RDF, Jena, SparQL and the "Semantic Web"

Michael Grobe
Indiana University
Indianapolis, Indiana
USA
1.317.278.6891
dgrobe@iupui.edu

ABSTRACT
The Resource Description Format (RDF) is used to represent information modeled as a "graph": a set of individual objects, along with a set of connections among those objects. In that role, RDF is one of the pillars of the so-called Semantic Web. This paper describes how RDF-XML is used to serialize information represented using graphs, how RDF graphs can be read and written by using the Jena software package, and how distributed graphs can be queried using the SparQL query language. It includes examples showing how SparQL can be used to query data (such as the Gene Ontology) that is structured in hierarchies, and how SparQL queries can be submitted through SparQL "endpoints." It does not, however, delve into inference or the Web Ontology Language (OWL), but should provide a foundation for understanding those topics.

Categories and Subject Descriptors
H.3.3 [**Information Systems**]: Information storage and retrieval

General Terms
Languages, Standardization

Keywords
Resource Description Framework (RDF), SparQL, query languages, Semantic Web, Linked Data Web, URIs, metadata

1. INTRODUCTION
The Resource Description Format (RDF) [6, 9] is used to represent information modeled as a "graph": a set of individual objects, along with a set of connections among those objects. In that role, RDF is one of the pillars of the so-called Linked Data Web (nee Semantic Web). This paper describes how RDF-XML is used to serialize information represented using graphs, how RDF graphs can be read and written by using the Jena software package, and how distributed graphs can be queried using the SparQL query language.

SIGUCCS'09, October 11–14, 2009, St. Louis, Missouri, USA.

In general, these topics seem "simple," but are fraught with significant complications, limitations, and qualifications, especially when the casual user attempts to compare these components with their analogs in relational data approaches to the same or similar problems. As a result, this paper takes a "concrete" approach; it focuses on very basic examples and attempts to build on them in a systematic fashion to illustrate the various components underlying the use of graphs to represent data within the Linked Data Web.

2. USING GRAPHS TO REPRESENT DATA
Here are 2 graphs that represent 2 kinds of information associated with 4 different persons. The first shows the ages and the second shows the "favorite friend of each person:

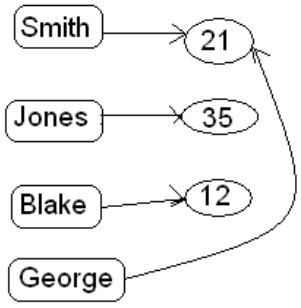

Figure 1. Graph #1: Person ages

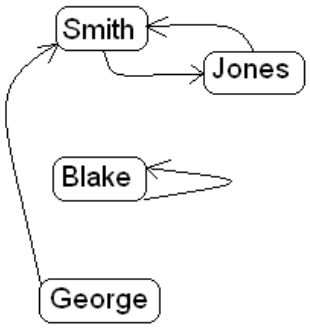

Figure 2. Graph #2: Favorite Friends

Figure 3 shows the 2 graphs combined using **named edges** to represent the same information associated with the same 4 persons.

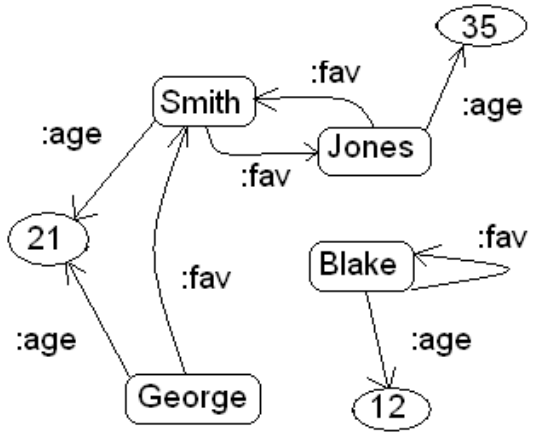

Figure 3. Graph #3: Person ages (:age) and favorite friends (:fav)

Read these links as "Smith has age 21" or "Jones has favorite friend Smith" to make them more "sentence-like". Each edge is like the "predicate" of a sentence, connecting a "subject" with an "object", so that the whole collection of edges represents a set of "sentences". Note, however, that a person node may be connected to more than one arc of the same type, and that a node may not be connected to every edge type. That is, Smith's age could be unknown, or Smith may not have a favorite friend, or may have not reported a friendship.

Such data is sometimes represented using so-called "blank nodes" to help cluster attributes. Figure 4 shows the graphs above reorganized using blank nodes.

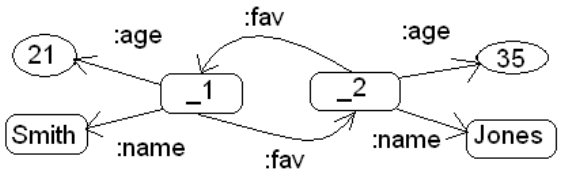

Figure 4. A portion of Graph #3 done with blank node.

Blank nodes are useful for specifying lists of items, but are discouraged within the Semantic Web.

2.1 Using URIs and URLs to Identify Edges

Now if it hadn't already happened, someone would certainly come up with the idea to use Uniform Resource Locators (URL) to point to Web documents that describe the exact meaning (semantics) of each edge type.

For example, some popular magazine could publish their definition of "favorite friend" on a page like

http://SomeCelebrityMagazine.com/fav

and other documents could define "BFF", "long-time-friend", "family-friend", "friend with benefits", etc, And, in fact, these definitions could themselves refer to other definitions like some "superset" of relationships such as:

http://SomeCelebrityMagazine.com/personal_relationships

or the personal_relationships file, itself, could include a **collection** of definitions, including "favorite friend, or "fav", that we might refer to as:

http://SomeCelebrityMagazine.com/personal_relationships#fav

using the # convention for targeting a specific location within a URL.

Such information serves to make the meaning both clear and sharable, so that other sources of personal information could employ the same predicate definitions within their own data collections.

Of course, for a lot of applications this would all be unnecessary; some arbitrary Uniform Resource Identifier (URI) can just be used to indicate an edge type known only to the file creator.

3. USING RDF TO SERIALIZE GRAPHS

These approaches to naming edges can be generalized to help represent, or "serialize" graphs in a text format, so they can be exchanged with other programs.

When graphs are serialized, each connection is again taken to be composed of 3 components, a so-called RDF "**triple**", composed of a "**subject**", "**predicate**" and an "**object**", where each edge becomes a named "predicate", although some writers speak of "object", "property", and "property value", rather than "subject", "predicate," and "object".

Each **subject** is represented as:

- a blank node, such as "_2", or
- a URI, like **http://fake.host.edu/smith**

Each **object** is represented as:

- a blank node,
- a literal value, such as **"some_value"^^some_type** where some_type is a URI, that defines a data type, as in **"Thomas_Jefferson"^^xsd:string**, or
- a URI

Each **predicate** is represented as:

- a URI, like **http://fake.host.edu/example-schema#fav**, as described above.

Note that URIs may appear in **abbreviated** forms within syntactic components like **example:age** and **xsd:string**, which will be expanded by substituting full URI values for the strings "example:" and "xsd:".

The following table displays Graph #3 as a set of 12 triples (3 for each person):

Subject	Predicate	Object
“Blake”	example:fav	“Blake”
“Blake”	example:age	"12"
“Blake”	example:name	"Blake"
“Jones”	example:fav	“Smith”
“Jones”	example:age	"35"
“Jones”	example:name	"Jones"
“George”	example:fav	“Smith”
“George”	example:age	"21"
“George”	example:name	"George"
“Smith”	example:fav	“Jones”
“Smith”	example:age	"21"
“Smith”	example:name	"Smith"

Table 1. Graph #3 as a set of triples, in which `http://fake.host.edu/example-schema#` is represented by the string “example:”

Here are two ways to represent the Graph #3 triples using RDF-XML. It may be important to note that the triple structures are somewhat obscured in both of these serialized representations of the data. The triple values remain, but they have been represented with less redundancy.

1) Properties encoded as XML **entities**:

```
<rdf:RDF
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
syntax-ns#"
     xmlns:example="http://fake.host.edu/example-
schema#">

    <example:Person>

        <example:name>Smith</example:name>

        <example:age>21</example:age>

        <example:fav>Jones</example>

    </example:Person>

</rdf:RDF>
```

2) Properties encoded as XML **attributes**:

```
<rdf:RDF
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
syntax-ns#"
     xmlns:example="http://fake.host.edu/example-
schema#">

   <rdf:Description  example:name=“Smith”

                        example:age=“21”

                           example:fav=“Jones”
     </rdf:Description>

</rdf:RDF>
```

3.1 Representing URIs

In work with RDF you will see URIs abbreviated in several ways, using: namespace, PREFIX and ENTITY definitions, depending on the context:

```
xmlns:lib=“http://some.host.edu/directory”
```

or

```
PREFIX <lib:http://some.host.edu/directory>
```

or

```
!ENTITY lib “http://some.host.edu/directory”
```

If the namespace abbreviation for “example” is substituted for each occurrence of “example:” in the Smith data encoding using XML **entities** above, then

```
<example:name>Smith</example:name>
```

is actually being represented as:

```
<http://fake.host.edu/example-schema#name>
   Smith
</http://fake.host.edu/example-schema#name>
```

3.2 RDF Resources to Model Each Person

Persons identified in Graph #3 can modeled as RDF “resources” by replacing the strings for each node identifier with URIs. Table 2 shows the result for the Blake data:

Subject	**Predicate**	**Object**
<http://fake.host.edu/blake>	http://fake.host.edu/example-schema#fav	<http://fake.host.edu/blake>
<http://fake.host.edu/blake>	http://fake.host.edu/example-schema#age	"12"
<http://fake.host.edu/blake>	http://fake.host.edu/example-schema#name	"Blake"

Table 2. Modeling Persons using RDF resources

These subject and object URIs need not be dereferenceable, but the general recommendation is to use dereferenceable URIs.

The entries in Graph #3 can also be represented as “RDF resources” in several ways. Here are 2, corresponding to the 2 formats described earlier. Both use “rdf:about” and “rdf:resource” attributes to specify resources:

Format 1

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
syntax-ns#"
```

```
   xmlns:example="http://fake.host.edu/example-
schema#">

   <example:Person
    rdf:about="http://fake.host.edu/smith">
      <example:name>Smith</example:name>
      <example:age>21</example:age>
      <example:fav
       rdf:resource="http://fake.host.edu/jones"/>
   </example:Person>

</rdf:RDF>
```

Format 2

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
syntax-ns#"
   xmlns:example="http://fake.host.edu/example-
schema#">

  <rdf:Description
     rdf:about="http://fake.host.edu/smith"
     example:name="Smith"
     example:age="21" />
     <example:fav
       rdf:resource="http://fake.host.edu/jones"/>

  </rdf:Description>

</rdf:RDF>
```

Note that the resource URI references in this example are not "real" documents; they are not "dereferenceable", though they could easily be made so.

If active URLs are used to identify resources, then this data can become part of the "**Gigantic Global Graph**", usually know as the Linked Data Web or the Semantic Web.

As Tim Berners-Lee writes [2]: "If HTML and the Web make all online documents look like one huge book, RDF, schema, and inference languages will make all the data in the world look like one huge database." And, in fact, there exist RDF browsers like the Firefox "Tabulator" plug in, that allow the user to follow URIs in RDF documents just as if they were HTML links. In addition the Tabulator will provide views of the data in multiple formats: XML, N3, Excel, etc.

4. USING GRAPHS IN RDF FORMAT

RDF graphs may be interrogated or manipulated in several ways**:**

- by physical inspection as text documents or by using a graphical display tool such as **RDF gravity** [14] which will render RDF input as graphical images that can be manipulated,
- by writing programs using packages such as Jena,
- by using command-line tools that apply SparQL queries,
- by using GUI interfaces accepting SparQL commands that are
  - written in text, or
  - represented graphically
- by submitting URLs containing form parameters to SparQL "endpoints" [16].

The next sections will discuss several of these approaches.

4.1 Manipulating RDF Graphs using Jena

The Java-based Jena package [9, 11] from HP Labs allows users to manipulate and query RDF graphs. You can write a program that uses Jena classes to

- retrieve and parse an RDF file containing a graph or a collection of graphs,
- store it in memory,
- examine each triple in turn, examine one component (say, the subject) of each triple in turn, or examine only triples that meet specified criteria, and,
- write a serialized version of a graph to a file or STDOT.

For example, one might examine each stored triple searching for a specific reference URI, or for a specific literal value, as with a search for triples containing a specific value, `"21"^^xsd:age`, in their object portions.

An RDF graph is stored in Jena as a "model", and a Jena model is created by a factory, as in:

```
Model m = ModelFactory.createDefaultModel();
```

Once a model has been defined, Jena can populate it by reading data from files, backend data bases, etc. in various formats, and once it has been populated, Jena can perform set operations on pairs of populated models and/or search models for specific values or combinations (patterns) of values.

For example, there are several methods for creating iterators over a model so you can access specific components. Iterators may be built by

- listing the components of each triple:
  - `model.listSubjects();`
  - `model.listObjects();`
- comparing a specific component with a specified value, as in:

```
model.listSubjectsWithProperty( Prop p,
        RDFNode object );
```

  which will get you a collection of subjects possessing property/predicate **p** and specific value **object**)

- comparing all components against specific values in 2 steps:
  - define a "selector" possessing specific values s, p and o, where **null** or **(RDFNode) null** matches anything:

```
Selector selector = new SimpleSelector(
    subject, predicate, object )
```

  - and then build the statement list:

```
model.listStatements( selector );
```

This selector capability is the basis for making ad hoc queries against the RDF data, and underlies the SparQL query language.

4.2 SparQL: A Graph-based Query Language

SparQL [7,10] is a language that lets users query RDF graphs by specifying "templates" against which to compare graph components. Data which matches or "satisfies" a template is returned from the query.

A triple template will contain variables that represent triplet components (e.g., a subject, predicate, or object within a triplet). For example the template:

```
   ?person  <example:age>  "21"^^example:age .
```

identifies a list of triplet subjects that have an example:age property of "21", and is analogous to asking "Who has age 21?" The SparQL query engine will return an exhaustive list of the subject component of triples that satisfy each query through value substitution. This is basically "query by example" (QBE) where the user defines an example pattern that the query engine will attempt to match using components from the data store.

This process is reasonably intuitive, and similar to QBE approaches applied to relational data and pattern matching within regular expressions or SQL.

SparQL is implemented in Jena through the **ARQ** package, and queries may be made from within Java scripts or via a SparQL client distributed with Jena.

Here is an example SparQL query that simply asks for a list of up to 10 of the subject and object portions of the triples in the file specified in the FROM clause:

```
PREFIX rdf:
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX example
   <http://fake.host.edu/example-schema#>
select $s $o
from
<http://myhost.edu/rdf-example-1.rdf>
where
{
  $s $p $o .
}
LIMIT 10
```

$s, $p,** and **$o are variable names that will each be assigned a value as the query is "**satisfied**," and the triplet pattern "**$s, $p, $o**" will match any triple that has 3 parts, so all triples should be displayed. Note that variable names may also start with "?", and may be full words.

A **subset** of the basic syntax of a SparQL **select** query is shown below:

```
BASE < some URI from which relative FROM and
     PREFIX entries will be offset >
PREFIX  prefix_abbreviation:  < some_URI >
SELECT
      some_variable_list
FROM
      < some_RDF_source_URL >
WHERE
{
    { some_triple_pattern .
      another_triple_pattern . }.
}
```

Notes:

- the "<" and ">" characters are required literals,
- the **BASE** and **PREFIX** entries are optional and **BASE** applies to relative URIs appearing in either **PREFIX** or **FROM** clauses,
- other commands that can appear in place of **SELECT** are: **CONSTRUCT**, **ASK** and **DESCRIBE**,
- ***** is a valid variable list, specifying any variable returned by the query engine, and may be preceded by **DISTINCT**, which will omit duplicate triples from the resulting list,
- there may be multiple **FROM** clauses, whose targets will be combined and treated as a single store,
- a "." separating multiple triple patterns is intuitively similar to an "and" operator,
- the term **WHERE** is optional, and may be omitted.

This syntax resembles SQL, and it has *similar* semantics. In particular, SQL semantics revolve around joining tables together and then looking through every row to see if the contents of row fields meet specified conditions. If one thinks of a collection of triples containing the same predicate as a (distributed) **table** named by the triplet predicate and containing 2 columns, the triplet subject and object, then the "." operator in SparQL queries is similar to a join, in which shared SparQL variables within triple patterns essentially define a join condition specifying equality.

Here is a SparQL query that can be used to search 4 files holding "live" data in the first representation format above:

```
PREFIX  rdf:
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  example:
   <http://fake.host.edu/example-schema#>
select *
from <http://myhostname.edu/smith>
from <http://myhostname.edu/jones>
from <http://myhostname.edu/george>
from <http://myhostname.edu/blake>
where
{
  $s $p $o .
}
```

If the data were all in one file, only one FROM clause would have been required. Here is a **representation of** the query results:

```
-----------------------------------------------
| s             | p            | o              |
===============================================|
| <myname/blake> | example:fav | myname/blake   |
| <myname/blake> | example:age | "12"           |
| <myname/blake> | example:name| "Blake"        |
| <myname/blake> | rdf:type    | example:Person |
| <myname/jones> | example:fav | myname/smith   |
| <myname/jones> | example:age | "35"           |
| <myname/jones> | example:name| "Jones"        |
| <myname/jones> | rdf:type    | example:Person |
| <myname/george>| example:fav | myname/smith   |
| <myname/george>| example:age | "21"           |
| <myname/george>| example:name| "George"       |
| <myname/george>| rdf:type    | example:Person |
| <myname/smith> | example:fav | myname/jones   |
| <myname/smith> | example:age | "21"           |
| <myname/smith> | example:name| "Smith"        |
| <myname/smith> | rdf:type    | example:Person |
-----------------------------------------------
```

where "myname" is an abbreviation for "http://myhostname.edu".

In this query all 4 files were searched as if they were in a single file. (Note that the URI contents are different in this live example.)

4.3 The Gene Ontology (GO)

The term "ontology" is used in different ways by different people. Pidcock [13] writes that "People use the word to mean different things, e.g.: glossaries and data dictionaries, thesauri and taxonomies, schema and data models, and formal ontologies and inference."

And Uschold [15] writes "An ontology may take a variety of forms, but necessarily it will include a *vocabulary of terms*, and some *specification of their meaning*. . .This includes definitions and an indication of **how concepts are inter-related** which collectively impose a structure on the domain and constrain the possible interpretations of terms."

As a working example, we will consider the Gene Ontology [1], widely used in bioinformatics and biological research. The Gene Ontology actually has 3 major components, separate sections for defining terms related to Biological Process (see the "namespace" entry below), Cellular Component (physical structures or locations within biological cells), and Molecular Function, each of which defines several thousand terms.

To begin unpacking Uschold's definition, we can look at two entries (of over 26,000) from the RDF version of the Gene Ontology. This listing includes the ROOT category, here known as "all", and the root of the molecular function component of the Gene Ontology, GO:0003674:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf=
    http://www.w3.org/1999/02/22-rdf-syntax-ns#
    xmlns:go=
    "http://www.geneontology.org/dtds/go.dtd#">
  <go:term rdf:about=
       "http://www.geneontology.org/go#all">
    <go:accession>all</go:accession>
    <go:name>all</go:name>
    <go:definition>
      This term is the most general term possible
    </go:definition>
  </go:term>
  <go:term rdf:about=
    "http://www.geneontology.org/go#GO:0003674">
    <go:accession>GO:0003674</go:accession>
    <go:name>molecular_function</go:name>
    <go:synonym>GO:0005554</go:synonym>
    <go:synonym>molecular function</go:synonym>
    <go:definition>Elemental activities, such as
catalysis or binding, describing the actions of a
gene product at the molecular level. A given gene
product may exhibit one or more molecular
functions.
    </go:definition>
    <go:is_a rdf:resource=
        "http://www.geneontology.org/go#all" />
  </go:term>
 </rdf:RDF>
```

Since all GO terms except the ROOT have at least one "is_a" relationship to other terms, a collection of GO terms can be easily represented as a graph, where all entries except the ROOT may have multiple parents, sometimes referred to as a "directed acyclic graph" or DAG. Figure 5 (drawn using RDF Gravity [14]) shows a DAG, inspired by Ashburner [1] but heavily modified, in which molecular function categories are shown as subtypes of one another. For example, this ontology asserts that the molecular function "chromatin binding" is a type of "DNA binding".

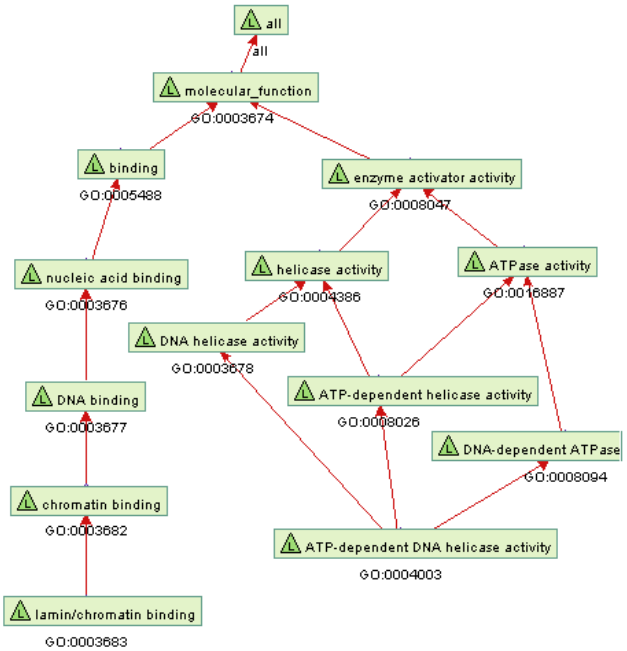

Figure 5. Some possible relationships among a subset of the terms in the GO molecular function ontology (example only)

Note that there is no gene data within the Gene Ontology, because the ontology is actually a **collection of terms** that can be used to **describe or annotate** genes. In fact, many sources of gene data use GO to annotate their contents.

4.4 A Gene Ontology Query

Here is a SparQL query to find **all of the parents of GO:0004003** in the example GO subset:

```
PREFIX xsd:
    <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf:
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX go:
    <http://www.geneontology.org/dtds/go.dtd#>
select *
from <http://hostname.edu/Some-GO-entries.rdf>
where
{
    http://www.geneontology.org/go#GO:0004003
    go:is_a  $parent .
}
```

Running this query produces a result like:

```
-------------------------------------------
| parent                                  |
===========================================
| http://www.geneontology.org/go#GO:0008094 |
| http://www.geneontology.org/go#GO:0008026 |
| http://www.geneontology.org/go#GO:0003678 |
-------------------------------------------
```

Here's a query that finds **all 3 element paths** up (towards the root) from GO:0004003:

```
PREFIX go:
    http://www.geneontology.org/dtds/go.dtd#
select *
from
  <http://hostname.edu/Some-GO-entries.rdf>
where
{
   <http://www.geneontology.org/go#GO:0004003>
        go:is_a   $a .
   $a   go:is_a   $b .
   $b   go:is_a   $c .
}
```

This query will print all combinations of the variables **$a**, **$b**, and **$c** that can be mapped to 3 triples in the data collection, such that the first element of a data triple "is a" subset of the third element of that triple, and that same third element is also the first element of another triple which asserts that element to be a subset of another third element, etc.

One might imagine that SparQL would allow traversing such an ontology to an arbitrary "depth", and/or derive inferences about category ancestors and descendents. However, these topics are beyond the scope of this paper, which only attempts to provide a foundation upon which to continue investigation of such issues.

4.5 Triplestores and SparQL Endpoints

In the examples so far presented, RDF has been shown as free-standing content that may be "published" by simply making it available via a Web server. There exist SparQL processors, such as Twinkle [8], that can retrieve such RDF files and use them to satisfy queries. However, most content appears to be stored and served from database management systems, sometimes called "triplestores," that have been customized to handle RDF. Two examples are Sesame [5] and OpenLink's Virtuoso system [12].

For example, both the **DBpedia** and the **Bio2RDF** "atlas of post genomic knowledge" are housed in OpenLink Virtuoso database management systems, and may be accessed through several SparQL interfaces, or "endpoints", enabled by Virtuoso. The "DBpedia is a community effort to extract structured information from Wikipedia and to make this information available on the Web" (http://dbpedia.org/About), and the Bio2RDF atlas (http://bio2rdf.org) currently includes over 2 billion triples taken from some 40 data resources useful to the bioinformatics community.

The following SparQL query requests a list of entries about "Goethe" from the DBpedia collection:

```
select distinct *
from  <http://dbpedia.org>
where
{
   ?s ?p ?o .
   ?o bif:contains "Goethe_Johann_Wolfgang"
}
```

Figure 6 shows this query entered through the DBpedia iSparQL interface at http://dbpedia.org:8890/isparql via Firefox.

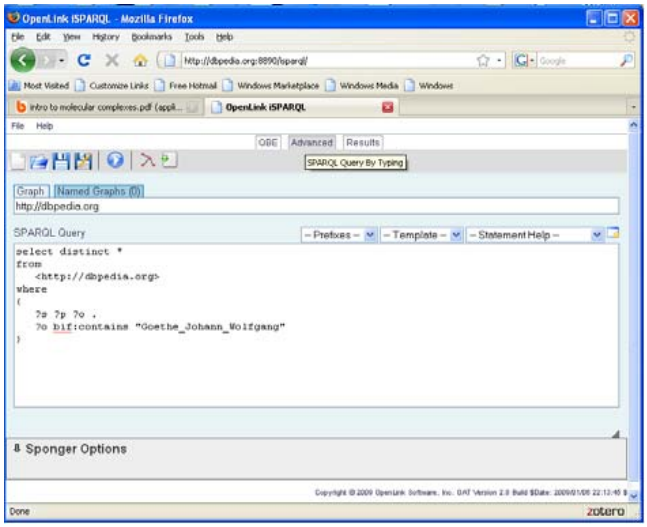

Figure 6. A SparQL DBPedia query via iSparql

Note that the predicate "**bif:contains**" is a special Virtuoso predicate that searches specially prepared back-end text indexes, that allow this query to ask for all triples that include an "object" containing the specified text.

Figure 7 shows the same query using the iSparql "graphical" Query-by-Example (QBE) interface which allows users to drag and drop components of the query graph that defines the query:

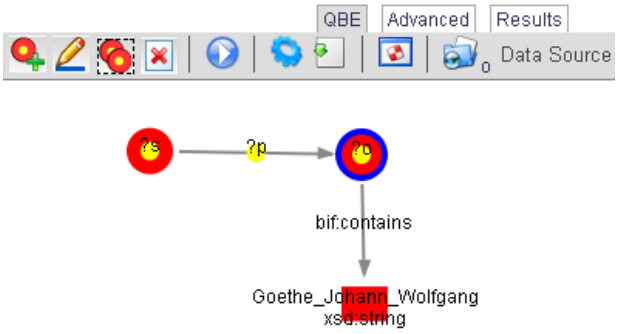

Figure 7. A SparQL query within the iSparql QBE interface

Since triple patterns within SparQL are themselves graphs, it's easy to see how SparQL can be adapted to this "graphical" medium.

The same query can be sent to the DBpedia SparQL endpoint embedded in a URL containing form parameters (with line breaks added for readability):

```
http://dbpedia.org/sparql?query=SELECT distinct *
WHERE { $s $p $o .
$o bif:contains "Goethe_Johann_Wolfgang" . }
```

4.6 Publishing Relational Data

One might ask how data already committed to storage in some other form, such as that within relational database management systems, might be made available.

The open-source **D2R server** [3], among others, does exactly this. To make SQL-based data available via D2R the user must:

- **interrogate** the database via JDBC using the **generate-mapping** script to build a configuration ("mapping") file from the relational table definitions, and then
- **start** the **D2R** server with the configuration file.

This seems "fairly" straightforward if one imagines that each table row becomes a separate resource/graph, primary keys (if any) become resource identifiers, and foreign keys become graph edges or guide incorporation into larger resources.

The D2R server exposes a SparQL endpoint to which users can send queries embedded within URLs or access a query form based on a Javascript component called SNORQL that employs AJAX. It also provides an interface for directly browsing content.

5. ACKNOWLEDGEMENTS

Thanks to Malika Mahoui of the Indiana University School of Informatics for reviewing an early version of this paper, and to Andy Arenson and Michel Tavares of the Pervasive Technology Institute at Indiana University for reviewing a later version.

6. REFERENCES

[1] Ashburner, M, et al., "Gene ontology: a tool for the unification of biology", Nature Genetics, 25, 25-29 (2000).

[2] Berners-Lee, Tim, "Linked Data", 2006. http://www.w3.org/DesignIssues/LinkedData.html

[3] Bizer, Chris, "The D2RQ Platform - Treating Non-RDF Databases as Virtual RDF Graphs", http://www4.wiwiss.fu-berlin.de/bizer/d2rq/

[4] Bizer, Chris, Richard Cyganiak, Tom Heath, "How to Publish Linked Data on the Web", 2007. http://www4.wiwiss.fu-berlin.de/bizer/pub/LinkedDataTutorial/

[5] Broekstra, Jeen, et al., "Sesame: An Architecture for Storing and Querying RDF Data and Schema Information", http://www.cs.vu.nl/~frankh/postscript/MIT01.pdf

[6] Davis, Ian, "An Introduction to RDF", http://research.talis.com/2005/rdf-intro/

[7] Dodds, Leigh, "Introducing SparQL: Querying the Semantic Web", 2005. http://www.xml.com/lpt/a/1628

[8] Dodds, Leigh," Twinkle: A SparQL Query Tool". http://www.ldodds.com/projects/twinkle/

[9] McBride, Brian, "An Introduction to RDF and the Jena RDF API ", 2007. http://jena.sourceforge.net/tutorial/RDF_API/index.html

[10] McCarthy, Philip, "Search RDF data with SPARQL", 2005. http://www.ibm.com/developerworks/xml/library/j-sparql/

[11] McCarthy, Philip, "Introduction to Jena", 2004. http://www.ibm.com/developerworks/xml/library/j-jena/

[12] OpenLink Software, "Virtuoso: Universal Server Platform for the Real-Time Enterprise, 2009. http://www.openlinksw.com/virtuoso/

[13] Pidcock, Woody, "What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model?", 2003. http://www.metamodel.com/article.php?story=20030115211223271

[14] Goyal, Sunil, Rupert Westenthaler, RDF Gravity (RDF Graph Visualization Tool), 2009. http://semweb.salzburgresearch.at/apps/rdf-gravity/

[15] Uschold, Mike, "Building Ontologies: Towards a Unified Methodology", AIAI-TR-197, 1999. http://www.aiai.ed.ac.uk/project/pub/documents/1996/96-es96-unified-method.ps

[16] W3C, "SparQL protocol for RDF", 2008. http://www.w3.org/TR/rdf-sparql-protocol