Search RDF data with SPARQL

SPARQL and the Jena Toolkit open up the semantic Web

Philip McCarthy May 10, 2005

As more data is being stored in RDF formats like RSS, a need has arisen for a simple way to locate specific information. SPARQL, a powerful new query language fills that space, making it easy to find the data you need in the RDF haystack. Take a tour of SPARQL's features and learn how to use SPARQL queries from your own Java™ applications with the Jena Semantic Web Toolkit.

The *Resource Description Framework,* or RDF, enables data to be decentralized and distributed. RDF models can be merged together easily, and serialized RDF can be simply exchanged over HTTP. Applications can be loosely coupled to multiple RDF data sources over the Web. At PlanetRDF.com, for example, we syndicate weblogs from authors who provide their content as RDF in a RSS 1.0 feed. The URLs of the authors' feeds are themselves held in an RDF graph, called *bloggers.rdf.*

But how can you find and manipulate the data you need within RDF graphs? The SPARQL Protocol And RDF Query Language (SPARQL) is currently under discussion as a W3C Working Draft. SPARQL builds on previous RDF query languages such as rdfDB, RDQL, and SeRQL, and has several valuable new features of its own. In this article, we'll use the three types of RDF graph that drive PlanetRDF -- FOAF graphs describing contributors, their RSS 1.0 feeds, and the bloggers graph -- to demonstrate some of the interesting things SPARQL queries can do with your data. Implementations of SPARQL exist for a variety of platforms and languages; this article will focus on the Jena Semantic Web Toolkit for the Java platform.

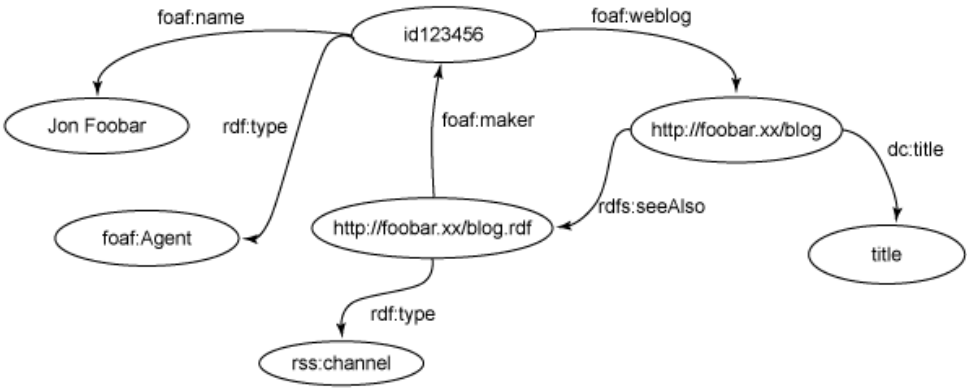
This article assumes that you have a working knowledge of RDF, and are familiar with RDF vocabularies such as Dublin Core, FOAF, and RSS 1.0. It also assumes you have some experience with the Jena Semantic Web Toolkit. To get up to speed on all of these technologies, check out the links in the Related topics section below.

Anatomy of a simple SPARQL query

Let's start by looking at PlanetRDF's bloggers.rdf model. It is fairly straightforward, using the FOAF and Dublin Core vocabularies to provide a name, a blog title and URL, and an RSS feed

description for each blog contributor. Figure 1 shows the basic graph structure for a single contributor. The full model simply repeats this structure for each blog that we aggregate.

Figure 1. Basic graph structure for a single contributor in bloggers.rdf



Now let's look at a very simple SPARQL query over the bloggers model. The query, in English, says, "Find the URL of the blog by the person named Jon Foobar," and is shown in Listing 1:

Listing 1. SPARQL query to find the URL of a contributor's blog

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?url
FROM  <bloggers.rdf>
WHERE  {
    ?contributor foaf:name "Jon Foobar" .
    ?contributor foaf:weblog ?url .
}
```

The first line of the query simply defines a `PREFIX` for the FOAF namespace, so that you don't have to type it in full each time it is referenced. The `SELECT` clause specifies what the query should return -- in this case, a variable named `url`. SPARQL variables are prefixed with either `?` or `$` -- the two are interchangeable, but I'll stick to `?` in this article. `FROM` is an optional clause that provides the URI of the dataset to use. Here, it's just pointing to a local file, but it could also indicate the URL of a graph somewhere on the Web. Finally, the `WHERE` clause consists of a series of triple patterns, expressed using Turtle-based syntax. These triples together comprise what is known as a *graph pattern.*

The query attempts to match the triples of the graph pattern to the model. Each matching binding of the graph pattern's variables to the model's nodes becomes a *query solution,* and the values of the variables named in the `SELECT` clause become part of the query results.

In this example, the first triple in the `WHERE` clause's graph pattern matches a node with a `foaf:name` property of "Jon Foobar," and binds it to the variable named `contributor`. In the bloggers.rdf model, `contributor` will match the `foaf:Agent` blank-node at the top of Figure 1. The graph pattern's second triple matches the object of the `contributor`'s `foaf:weblog` property. This is bound to the `url` variable, forming a query solution.

Using SPARQL with Jena

SPARQL support in Jena is currently available via a module called *ARQ.* In addition to implementing SPARQL, ARQ's query engine can also parse queries expressed in RDQL or its own internal query language. ARQ is under active development, and is not yet part of the standard Jena distribution. However, it is available from either Jena's CVS repository or as a self-contained download.

It's simple to get ARQ up and running. Just grab the latest ARQ distribution (see the Related topics section below for a link), unpack it, and set the environment variable `ARQROOT` to point to the ARQ directory. You may also need to restore read and execute permissions to the contents of the ARQ bin directory. It's convenient to add the bin directory to your execution path, as it contains wrapper scripts to invoke ARQ from the command line. To make sure that you are all set, call `sparql` from the command line, and make sure you see its usage message. All these steps are illustrated in Listing 2, which assumes that you are working on a UNIX-like platform, or with Cygwin under Windows. (ARQ also ships with `.bat` scripts to use under Windows, but their usage will vary slightly from the example shown here.)

Listing 2. Setting up your environment to use Jena ARQ

```
$ export ARQROOT=~/ARQ-0.9.5
$ chmod +rx $ARQROOT/bin/*
$ export PATH=$PATH:$ARQROOT/bin
$ sparql
 Usage: [--data URL] [exprString | --query file]
```

Executing SPARQL queries from the command line

Now you're ready to run a SPARQL query (see Listing 3). We'll use the dataset and query from Listing 1. Because the query uses the `FROM` keyword to specify the graph to use, it is necessary only to provide the location of the query file to the `sparql` command. However, the query needs to be run from the directory containing the graph, because it is specified using a relative URL in the query's `FROM` clause.

Listing 3. Executing a simple query with the sparql command

```
$ sparql --query jon-url.rq

---------------------------
| url                     |
===========================
| <http://foobar.xx/blog>  |
---------------------------
```

Sometimes, it makes sense to omit the `FROM` clause from a query. This allows a graph to be passed to the query when it is executed. Avoiding binding the dataset to the query is good practice when using SPARQL from application code -- it allows the same query to be reused with different graphs, for instance. A graph is specified at runtime on the command line with the `sparql --data URL` option, where `URL` is the graph's location. This URL could either be the location of a local file or the Web address of a remote graph.

Executing SPARQL queries with the Jena API

While the command-line `sparql` tool is useful for running standalone queries, Java applications can call on Jena's SPARQL capabilities directly. SPARQL queries are created and executed with Jena via classes in the `com.hp.hpl.jena.query` package. Using `QueryFactory` is the simplest approach. `QueryFactory` has various `create()` methods to read a textual query from a file or from a `String`. These `create()` methods return a `Query` object, which encapsulates a parsed query.

The next step is to create an instance of `QueryExecution`, a class that represents a single execution of a query. To obtain a `QueryExecution`, call `QueryExecutionFactory.create(query, model)`, passing in the `Query` to execute and the `Model` to run it against. Because the data for the query is provided programmatically, the query does not need a `FROM` clause.

There are several different execute methods on `QueryExecution`, each performing a different type of query (see the sidebar entitled "Other types of SPARQL queries" for more information). For a simple `SELECT` query, call `execSelect()`, which returns a `ResultSet`. The `ResultSet` allows you to iterate over each `QuerySolution` returned by the query, providing access to each bound variable's value. Alternatively, `ResultSetFormatter` can be used to output query results in various formats.

Listing 4 shows a simple way to put these steps together. It executes a query against bloggers.rdf and outputs the results to the console.

Listing 4. Executing a simple query using Jena's API

```
// Open the bloggers RDF graph from the filesystem
InputStream in = new FileInputStream(new File("bloggers.rdf"));

// Create an empty in-memory model and populate it from the graph
Model model = ModelFactory.createMemModelMaker().createModel();
model.read(in,null); // null base URI, since model URIs are absolute
in.close();

// Create a new query
String queryString =
 "PREFIX foaf: <http://xmlns.com/foaf/0.1/> " +
 "SELECT ?url " +
 "WHERE {" +
 "      ?contributor foaf:name \"Jon Foobar\" . " +
 "      ?contributor foaf:weblog ?url . " +
 "      }";

Query query = QueryFactory.create(queryString);

// Execute the query and obtain results
QueryExecution qe = QueryExecutionFactory.create(query, model);
ResultSet results = qe.execSelect();

// Output query results
ResultSetFormatter.out(System.out, results, query);

// Important - free up resources used running the query
qe.close();
```

Refining a query's results

To further refine the results of a query, SPARQL has `DISTINCT`, `LIMIT`, `OFFSET`, and `ORDER BY` keywords that operate more or less like their SQL counterparts. `DISTINCT` may be used

only with `SELECT` queries, in the form `SELECT DISTINCT`. This will remove duplicate query solutions from the result set, leaving each remaining solution unique. The other keywords are all placed after the `WHERE` clause of a query. `LIMIT` *n* restricts the number of results returned from a query to *n*, while `OFFSET` *n* omits the first *n* results. `ORDER BY ?var` will sort the results by the natural ordering of `?var` -- lexically if `var` is a string literal, for instance. `ASC[?var]` and `DESC[?var]` can be used to specify the direction of the sort.

Or course, it is possible to combine `DISTINCT`, `LIMIT`, `OFFSET` and `ORDER BY` in a query. For instance, `ORDER BY DESC[?date] LIMIT 10` could be used to find the ten most recent entries in an RSS feed.

Writing more complex queries

So far, you've seen two ways to run a simple SPARQL query: using the command-line `sparql` utility, and using Java code with the Jena API. In this section, I'll introduce more of SPARQL's features, and the more complex queries they enable.

RDF is often used to represent *semi-structured* data. This means that two nodes of the same type in a model may have different sets of properties. For instance, a description of a person in a FOAF model may consist of only an e-mail address; alternatively, it could incorporate a real name, IRC nicknames, URLs of photos depicting the individual, and so on.

Listing 5 shows a very small FOAF graph, expressed in Turtle syntax. It contains descriptions of four fictional people, but the descriptions each have different sets of properties.

Listing 5. A small FOAF graph describing four people

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

_:a foaf:name         "Jon Foobar" ;
    foaf:mbox         <mailto:jon@foobar.xx> ;
    foaf:depiction    <http://foobar.xx/2005/04/jon.jpg> .

_:b foaf:name         "A. N. O'Ther" ;
    foaf:mbox         <mailto:a.n.other@example.net> ;
    foaf:depiction    <http://example.net/photos/an-2005.jpg> .

_:c foaf:name         "Liz Somebody" ;
    foaf:mbox_sha1sum "3f01fa9929df769aff173f57dec2fe0c2290aeea"

_:d foaf:name         "M Benn" ;
    foaf:depiction    <http://mbe.nn/pics/me.jpeg> .
```

Optional matches

Suppose you want to write a query that returns the name of every person described by the graph in Listing 5, along with a link to a photograph for each, if one's available. A `SELECT` query whose graph pattern included `foaf:depiction` would only find three solutions. Liz Somebody would not form a solution, because she has a `foaf:name` but no `foaf:depiction` property, and needs both to match the query.

Help is at hand in the form of SPARQL's `OPTIONAL` keyword. *Optional blocks* define additional graph patterns that do not cause solutions to be rejected if they are not matched, but do bind to the graph when they can be matched. Listing 6 demonstrates a query that finds the `foaf:name` of each person in the FOAF data in Listing 5, and then optionally finds the accompanying `foaf:depiction`.

Listing 6. Querying FOAF data with an optional block

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

Listing 7 shows the results of running the query in Listing 6. While all query solutions contain the person's name, the optional graph pattern is only bound when a `foaf:depiction` property exists; otherwise, it is simply omitted from the solution. In this regard, the query is similar to a left outer join in SQL.

Listing 7. Results of the query from Listing 6

```
-----------------------------------------------------------
| name           | depiction                              |
===========================================================
| "A. N. O'Ther" | <http://example.net/photos/an-2005.jpg> |
| "Jon Foobar"   | <http://foobar.xx/2005/04/jon.jpg>      |
| "Liz Somebody" |                                         |
| "M Benn"       | <http://mbe.nn/pics/me.jpeg>            |
-----------------------------------------------------------
```

An optional block can contain any graph pattern, not just a single triple as shown in Listing 6. The whole query pattern in an optional block must be matched in order for the optional pattern to form part of a query solution. If a query has multiple optional blocks, they act independently of one another -- each one may be omitted from, or present in, a solution. Optional blocks can also be nested, in which case the inner optional block is only considered when the outer optional block's pattern matches the graph.

Alternative matches

FOAF graphs use people's e-mail addresses to uniquely identify them. In the interests of privacy, some people prefer to use hashcodes of e-mail addresses. Plain text e-mail addresses are expressed using the `foaf:mbox` property, while hashcodes of e-mail addresses are expressed using the `foaf:mbox_sha1sum` property; the two are usually mutually exclusive in a FOAF description of a person. In situations like this, you can use SPARQL's *alternative matches* feature to write queries that return whichever of the properties is available.

Alternative matches are defined by stating multiple alternative graph patterns, with the `UNION` keyword between them. The query shown in Listing 8 finds the name of each person in the FOAF graph of Listing 5, along with either their `foaf:mbox` or their `foaf:mbox_sha1sum`. M Benn is not a query solution, because he has neither a `foaf:mbox` or a `foaf:mbox_sha1sum` property. In contrast with `OPTIONAL` graph patterns, *exactly* one of the alternatives must be matched by any query solution; if both branches of the `UNION` match, two solutions will be generated.

Listing 8. A query with alternative matches, and its results

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?name ?mbox
WHERE {
  ?person foaf:name ?name .
  {
      { ?person foaf:mbox ?mbox } UNION { ?person foaf:mbox_sha1sum ?mbox }
  }
}

-----------------------------------------------------------------------
| name                | mbox                                          |
=======================================================================
| "Jon Foobar"        | <mailto:jon@foobar.xx>                        |
| "A. N. O'Ther"      | <mailto:a.n.other@example.net>                |
| "Liz Somebody"      | "3f01fa9929df769aff173f57dec2fe0c2290aeea" |
-----------------------------------------------------------------------
```

Value constraints

The `FILTER` keyword in SPARQL restricts the results of a query by imposing constraints on values of bound variables. These value constraints are logical expressions that evaluate to boolean values, and may be combined with logical `&&` and `||` operators. For instance, a query that returns a list of names could be modified with a filter to return only names that match a given regular expression. Or, as shown in Listing 9, items in an RSS feed published between two dates can be found with a filter that places bounds on items' publication dates. Listing 9 also shows how SPARQL's XPath-style casting feature is used (here to cast the `date` variable into an XML Schema `dateTime` value), and how you can specify the same datatype on literal date strings with `^^xsd:dateTime`. This ensures that the date comparison is used in the query, rather than standard string comparison.

Listing 9. Using a filter to retrieve RSS feed items published in April 2005

```
PREFIX rss:   <http://purl.org/rss/1.0/>
PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>
PREFIX  dc:   <http://purl.org/dc/elements/1.1/>
SELECT ?item_title ?pub_date
WHERE {
      ?item rss:title ?item_title .
      ?item dc:date ?pub_date .
      FILTER xsd:dateTime(?pub_date) >= "2005-04-01T00:00:00Z"^^xsd:dateTime &&
             xsd:dateTime(?pub_date) < "2005-05-01T00:00:00Z"^^xsd:dateTime
      }
```

Working with multiple graphs

So far, all of the queries I've demonstrated have involved datasets consisting of a single RDF graph. In SPARQL terminology, these queries have run against the *background graph.* The background graph is what is specified by a query's `FROM` clause, by the `sparql` command's `--data` switch, or by passing a model to `QueryExecutionFactory.create()` when using Jena's API.

Other types of SPARQL queries

In addition to the `SELECT` queries used in this article, SPARQL supports three other query types. `ASK` simply returns "yes" if the query's graph pattern has any matches in the dataset,

and "no" if it does not. `DESCRIBE` returns a graph containing information related to the nodes matched in the graph pattern. For instance, `DESCRIBE ?person WHERE { ?person foaf:name "Jon Foobar" }` would return a graph containing triples from the model about Jon Foobar. Finally, `CONSTRUCT` is used to output a graph pattern for each query solution. This allows a new RDF graph to be created directly from the results of the query. You can think of a `CONSTRUCT` query on an RDF graph as somewhat analogous to an XSL transformation of XML data.

In addition to the background graph, SPARQL can query any number of *named graphs.* These additional graphs are identified by their URIs, and are each distinct within a query. Before examining the ways that named graphs can be used, I'll explain how to provide them to a query. As with the background graph, named graphs can be specified within the query itself, using `FROM NAMED <URI>`, where `URI` specifies the graph. Alternatively, named graphs can be provided to the `sparql` command with `--named URL`, with the URL giving the location of the graph. Finally, Jena's `DataSetFactory` class can be used to specify named graphs to be queried programmatically.

Named graphs are used within a SPARQL query with the `GRAPH` keyword, in conjunction with either a graph URI or a variable name. This keyword is followed by a graph pattern to match against the graph in question.

Finding matches in a specific graph

When the `GRAPH` keyword is used with a graph's URI (or a variable already bound to a graph's URI), the graph pattern is applied to whichever graph is identified by that URI. If matches are found in the specified graph, they form part of a query solution. In Listing 10, two named FOAF graphs are passed to the query. Query solutions are those people's names that are found in both of the graphs. Note that the nodes representing people in each of the two FOAF graphs are blank nodes, and have scope only within the graph that contains them. This means that the same person node can not exist in both named graphs in the query, and so different variables (`x` and `y`) must be used to represent them.

Listing 10. A query to find people described in two named FOAF graphs

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?name
FROM NAMED <jon-foaf.rdf>
FROM NAMED <liz-foaf.rdf>
WHERE {
  GRAPH <jon-foaf.rdf> {
    ?x rdf:type foaf:Person .
    ?x foaf:name ?name .
  } .
  GRAPH <liz-foaf.rdf> {
    ?y rdf:type foaf:Person .
    ?y foaf:name ?name .
  } .
}
```

Finding the graph that contains a pattern

The other way to use `GRAPH` is to follow it with an unbound variable. In such a case, the graph pattern is applied to each of the named graphs available to the query. If the pattern matches

against one of the named graphs, then that graph's URI is bound to the `GRAPH`'s variable. Listing 11's `GRAPH` clause matches on each person node in the named FOAF graphs given to the query. The matched person's name is bound to the `name` variable, and the `graph_uri` variable binds to the URI of the graph that matched the pattern. The results of the query are also shown. One name, A. N. O'Ther, is matched twice, because that person is described in both jon-foaf.rdf and liz-foaf.rdf.

Listing 11. Determining which graph describes different people

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?name ?graph_uri
FROM NAMED <jon-foaf.rdf>
FROM NAMED <liz-foaf.rdf>
WHERE {
    GRAPH ?graph_uri {
      ?x rdf:type foaf:Person .
      ?x foaf:name ?name .
    }
}

-------------------------------------------------------
| name                | graph_uri                     |
=======================================================
| "Liz Somebody"      | <file://.../jon-foaf.rdf>     |
| "A. N. O'Ther"      | <file://.../jon-foaf.rdf>     |
| "Jon Foobar"        | <file://.../liz-foaf.rdf>     |
| "A. N. O'Ther"      | <file://.../liz-foaf.rdf>     |
-------------------------------------------------------
```

Query results in XML format

SPARQL allows query results to be returned as XML, in a simple format known as the *SPARQL Variable Binding Results XML Format.* This schema-defined format acts as a bridge between RDF queries and XML tools and libraries.

There are a number of potential uses for this capability. You could transform the results of a SPARQL query into a Web page or RSS feed via XSLT, access the results via XPath, or return the result document to a SOAP or AJAX client. To output query results as XML, use the `ResultSetFormatter.outputAsXML()` method, or specify `--results rs/xml` on the command line.

Combining background data and named graphs

Queries may also use background data in conjunction with named graphs. Listing 12 uses the live aggregated RSS feed from PlanetRDF.com as background data, and queries it in conjunction with a named graph containing my own FOAF profile. The idea is to create a personalized feed by finding the most recent ten posts by bloggers that I know. The first part of the query finds the node that represents me in the FOAF file, and then finds the names of the people that the graph says I know. The second part looks for items in the RSS feed that are created by those people. Finally, the result set is sorted by the items' creation date, and limited to return only ten results.

Listing 12. Getting a personalized live PlanetRDF feed

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rss:  <http://purl.org/rss/1.0/>
```

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

SELECT ?title ?known_name ?link

FROM <http://planetrdf.com/index.rdf>
FROM NAMED <phil-foaf.rdf>

WHERE {
        GRAPH <phil-foaf.rdf> {
          ?me foaf:name "Phil McCarthy" .
          ?me foaf:knows ?known_person .
          ?known_person foaf:name ?known_name .
        } .

        ?item dc:creator ?known_name .
        ?item rss:title ?title .
        ?item rss:link ?link .
        ?item dc:date ?date.
      }

ORDER BY DESC[?date] LIMIT 10
```

While this query simply returns a list of titles, names, and URLs, a more complex query could extract all of the data present in the matched items. Using SPARQL's XML results format (see the sidebar titled "Query results in XML format") in conjunction with an XSL stylesheet, it would be possible to create a personalized HTML view of the blog items, or even produce another RSS feed.

Summary

The examples in this article should help you to understand the fundamental features and syntax of SPARQL, along with the benefits it brings to RDF applications. You've seen how SPARQL allows you to approach the semi-structured nature of real-world RDF graphs, with the aid of optional and alternative matches. Examples using named graphs have shown you how combining multiple graphs in SPARQL opens up your querying options. You've also seen how simple it is to run SPARQL queries from Java code using Jena's API.

There's a lot more to SPARQL than it is possible to cover here, so do use the Related topics below to find out more about SPARQL's features. Look at the SPARQL spec to learn in detail about SPARQL's built-in functions, operators, query forms, and syntax, or to see many more example SPARQL queries.

Of course, the best way to learn about SPARQL is by writing some queries of your own. Just grab some RDF data from the Web, download the Jena ARQ module, and start experimenting!

Related topics

- "Introduction to Jena," also by Philip McCarthy (developerWorks, June 2004), provides an overview of the Jena Semantic Web Toolkit, including details of the RDQL query language.
- The latest version of the SPARQL specification provides the definitive description of SPARQL's query syntax and functionality.
- Dave Beckett has produced a compact SPARQL Language Quick Reference that you can print out and pin up in your cube.
- SPARQL Query Results XML Format describes the results format discussed in "Query results in XML format."
- The Jena SourceForge project hosts documentation and downloads of the Jena Semantic Web Toolkit.
- The FOAF Project is an experiment in describing people and relationships using a machine-readable RDF vocabulary. The FOAF Vocabulary Specification defines the FOAF terms used in this article.
- SPARQL's graph patterns are based on the syntax of Turtle -- Terse RDF Triple Language.
- PlanetRDF.com aggregates weblogs of developers in the semantic Web community, providing a convenient means to keep up with developments in the field. This article uses PlanetRDF's RSS feed and RDF blogroll in its examples.