Apache Jena

- Home
- Download
- Learn
  - Tutorials
  - Overview
  - RDF core API tutorial
  - SPARQL tutorial
  - Manipulating SPARQL using ARQ
  - Using Jena with Eclipse
  - How-To's
  -
  - References
  - Overview
  - Javadoc
  - RDF API
  - RDF I/O
  - ARQ (SPARQL)
  - RDF Connection - SPARQL API
  - Elephas - tools for RDF on Hadoop
  - Text Search
  - TDB
  - SDB
  - SPARQL over JDBC
  - Fuseki
  - Permissions
  - Assembler
  - Ontology API
  - Inference API
  - Command-line tools
  - Extras
- Javadoc
  - Jena Core
  - ARQ
  - TDB
  - Fuseki
  - Elephas
  - Text Search
  - Spatial Search
  - Permissions
  - JDBC
  - All Javadoc
- Ask
- Get involved
  - Contribute
  - Report a bug
  -
  - Project
  - About Jena
  - Roadmap
  - Architecture
  - Project team

SPARQL Tutorial - A First SPARQL Query

In this section, we look at a simple first query and show how to execute it with Jena.

A "hello world" of queries

The file "q1.rq" contains the following query:

```
SELECT ?x
WHERE { ?x  <http://www.w3.org/2001/vcard-rdf/3.0#FN>  "John Smith" }
```

executing that query with the command line query application;

```
---------------------------------
| x                             |
=================================
| <http://somewhere/JohnSmith/> |
---------------------------------
```

This works by matching the triple pattern in the `WHERE` clause against the triples in the RDF graph. The predicate and object of the triple are fixed values so the pattern is going to match only triples with those values. The subject is a variable, and there are no other restrictions on the variable. The pattern matches any triples with these predicate and object values, and it matches with solutions for `x`.

The item enclosed in <> is a URI (actually, it's an IRI) and the item enclosed in "" is a plain literal. Just like Turtle, N3 or N-triples, typed literals are written with \^\^ and language tags can be added with @.

?x is a variable called x. The ? does not form part of the name which is why it does not appear in the table output.

There is one match. The query returns the match in the `x` query variable. The output shown was obtained by using one of ARQ's command line applications.

Executing the query

There are helper scripts in the Jena distribution `bat/` and `bin/` directories. You should check these scripts before use. They can be placed on the shell command path.

Windows setup

Execute:

```
bat\sparql.bat --data=doc\Tutorial\vc-db-1.rdf --query=doc\Tutorial\q1.rq
```

You can just put the `bat/` directory on your classpath or copy the programs out of it.

bash scripts for Linux/Cygwin/Unix

Execute:

```
bin/sparql --data=doc/Tutorial/vc-db-1.rdf --query=doc/Tutorial/q1.rq
```

Using the Java command line applications directly

(This is not necessary.)

You will need to set the classpath to include *all* the jar files in the Jena distribution `lib/` directory.

```
java -cp 'DIST/lib/*' arq.sparql ...
```

where `DIST` is the `apache-jena-VERSION` directory.

Next: basic patterns

SPARQL Tutorial - Basic Patterns

This section covers basic patterns and solutions, the main building blocks of SPARQL queries.

Solutions

Query solutions are a set of pairs of a variable name with a value. A `SELECT` query directly exposes the solutions (after order/limit/offset are applied) as the result set - other query forms use the solutions to make a graph. The solution is the way the pattern matched - which values the variables must take for a pattern to match.

The first query example had a single solution. Change the pattern to this second query: (q-bp1.rq):

```
SELECT ?x ?fname
WHERE {?x  <http://www.w3.org/2001/vcard-rdf/3.0#FN>  ?fname}
```

This has 4 solutions, one for each VCARD name property triples in the data source

```
----------------------------------------------------
| x                                | name          |
====================================================
| <http://somewhere/RebeccaSmith/> | "Becky Smith" |
| <http://somewhere/SarahJones/>   | "Sarah Jones" |
| <http://somewhere/JohnSmith/>    | "John Smith"  |
| <http://somewhere/MattJones/>    | "Matt Jones"  |
----------------------------------------------------
```

So far, with triple patterns and basic patterns, every variable will be defined in every solution. The solutions to a query can be thought of a table, but in the general case, it is a table where not every row will have a value for every column. All the solutions to a given SPARQL query don't have to have values for all the variables in every solution as we shall see later.

Basic Patterns

A basic pattern is a set of triple patterns. It matches when the triple patterns all match with the same value used each time the variable with the same name is used.

```
SELECT ?givenName
WHERE
  { ?y  <http://www.w3.org/2001/vcard-rdf/3.0#Family>  "Smith" .
    ?y  <http://www.w3.org/2001/vcard-rdf/3.0#Given>  ?givenName .
  }
```

This query (q-bp2.rq) involves two triple patterns, each triple ends in a '.' (but the dot after the last one can be omitted like it was in the one triple pattern example). The variable y has to be the same for each triple pattern match. The solutions are:

```
-------------
| givenName |
=============
| "John"    |
| "Rebecca" |
-------------
```

QNames

There is shorthand mechanism for writing long URIs using prefixes. The query above is more clearly written as the query (q-bp3.rq):

```
PREFIX vcard:      <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?givenName
WHERE
 { ?y vcard:Family "Smith" .
   ?y vcard:Given  ?givenName .
 }
```

This is a prefixing mechanism - the two parts of the URIs, from the prefix declaration and from the part after the ":" in the qname, are concatenated together. This is strictly not what an XML qname is but uses the RDF rule for turning a qname into a URI by concatenating the parts.

Blank Nodes

Change the query just a little to return y as well (q-bp4.rq) :

```
PREFIX vcard:      <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?y ?givenName
WHERE
 { ?y vcard:Family "Smith" .
   ?y vcard:Given  ?givenName .
 }
```

and the blank nodes appear

```
--------------------
| y    | givenName |
====================
| _:b0 | "John"    |
| _:b1 | "Rebecca" |
--------------------
```

as odd looking qnames starting _:. This isn't the internal label for the blank node - it is ARQ printing them out that assigned the _:b0, _:b1 to show when two blank nodes are the same. Here they are different. It does not reveal the internal label used for the blank node although that is available when using the Java API.

Next: Filters

Apache Jena

- Home
- Download
- Learn
  - Tutorials
  - Overview
  - RDF core API tutorial
  - SPARQL tutorial
  - Manipulating SPARQL using ARQ
  - Using Jena with Eclipse
  - How-To's
  -
  - References
  - Overview
  - Javadoc
  - RDF API
  - RDF I/O
  - ARQ (SPARQL)
  - RDF Connection - SPARQL API
  - Elephas - tools for RDF on Hadoop
  - Text Search
  - TDB
  - SDB
  - SPARQL over JDBC
  - Fuseki
  - Permissions
  - Assembler
  - Ontology API
  - Inference API
  - Command-line tools
  - Extras
- Javadoc
  - Jena Core
  - ARQ
  - TDB
  - Fuseki
  - Elephas
  - Text Search
  - Spatial Search
  - Permissions
  - JDBC
  - All Javadoc
- Ask
- Get involved
  - Contribute
  - Report a bug
  -
  - Project
  - About Jena
  - Roadmap
  - Architecture
  - Project team

SPARQL Tutorial - Filters

Graph matching allows patterns in the graph to be found. This section describes how the values in a solution can be restricted. There are many comparisons available - we just cover two cases here.

String Matching

SPARQL provides an operation to test strings, based on regular expressions. This includes the ability to ask SQL "LIKE" style tests, although the syntax of the regular expression is different from SQL.

The syntax is:

```
FILTER regex(?x, "pattern" [, "flags"])
```

The flags argument is optional. The flag "i" means a case-insensitive pattern match is done.

The example query (q-f1.rq) finds given names with an "r" or "R" in them.

```
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?g
WHERE
{ ?y vcard:Given ?g .
  FILTER regex(?g, "r", "i") }
```

with the results:

```
-------------
| g         |
=============
| "Rebecca" |
| "Sarah"   |
-------------
```

The regular expression language is the same as the XQuery regular expression language which is codified version of that found in Perl.

Testing Values

There are times when the application wants to filter on the value of a variable. In the data file vc-db-2.rdf, we have added an extra field for age. Age is not defined by the vCard schema so we have created a new property for the purpose of this tutorial. RDF allows such mixing of different definitions of information because URIs are unique. Note also that the `info:age` property value is typed.

In this extract of the data, we show the typed value. It can also be written plain 23.

```
<http://somewhere/RebeccaSmith/>
    info:age "23"^^xsd:integer ;
    vCard:FN "Becky Smith" ;
    vCard:N [ vCard:Family "Smith" ;
              vCard:Given  "Rebecca" ] .
```

So, a query (q-f2.rq) to find the names of people who are older than 24 is:

```
PREFIX info: <http://somewhere/peopleInfo#>

SELECT ?resource
WHERE
  {
    ?resource info:age ?age .
    FILTER (?age >= 24)
  }
```

The arithmetic expression must be in parentheses (round brackets). The only solution is:

```
---------------------------------
| resource                      |
=================================
| <http://somewhere/JohnSmith/> |
---------------------------------
```

Just one match, resulting in the resource URI for John Smith. Turning this round to ask for those less than 24 also yields one match for Rebecca Smith. Nothing about the Jones's.

The database contains no age information about the Jones: there are no info:age properties on these vCards so the variable `age` did not get a value and so was not tested by the filter.

Next: Optionals

Apache Jena

- Home
- Download
- Learn
  - Tutorials
  - Overview
  - RDF core API tutorial
  - SPARQL tutorial
  - Manipulating SPARQL using ARQ
  - Using Jena with Eclipse
  - How-To's
  -
  - References
  - Overview
  - Javadoc
  - RDF API
  - RDF I/O
  - ARQ (SPARQL)
  - RDF Connection - SPARQL API
  - Elephas - tools for RDF on Hadoop
  - Text Search
  - TDB
  - SDB
  - SPARQL over JDBC
  - Fuseki
  - Permissions
  - Assembler
  - Ontology API
  - Inference API
  - Command-line tools
  - Extras
- Javadoc
  - Jena Core
  - ARQ
  - TDB
  - Fuseki
  - Elephas
  - Text Search
  - Spatial Search
  - Permissions
  - JDBC
  - All Javadoc
- Ask
- Get involved
  - Contribute
  - Report a bug
  -
  - Project
  - About Jena
  - Roadmap
  - Architecture
  - Project team

  - Related projects
  -
  - ASF
  - Apache Software Foundation
  - License
  - Thanks
  - Become a Sponsor
  - Security
- Improve this Page

1. TUTORIALS
2. SPARQL OPTIONALS

SPARQL Tutorial - Optional Information

RDF is semi-structured data so SPARQL has a the ability to query for data but not to fail query when that data does not exist. The query is using an optional part to extend the information found in a query solution but to return the non-optional information anyway.

OPTIONALs

This query (q-opt1.rq) gets the name of a person and also their age if that piece of information is available.

```
PREFIX info:    <http://somewhere/peopleInfo#>
PREFIX vcard:   <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name ?age
WHERE
{
    ?person vcard:FN  ?name .
    OPTIONAL { ?person info:age ?age }
}
```

Two of the four people in the data (vc-db-2.rdf)have age properties so two of the query solutions have that information. However, because the triple pattern for the age is optional, there is a pattern solution for the people who don't have age information.

```
------------------------
| name          | age |
=======================
| "Becky Smith" | 23  |
| "Sarah Jones" |     |
| "John Smith"  | 25  |
| "Matt Jones"  |     |
-----------------------
```

If the optional clause had not been there, no age information would have been retrieved. If the triple pattern had been included but not optional then we would have the query (q-opt2.rq):

```
PREFIX info:   <http://somewhere/peopleInfo#>
PREFIX vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name ?age
WHERE
{
    ?person vcard:FN  ?name .
    ?person info:age ?age .
}
```

with only two solutions:

```
-----------------------
| name           | age |
=======================
| "Becky Smith" | 23  |
| "John Smith"  | 25  |
-----------------------
```

because the `info:age` property must now be present in a solution.

OPTIONALs with FILTERs

`OPTIONAL` is a binary operator that combines two graph patterns. The optional pattern is any group pattern and may involve any SPARQL pattern types. If the group matches, the solution is extended, if not, the original solution is given (q-opt3.rq).

```
PREFIX info:        <http://somewhere/peopleInfo#>
PREFIX vcard:      <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name ?age
WHERE
{
    ?person vcard:FN  ?name .
    OPTIONAL { ?person info:age ?age . FILTER ( ?age > 24 ) }
}
```

So, if we filter for ages greater than 24 in the optional part, we will still get 4 solutions (from the `vcard:FN` pattern) but only get ages if they pass the test.

```
-----------------------
| name           | age |
=======================
| "Becky Smith" |     |
| "Sarah Jones" |     |
| "John Smith"  | 25  |
| "Matt Jones"  |     |
-----------------------
```

No age included for "Becky Smith" because it is less than 24.

If the filter condition is moved out of the optional part, then it can influence the number of solutions but it may be necessary to make the filter more complicated to allow for variable `age` being unbound (q-opt4.rq).

```
PREFIX info:        <http://somewhere/peopleInfo#>
PREFIX vcard:      <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name ?age
WHERE
{
    ?person vcard:FN  ?name .
    OPTIONAL { ?person info:age ?age . }
    FILTER ( !bound(?age) || ?age > 24 )
}
```

If a solution has an `age` variable, then it must be greater than 24. It can also be unbound. There are now three solutions:

```
-----------------------
| name          | age |
=======================
| "Sarah Jones" |     |
| "John Smith"  | 25  |
| "Matt Jones"  |     |
-----------------------
```

Evaluating an expression which has an unbound variables where a bound one was expected causes an evaluation exception and the whole expression fails.

OPTIONALs and Order Dependent Queries

One thing to be careful of is using the same variable in two or more optional clauses (and not in some basic pattern as well):

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name
WHERE
{
  ?x a foaf:Person .
  OPTIONAL { ?x foaf:name ?name }
  OPTIONAL { ?x vCard:FN  ?name }
}
```

If the first optional binds `?name` and `?x` to some values, the second `OPTIONAL` is an attempt to match the ground triples (`?x` and `<kbd>?name</kbd>` have values). If the first optional did not match the optional part, then the second one is an attempt to match its triple with two variables.

Next: union queries

Apache Jena

- Home
- Download
- Learn
  - Tutorials
  - Overview
  - RDF core API tutorial
  - SPARQL tutorial
  - Manipulating SPARQL using ARQ
  - Using Jena with Eclipse
  - How-To's
  -
  - References
  - Overview
  - Javadoc
  - RDF API
  - RDF I/O
  - ARQ (SPARQL)
  - RDF Connection - SPARQL API
  - Elephas - tools for RDF on Hadoop
  - Text Search
  - TDB
  - SDB
  - SPARQL over JDBC
  - Fuseki
  - Permissions
  - Assembler
  - Ontology API
  - Inference API
  - Command-line tools
  - Extras
- Javadoc
  - Jena Core
  - ARQ
  - TDB
  - Fuseki
  - Elephas
  - Text Search
  - Spatial Search
  - Permissions
  - JDBC
  - All Javadoc
- Ask
- Get involved
  - Contribute
  - Report a bug
  -
  - Project
  - About Jena
  - Roadmap
  - Architecture
  - Project team

  - Related projects
  -
  - ASF
  - Apache Software Foundation
  - License
  - Thanks
  - Become a Sponsor
  - Security
- Improve this Page

1. TUTORIALS
2. SPARQL UNION

SPARQL Tutorial - Alternatives in a Pattern

Another way of dealing with the semi-structured data is to query for one of a number of possibilities. This section covers `UNION` patterns, where one of a number of possibilities is tried.

UNION - two ways to the same data

Both the vCard vocabulary and the FOAF vocabulary have properties for people's names. In vCard, it is vCard:FN, the "formatted name", and in FOAF, it is foaf:name. In this section, we will look at a small set of data where the names of people can be given by either the FOAF or the vCard vocabulary.

Suppose we have an RDF graph that contains name information using both the vCard and FOAF vocabularies.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .

_:a foaf:name   "Matt Jones" .

_:b foaf:name   "Sarah Jones" .

_:c vcard:FN    "Becky Smith" .

_:d vcard:FN    "John Smith" .
```

A query to access the name information, when it can be in either form, could be (q-union1.rq):

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name
WHERE
{
   { [] foaf:name ?name } UNION { [] vCard:FN ?name }
}
```

This returns the results:

```
-----------------
| name          |
=================
| "Matt Jones"  |
| "Sarah Jones" |
| "Becky Smith" |
| "John Smith"  |
-----------------
```

It didn't matter which form of expression was used for the name, the ?name variable is set. This can be achieved using a `FILTER` as this query (q-union-1alt.rq) shows:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name
WHERE
{
  [] ?p ?name
  FILTER ( ?p = foaf:name || ?p = vCard:FN )
}
```

testing whether the property is one URI or another. The solutions may not come out in the same order. The first form is more likely to be faster, depending on the data and the storage used, because the second form may have to get all the triples from the graph to match the triple pattern with unbound variables (or blank nodes) in each slot, then test each `?p` to see if it matches one of the values. It will depend on the sophistication of the query optimizer as to whether it spots that it can perform the query more efficiently and is able to pass the constraint down as well as to the storage layer.

UNION - remembering where the data was found.

The example above used the same variable in each branch. If different variables are used, the application can discover which sub-pattern caused the match (q-union2.rq):

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name1 ?name2
WHERE
{
   { [] foaf:name ?name1 } UNION { [] vCard:FN ?name2 }
}
```

```
---------------------------------
| name1         | name2         |
=================================
| "Matt Jones"  |               |
| "Sarah Jones" |               |
|               | "Becky Smith" |
|               | "John Smith"  |
---------------------------------
```

This second query has retained information of where the name of the person came from by assigning the name to different variables.

OPTIONAL and UNION

In practice, `OPTIONAL` is more common than `UNION` but they both have their uses. `OPTIONAL` are useful for augmenting the solutions found, `UNION` is useful for concatenating the solutions from two possibilities. They don't necessary return the information in the same way:

Query(q-union3.rq):

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?name1 ?name2
WHERE
{
  ?x a foaf:Person
  OPTIONAL { ?x  foaf:name  ?name1 }
  OPTIONAL { ?x  vCard:FN   ?name2 }
}

---------------------------------
| name1          | name2          |
=================================
| "Matt Jones"  |                |
| "Sarah Jones" |                |
|               | "Becky Smith" |
|               | "John Smith"  |
---------------------------------
```

but beware of using `?name` in each `OPTIONAL` because that is an order-dependent query.

[Next: Named Graphs]

- Related projects
-
- ASF
- Apache Software Foundation
- License
- Thanks
- Become a Sponsor
- Security

- Improve this Page

1. TUTORIALS
2. SPARQL DATASETS

SPARQL Tutorial - Datasets

This section covers RDF Datasets - an RDF Dataset is the unit that is queried by a SPARQL query. It consists of a default graph, and a number of named graphs.

Querying datasets

The graph matching operation (basic patterns, `OPTIONAL`s, and `UNION`s) work on one RDF graph. This starts out being the default graph of the dataset but it can be changed by the `GRAPH` keyword.

```
GRAPH uri { ... pattern ... }
```

```
GRAPH var { ... pattern ... }
```

If a URI is given, the pattern will be matched against the graph in the dataset with that name - if there isn't one, the `GRAPH` clause fails to match at all.

If a variable is given, all the named graphs (not the default graph) are tried. The variable may be used elsewhere so that if, during execution, its value is already known for a solution, only the specific named graph is tried.

Example Data

An RDF dataset can take a variety of forms. Two common setups are to have the default graph being the union (the RDF merge) of all the named graphs or to have the default graph be an inventory of the named graphs (where they came from, when they were read etc). There are no limitations - one graph can be included twice under different names, or some graphs may share triples with others.

In the examples below we will use the following dataset that might occur for an RDF aggregator of book details:

Default graph (ds-dft.ttl):

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<ds-ng-1.ttl> dc:date "2005-07-14T03:18:56+0100"^^xsd:dateTime .
<ds-ng-2.ttl> dc:date "2005-09-22T05:53:05+0100"^^xsd:dateTime .
```

Named graph (ds-ng-1.ttl):

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

```
[] dc:title "Harry Potter and the Philospher's Stone" .
[] dc:title "Harry Potter and the Chamber of Secrets" .
```

Named graph (ds-ng-2.ttl):

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

[] dc:title "Harry Potter and the Sorcerer's Stone" .
[] dc:title "Harry Potter and the Chamber of Secrets" .
```

That is, we have two small graphs describing some books, and we have a default graph which records when these graphs were last read.

Queries can be run with the command line application (this would be all one line):

```
java -cp ... arq.sparql
    --graph ds-dft.ttl --namedgraph ds-ng-1.ttl --namedgraph ds-ng-2.ttl
    --query query file
```

Datasets don't have to be created just for the lifetime of the query. They can be created and stored in a database, as would be more usual for an aggregator application.

Accessing the Dataset

The first example just accesses the default graph (q-ds-1.rq):

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <.>

SELECT *
{ ?s ?p ?o }
```

(The "`PREFIX : <.>`" just helps format the output)

```
----------------------------------------------------------------------
| s             | p       | o                                        |
======================================================================
| :ds-ng-2.ttl | dc:date | "2005-09-22T05:53:05+01:00"^^xsd:dateTime |
| :ds-ng-1.ttl | dc:date | "2005-07-14T03:18:56+01:00"^^xsd:dateTime |
----------------------------------------------------------------------
```

This is the default graph only - nothing from the named graphs because they aren't queried unless explicitly indicated via `GRAPH`.

We can query for all triples by querying the default graph and the named graphs (q-ds-2.rq):

```
PREFIX  xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX  dc:     <http://purl.org/dc/elements/1.1/>
PREFIX  :       <.>

SELECT *
{
    { ?s ?p ?o } UNION { GRAPH ?g { ?s ?p ?o } }
}
```

giving:

```
---------------------------------------------------------------------------------------
| s             | p        | o                                            | g            |
=======================================================================================
| :ds-ng-2.ttl | dc:date  | "2005-09-22T05:53:05+01:00"^^xsd:dateTime  |              |
| :ds-ng-1.ttl | dc:date  | "2005-07-14T03:18:56+01:00"^^xsd:dateTime  |              |
| _:b0         | dc:title | "Harry Potter and the Sorcerer's Stone"    | :ds-ng-2.ttl |
| _:b1         | dc:title | "Harry Potter and the Chamber of Secrets"  | :ds-ng-2.ttl |
| _:b2         | dc:title | "Harry Potter and the Chamber of Secrets"  | :ds-ng-1.ttl |
| _:b3         | dc:title | "Harry Potter and the Philospher's Stone"  | :ds-ng-1.ttl |
-------------------------------------------------------------------------------------
```

Querying a specific graph

If the application knows the name graph, it can directly ask a query such as finding all the titles in a given graph (q-ds-3.rq):

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

SELECT ?title
{
  GRAPH :ds-ng-2.ttl
    { ?b dc:title ?title }
}
```

Results:

```
---------------------------------------------
| title                                     |
=============================================
| "Harry Potter and the Sorcerer's Stone"   |
| "Harry Potter and the Chamber of Secrets" |
---------------------------------------------
```

Querying to find data from graphs that match a pattern

The name of the graphs to be queried can be determined with the query itself. The same process for variables applies whether they are part of a graph pattern or the `GRAPH` form. The query below (q-ds-4.rq) sets a condition on the variable used to select named graphs, based on information in the default graph.

```
PREFIX  xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX  dc:     <http://purl.org/dc/elements/1.1/>
PREFIX  :       <.>

SELECT ?date ?title
{
  ?g dc:date ?date . FILTER (?date > "2005-08-01T00:00:00Z"^^xsd:dateTime )
  GRAPH ?g
      { ?b dc:title ?title }
}
```

The results of executing this query on the example dataset are the titles in one of the graphs, the one with the date later than 1 August 2005.

```
-----------------------------------------------------------------------------------------
| date                                      | title                                     |
=========================================================================================
| "2005-09-22T05:53:05+01:00"^^xsd:dateTime | "Harry Potter and the Sorcerer's Stone"   |
| "2005-09-22T05:53:05+01:00"^^xsd:dateTime | "Harry Potter and the Chamber of Secrets" |
-----------------------------------------------------------------------------------------
```

Describing RDF Datasets - `FROM` and `FROM NAMED`

A query execution can be given the dataset when the execution object is built or it can be described in the query itself. When the details are on the command line, a temporary dataset is created but an application can create datasets and then use them in many queries.

When described in the query, `FROM <url>` is used to identify the contents to be in the default graph. There can be more than one `FROM` clause and the default graph is result of reading each file into the default graph. It is the RDF merge of the individual graphs.

Don't be confused by the fact the default graph is described by one or more URLs in `FROM` clauses. This is where the data is read from, not the name of the graph. As several FROM clauses can be given, the data can be read in from several places but none of them become the graph name.

`FROM NAMED <url>` is used to identify a named graph. The graph is given the name *url* and the data is read from that location. Multiple `FROM NAMED` clauses cause multiple graphs to be added to the dataset.

Note that graphs are loaded with the Jena FileManager which includes the ability to provide alternative locations for files. For example, the query may have `FROM NAMED <http://example/data>`, and the data actually be read from `file:local.rdf`. The name of the graph will be <http://example/data> as in the query.

For example, the query to find all the triples in both default graph and named graphs could be written as (q-ds-5.rq):

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dc:  <http://purl.org/dc/elements/1.1/>
PREFIX :    <.>

SELECT *
FROM       <ds-dft.ttl>
FROM NAMED <ds-ng-1.ttl>
FROM NAMED <ds-ng-2.ttl>
{
   { ?s ?p ?o } UNION { GRAPH ?g { ?s ?p ?o } }
}
```

Next: results

Apache Jena

- Home
- Download
- Learn
    - Tutorials
    - Overview
    - RDF core API tutorial
    - SPARQL tutorial
    - Manipulating SPARQL using ARQ
    - Using Jena with Eclipse
    - How-To's
    -
    - References
    - Overview
    - Javadoc
    - RDF API
    - RDF I/O
    - ARQ (SPARQL)
    - RDF Connection - SPARQL API
    - Elephas - tools for RDF on Hadoop
    - Text Search
    - TDB
    - SDB
    - SPARQL over JDBC
    - Fuseki
    - Permissions
    - Assembler
    - Ontology API
    - Inference API
    - Command-line tools
    - Extras
- Javadoc
    - Jena Core
    - ARQ
    - TDB
    - Fuseki
    - Elephas
    - Text Search
    - Spatial Search
    - Permissions
    - JDBC
    - All Javadoc
- Ask
- Get involved
    - Contribute
    - Report a bug
    -
    - Project
    - About Jena
    - Roadmap
    - Architecture
    - Project team

Producing Result Sets

SPARQL has four result forms:

- SELECT – Return a table of results.
- CONSTRUCT – Return an RDF graph, based on a template in the query.
- DESCRIBE – Return an RDF graph, based on what the query processor is configured to return.
- ASK – Ask a boolean query.

The SELECT form directly returns a table of solutions as a result set, while DESCRIBE and CONSTRUCT use the outcome of matching to build RDF graphs.

Solution Modifiers

Pattern matching produces a set of solutions. This set can be modified in various ways:

- Projection - keep only selected variables
- OFFSET/LIMIT - chop the number solutions (best used with ORDER BY)
- ORDER BY - sorted results
- DISTINCT - yield only one row for one combination of variables and values.

The solution modifiers OFFSET/LIMIT and ORDER BY always apply to all result forms.

OFFSET and LIMIT

A set of solutions can be abbreviated by specifying the offset (the start index) and the limit (the number of solutions) to be returned. Using LIMIT alone can be useful to ensure not too many solutions are returned, to restrict the effect of some unexpected situation. LIMIT and OFFSET can be used in conjunction with sorting to take a defined slice through the solutions found.

ORDER BY

SPARQL solutions are sorted by expression, including custom functions.

```
ORDER BY ?x ?y
```

```
ORDER BY DESC(?x)
```

```
ORDER BY x:func(?x)  # Custom sorting condition
```

DISTINCT

The SELECT result form can take the DISTINCT modifier which ensures that no two solutions returned are the same - this takes place after projection to the requested variables.

SELECT

The `SELECT` result form is a projection, with DISTINCT applied, of the solution set. `SELECT` identifies which named variables are in the result set. This may be "`*`" meaning "all named variables" (blank nodes in the query act like variables for matching but are never returned).

CONSTRUCT

CONSTRUCT builds an RDF based on a graph template. The graph template can have variables which are bound by a WHERE clause. The effect is to calculate the graph fragment, given the template, for each solution from the WHERE clause, after taking into account any solution modifiers. The graph fragments, one per solution, are merged into a single RDF graph which is the result.

Any blank nodes explicitly mentioned in the graph template are created afresh for each time the template is used for a solution.

DESCRIBE

The CONSTRUCT form, takes an application template for the graph results. The DESCRIBE form also creates a graph but the form of that graph is provided the query processor, not the application. For each URI found, or explicitly mentioned in the DESCRIBE clause, the query processor should provide a useful fragment of RDF, such as all the known details of a book. ARQ allows domain-specific description handlers to be written.

ASK

The ASK result form returns a boolean, true of the pattern matched otherwise false.

Return to index