)?*

<select-spec> ::= <var> | (<expr> AS <var>)

<block> is an enclosed in curly braces `{ }` dot-separated conjunction of different kinds of conditions, namely:
- graph patterns, (3.1)

- UNIONs of alternative blocks (3.2)
- OPTIONAL blocks (3.3)
- FILTERs (3.5)
- BIND conditions (3.6)

Expressions are explained in 3.4, and <*select-spec*> is either a variable or an expression bound to a variable:

<*select-spec*> *::=* <*var*> | (<*expr*> `AS` <*var*>)

3.1. Graph patterns

A variable can also be bound in a graph pattern, like

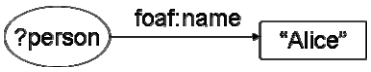
```
SELECT ?person
WHERE { ?person foaf:name "Alice" }
```

will bind the variable ?person to all subjects in the triples of the default dataset, where the predicate is a URI `foaf:name`, and the object is a string `"Alice"`, and return all those bindings.

Note: this query implies that the prefix `foaf:` is already defined in the session, which can be achieved by PREFIX directive:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>;
```

We can also use a graphical representation of a graph pattern, drawing predicates as arrows, subjects and objects as graph nodes. A node with value provided in the pattern will be shown as rectangle, a 'wildcard' node, will be depicted as oval, with a variable name if provided.



A graph pattern may be more complex and contain several variables, for example

```
SELECT ?friend_name
WHERE { ?person foaf:name "Alice" .
        ?person foaf:knows ?friend .
        ?friend foaf:name ?friend_name }
```

Here '`.`' is used as a conjunction, requiring that there should be a triple conforming to each pattern in order for the variable binding to be valid.

Note that not all the variables are interesting for us, only the bindings of `?friend_name` are returned.

This query can be simplified in several ways:

1) We can use '`;`' instead of '`.`' to indicate that the next pattern will have the same subject:

```
SELECT ?friend_name
WHERE { ?person foaf:name "Alice" ;
                foaf:knows ?friend .
        ?friend foaf:name ?friend_name }
```

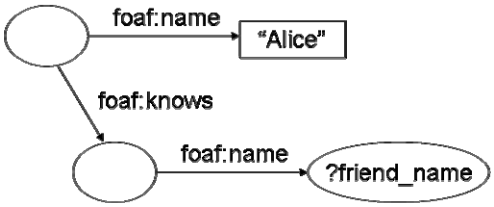
2) Since we are not interested in bindings for `?person`, we do not need to provide a variable name - instead we can tell the parser to generate a wildcard by using an unlabeled blank node '`[]`' in the query:

```
SELECT ?friend_name
WHERE { [] foaf:name "Alice" ;
           foaf:knows ?friend .
        ?friend foaf:name ?friend_name }
```

3) We can get rid of `?friend` variable as well, by substituting it with a blank-subject construct:

```
SELECT ?friend_name
WHERE { [] foaf:name "Alice" ;
           foaf:knows [ foaf:name ?friend_name ] }
```

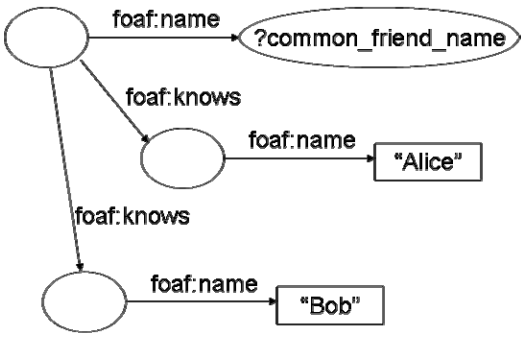
The `[ foaf:name ?friend_name ]` construct denotes a wildcard subject of a triple pattern with specified predicate and object, and can be used as a blank node anywhere in a graph pattern. So this graph pattern will contain two unnamed 'blank' nodes:



Another query will look for names of the people who know both Alice and Bob:

```
SELECT ?common_friend_name
WHERE { [] foaf:name ?common_friend_name ;
           foaf:knows [ foaf:name "Alice" ] ;
           foaf:knows [ foaf:name "Bob" ] }
```

or, graphically:



We can use '`,`' conjunction to indicate that the next triple pattern will have the same subject and predicate:

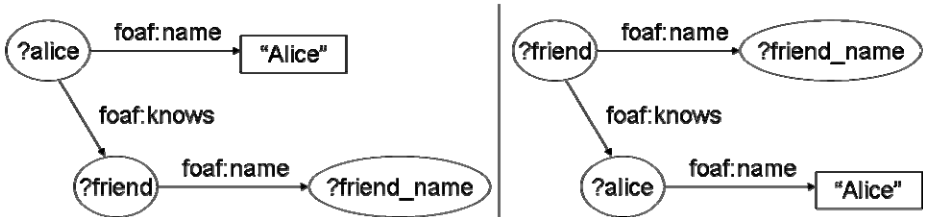
```
SELECT ?common_friend_name
WHERE { [] foaf:name ?common_friend_name ;
           foaf:knows [ foaf:name "Alice" ] ,
                      [ foaf:name "Bob" ] }
```

3.2. Matching alternatives and DISTINCT option

Consider that `foaf:knows` relationship is not restricted to be symmetric in the dataset, so we would like to trace it in either direction. The following query returns names of all the people who know Alice and all people whom Alice knows.

```
SELECT ?friend_name
WHERE { ?friend foaf:name ?friend_name .
        ?alice foaf:name "Alice" .
        { ?alice foaf:knows ?friend }
        UNION
        { ?friend foaf:knows ?alice } }
```

This query will effectively express two alternative graph patterns:



However, if `foaf:knows` relationship happens to be mutual in some case, same bindings will be generated twice for `?friend` and `?friend_name`. To avoid this, and return every person at most once, we should use DISTINCT option on the `?freind` variable in the SELECT clause:

```
SELECT DISTINCT ?friend ?friend_name
WHERE { ?friend foaf:name ?friend_name .
        ?alice foaf:name "Alice" .
        { ?alice foaf:knows ?friend }
        UNION
        { ?friend foaf:knows ?alice } }
```

Note that in this case we are required to include `?friend` in the result list, as the bindings for this variable are expected to be unique URIs (or dataset-unique blank nodes) identifying different persons. If we apply the DISTINCT only to the `?friend_name` variable, we will get a set of unique names, which might be shorter, as different people might happen to be namesakes.

There may be more than two UNION branches in the same conjunct, and a union branch can be any valid query block, including block containing nested UNIONs.

Additionally, different branches of the same union might provide bindings for different variables. For example, the following query might return a more informative result:

```
SELECT ?name_Alice_knows ?name_knows_Alice
WHERE { ?alice foaf:name "Alice" .
        { ?alice foaf:knows [ foaf:name ?name_Alice_knows ] }
        UNION
        { [ foaf:name ?name_knows_Alice ] foaf:knows ?alice } }
```

When applied to the dataset

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

_:alice foaf:name "Alice" ;
        foaf:knows _:bob ,
                   _:Cindy .
_:bob foaf:name "Bob" ;
      foaf:knows _:alice .
_:cindy foaf:name "cindy" .
_:erich foaf:name "erich" ;
        foaf:knows _:alice .
```

will return the following bindings for its SELECT variables

?name_Alice_knows	?name_knows_Alice
"Bob"	"Bob"
"Cindy"	
	"Erich"

The empty cells show that certain variables sometimes remain *unbound*. No further processing can be applied to them, and the result of any expression involving these variables will also be unbound, and any filter depending on such an expression will not be satisfied in that case. The only exception is the `bound()` function, which will return either `true` or `false`.

3.3. OPTIONAL graph patterns

Consider we would like to get the names of all the people Alice knows, and also their emails, if they are available in the dataset. Whenever there is no email information, the name of a person should still be returned:

```
SELECT ?friend_name ?friend_email
WHERE { [] foaf:name "Alice" ;
           foaf:knows ?friend .
        ?friend foaf:name ?friend_name .
        OPTIONAL { ?friend foaf:email ?friend_email } }
```

When applied to the dataset

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix countries: <http://example.org/Countries#> .

_:alice foaf:name "Alice" ;
        foaf:knows _:bob ,
                   _:cindy ,
                   _:dave .
_:bob foaf:name "Bob" ;
      foaf:email "bob@example.org" ;
      foaf:phone "+4912123456789" ;
      foaf:residesIn countries:Germany .
```

```
_:cindy foaf:name "Cindy" ;
       foaf:phone "+46701234567" .
_:dave foaf:name "Dave" ;
        foaf:residesIn countries:Australia .
```

will return the required and the optional bindings

?friend_name	?friend_email
"Bob"	"bob@example.org"
"Cindy"	
"Dave"	

Optional blocks can be nested. The nested block will provide new bindings for its variables only if the parent block succeeds. In the following query we are interested in the country information only if a phone number is returned:

```
SELECT ?friend_name ?friend_phone ?friend_country
WHERE { [] foaf:name "Alice" ;
           foaf:knows ?friend .
        ?friend foaf:name ?friend_name .
        OPTIONAL { ?friend foaf:phone ?friend_phone .
                   OPTIONAL { ?friend foaf:residesIn ?friend_country } } }
```

When applied to the same dataset, this query returns:

?friend_name	?friend_phone	?friend_country
"Bob"	"+4912123456789"	countries:Germany
"Cindy"	"+46701234567"	
"Dave"		

Note that the there is no country information returned for Dave, since there is no his phone of in the dataset.

There can also be several successive OPTIONAL blocks in a query block, and some of them might attempt to bind the same variable. In this case, the OPTIONAL block that appears earlier in the query gets the priority.

Consider the following query, where we are interested in the contact information Alice's friends - preferably an email, a phone number, or nothing but the name.

```
SELECT ?friend_name ?friend_contact
WHERE { [] foaf:name "Alice" ;
           foaf:knows ?friend .
        ?friend foaf:name ?friend_name .
        OPTIONAL { ?friend foaf:email ?friend_contact } .
        OPTIONAL { ?friend foaf:phone ?friend_contact } }
```

Applied to the same dataset, the query will return

?friend_name	?friend_contact
"Bob"	"bob@example.org"
"Cindy"	"+46701234567"
"Dave"	

If we swap the two OPTIONAL blocks in this query, a phone number would be returned for Bob as well.

3.4. Expressions

Logical and arithmetic expressions in SciSPARQL are formed by terms and operators.

Terms can be
- numeric, string or *typed* literals,
- URIs
- keywords `true` and `false` representing logical values
- variables
- function calls and typecasting
- array dereferences

3.4.1. Typed literals

Typed literals are syntactically formed by a string followed by ^^ delimiter and a complete or abbreviated URI indicating its type, for example

```
"1"^^xsd:integer
"10101110"^^<http://example.org/types/MyBitVector>
"2005-02-28T00:00:00Z"^^xsd:dateTime
```

The typed literals of type `xsd:integer`, `xsd:float`, `xsd:string`, `xsd:double`, `xsd:dateTime`, and `xsd:boolean`, found in SciSPARQL queries as well as in the imported Turtle/NTriples files are automatically converted to corresponding simple values. Other typed literals are stored together with their type URI and are considered comparable and equal when both the type URIs and value strings are the same.

3.4.2. Operators

Arithmetic operators are `+ - * /`, and are only applicable to numbers.

Comparison operators include `< <= > >=`, that are only applicable if both operands are numbers or both are strings, and `= !=`, that are applicable to the operands of comparable types. All numeric types are comparable with each other, typed literals are only comparable when they are completely equal, strings, dataTime, boolean and URI values are comparable with operands of the same type.

Boolean operators include || && !, and operate on *effective boolean values* that can be derived from operands of any type. Effective boolean values of non-boolean types are described in the following table

type	**effective boolean value**
`xsd:integer`	*false* if equal to 0, *true* otherwise
`xsd:float`	
`xsd:double`	
`xsd:string`	*false* if empty, *true* otherwise
`xsd:dateTime`	always *true*

URI	always *true*
other	*false* if string part is empty, *true* otherwise

3.4.3. Handling of unbound values

If a variable is *unbound* (e.g. due to its binding in an OPTIONAL graph pattern) the result of any expression involving that variable is also *unbound*. The exception are the boolean operators `||` and `&&`, that implement the following three-value logic, according to W3C SPARQL specifications:

A	**B**	**A || B**	**A && B**
true	*true*	*true*	*true*
true	*false*	*true*	*false*
false	*true*	*true*	*false*
false	*false*	*false*	*false*
true	*unbound*	*true*	*unbound*
unbound	*true*	*true*	*unbound*
false	*unbound*	*unbound*	*false*
unbound	*false*	*unbound*	*false*
unbound	*unbound*	*unbound*	*unbound*

Most built-in and all foreign functions are also not applicable to the *unbound* values, effectively returning *unbound* result. The only exception is `bound()` function, that will return *false* if the argument is *unbound*.

If the operator is not applicable to the values of its operands (like comparing a number to a string) or applying the operator produces arithmetic error (like dividing by zero), no error is raised, and the result is also *unbound*.

Unbound values can be seen as "empty cells" in SELECT query results. In CONSTRUCT queries resulting triples with *unbound* terms are filtered out.

FILTER and HAVING conditions do not distinguish between *unbound* and effective boolean *false* results of their expressions.

3.4.4. Function calls and typecasting