A First Generation Semantic Portal

November 3, 2005

This paper is about the semantic portal project. A semantic portal is a web site that allows users to explore data stored in a semantic web language. This project was meant to produce a proof of concept semantic portal based on data expressed in RDF and OWL. It was also meant to provide a working web site for the MINDSWAP group. This project was an expansion of an earlier project that produced a working portal with no reasoning capabilities. The older version did not allow us to take advantage of all the information stored in RDF and OWL when producing web pages.

1 Background Information

The portal project displays data which is stored in RDF and OWL. RDF and OWL are both recommendations released by the World Wide Web Consortium (W3C).

1.1 RDF

RDF is a triple based format for describing metadata. Usually RDF is expressed in an XML based format called RDF/XML. The triples in RDF consist of a subject, predicate, and object, in that order. The parts of an RDF triple are either universal resource identifiers (URIs), literals (strings), or anonymous nodes. An anonymous node is an identifier which is only valid within a single document and may change when the document is update. In RDF, the subject can be a URI or an anonymous node. The object can be a literal, URI, or anonymous node. The predicate must be a URI.

1.2 OWL

OWL (the Web Ontology Language) is a web based language for knowledge representation. OWL is expressed in RDF, so any OWL document is also a RDF document. OWL allows users to describe classes of objects as well as individuals. Individuals have properties. An individual which is a member of a class is also referred to as an instance of that class.

2 Goals

We had several goals in designing this portal:

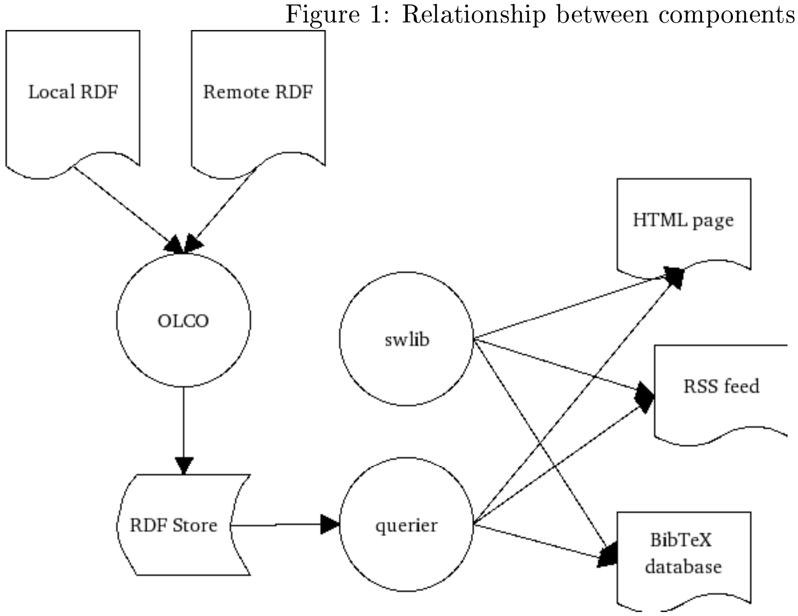
- to provide a useful API for querying a RDF triple store
- to provide a useful set of reasoning capabilities
- to display web pages as quickly as possible while meeting the other goals
- to separate query from display and be able to produce non-HTML web pages if necessary
- to be installable and modifiable by web-masters seeking to run a semantic portal
- to track provenance information about the data added to the portal

The current version of the portal meets all these goals. However, installing and modifying the portal is difficult enough that, while we have many copies of it running for different projects produced by our research group, most web-masters don't consider it worthwhile.

3 The site architecture

The portal consists of four major components: a reasoner and cannonicalizer (olco), a query API and evaluator (querier), a library to interact with the web server (swlib), and the actual web pages themselves. These are all built on top of the RDFLib triple store. The relationship between these components is visualized in figure 1 .

Figure 1: Relationship between components of the web site



3.1 Olco

When new data is added to the triple store a reasoner and canonicalizer is run. This is called olco (on load canonicalizer for OWL). Olco is a simple rule based system which creates new assertions in the triple store. It never deletes triples from the store since this creates problems when reloading data.

Because olco only adds triples to the store, when a file is modified olco can simply delete all the inferenced triples (without needing to know which ones are relevant), reload the data from the modified file, and reinfer. If olco deleted triples it might have to read in a fresh copy of all the original files, rather than only the modified one, before reinferring. Olco attempts to track provenance information as it adds triples.

3.1.1 Reasoning

The reasoning half of olco can make inferences about symmetric, inverse, functional, and inverse functional properties.

symmetric properties When olco sees a property which is declared symmetric it declares the property to be its own inverse.

inverse properties When olco sees that a property foo has an inverse bar it searches for all triples "a foo b" and asserts "b bar a" and searches for all triples "c bar d" and asserts "d foo c."

functional properties/inverse functional properties If a functional property has an inverse, olco declares the inverse to be an inverse functional property. If an inverse functional property has an inverse, olco declares the inverse to be a functional property.

inferring two things are the same with inverse functional properties Olco uses inverse functional properties to infer that two instances in the database are actually the same instance. If two instances have the same value for the same inverse functional property then olco asserts they are sameAs each other.

3.1.2 Canonicalization

The canonicalizing half of olco makes the following assertions:

sameAs Olco categorizes all the individuals in the data store into equivalence classes. It then picks one individual in each equivalence class and treats it as the canonical version. Each individual in that class is then declared mindswap:sameAs the canonical version

subclass Olco takes all the statements about subclass relationships and forms a directed acyclic graph with one root node which is owl:Thing. Any class which is not asserted to be a subclass of anything else is assumed to be a subclass of owl:Thing. Olco then expresses the directed acyclic graph in RDF using the mindswap:subClassOf property. Olco uses mindswap:subClassOf to distinguish this from the usual rdfs:subClassOf propery which is ____________________

subproperty The subproperty hierarchy is treated the same way as the subclass hierarchy.

3.1.3 Provenance

Each triple in the store has a context attached to it. This context includes information about the source of that triple (usually the URI of the source document) and when it was added to the store. When olco adds a new inferred triple to the store, it attempts to give it a meaningful context based on the contexts of the triples used in inferring this one. This allows the site to track provenance information using the sources of these contexts. For example, if a triple is inferred from two other triples then it would be given a new context that had all the sources associated with the contexts of the two original triples associated with it.

Olco provides the reasoning capabilities we desired for this web site. Because reasoning is performed when new data is added to the data store this step does not have to be performed each time a page is requested. This makes the web site feel more responsive. The canonicalization step is also designed to allow the query API to return results faster so that pages can generate their content faster.

3.2 Querier

The data in the triple store (which has already been processed by olco) is accessed through a query API called querier. Querier is an object oriented library which provides a simple frame based view of the data. An important feature of querier is that it allows programs to treat equivalent individuals as a single individual. This allows programs to request facts about an individual without considering whether there are any equivalent individuals those facts might have been declared about.

Querier consists of two major parts: the QuerierModel object and QuerierNode objects.

3.2.1 QuerierModel

The QuerierModel object has a number of methods that allow a program to retrieve sets of things such as all individuals that are members of a particular class. Some of the more important methods of QuerierModel are explained below.

individuals This method is passed a class and returns a list of all individuals that are members of that class.

specificIndividuals This method is passed a class and returns a list of all individuals that are most specifically members of that class. In other words, it returns all instances of the class that are not also instances of a subclass of that class.

subClasses This method is passed a class and returns all subclasses of the class (including subclasses of its subclasses and so on).

specificSubClasses This method is passed a class and returns the subclasses of the class that are not also subclasses of one of its subclasses.

subProperties This method is passed a property and returns all subproperties of that property similarly to subClasses.

specificSubProperties This method is passed a property and returns all subproperties of that property that are not also subproperties of one of its subproperties similarly to specificSubClasses

3.2.2 QuerierNode

Each QuerierNode represents an instance or a literal value. QuerierNode is also subclassed by QuerierLiteral and QuerierResource. The reason for this is that a literal can be the object of a triple in RDF but not the subject or the predicate, so some methods that make sense for a QuerierResource don't apply to QuerierLiterals. Some of the more interesting methods of these objects are listed below.

QuerierNode

labels This method returns a list of suitable labels for this node using the standard property rdfs:label and its subproperties.

equals This method takes another node and compares them, taking into account sameAs statements.

subjects This method takes a property and looks for triples of the form (*, property, this node). It takes into account sameAs and subproperties when doing this. It returns a list of the subjects found this way.

specificSubjects This method works like subjects, but it does not take into account subproperties.

hasType This method checks whether this node is an instance of a particular class.

rProperties This method looks for triples of the form (*, *, this node) and returns all the properties found this way.

QuerierResource

properties This method looks for triples of the form (this resource, *, *) and returns all the properties found this way.

objects This method takes a property and looks for triples of the form (this resource, property, *). It takes into account sameAs and subproperties when doing this. It returns a list of the objects found this way.

specificObjects This method works like objects, but it does not take into account subproperties.

Querier, in conjunction with olco, provides a useful API for querying a RDF triple store containing OWL data. This API is not limited to producing web pages. It lets us separate content and display on the site.

3.3 Interaction with the web server

A library called swlib manages the interaction between the individual pages and the web server. Swlib facilitates content negotiation, caching, and HTTP conditional GET support. Swlib also provides the standard header and footer for each page that requests it (based on a site configuration file).

Each page provides a list of content types it can produce, the functions to produce them, and how preferred they are. Swlib receives from the web server a list of content types the client will accept and information about which the client prefers. It uses this information to decide which function in the page to call to generate the content. This allows the same page to produce RSS in response to one client and HTML in response to another.

Caching generated pages in a separate database helps the site respond to large numbers of page requests in a short period of time. Many pages can be retrieved from the cache without any further work. It also can improve responsiveness for a single user if they repeatedly back up to the same indexing page. Conditional GET similarly reduces the load on the web site by allowing clients (particularly RSS readers) to avoid re-downloading a page that hasn't changed.

Because swlib provides a configurable header and footer, a web-master can dramatically change the look and feel of their site without modifying individual pages.

3.4 The pages

Each individual page uses querier to produce output which is returned to the web server through swlib. The pages are grouped together into page packs. Each web site has an installation script that copies a set of page packs into place and configures them to fit that site.

This allows a web-master who wants to create a semantic web site to choose a selection of suitable page packs and create an installation script to install them. The web-master can then create their own database of content, and even their own page packs, to customize the site.

Most of the pages output HTML. All of those pages output validating XHTML 1.0 which is formatted using CSS 2.1. Each site can provide its own CSS file to change the look and feel of the site.

4 Problems encountered and lessons learned

While writing this site we encountered many problems. We developed at least partial solutions to each of these. These problems are loosely grouped into three categories: inferencing, surface features, and data management.

4.1 inferencing

We encountered three main problems related to inferencing. One is that there is an inherent tradeoff between completeness, space, and time in reasoning. There is another tradeoff between read speed and write speed. A third problem is the difficulty of tracking provenance across reasoning.

4.1.1 completeness vs. space vs. time

Because the tradeoff between completeness, space, and time is pronounced for OWL reasoning (the best reasoners are prohibitively slow for use on a web site), we needed to consider what features we most needed a reasoner to support . Our conclusions were:

- inverse functional properties
- functional properties
- inverse properties
- symmetric properties
- transitive properties

- most specific class
- most general subclass

4.1.2 read speed vs. write speed

Inferencing can be performed on read, on write, or a combination of each. This decision doesn't necessarily affect the speed of the inferencing, but it does affect the users' experiences.

We chose to perform most of the inferencing when the data is modified since one of our goals was to return pages quickly.

Another option is to perform inferencing on read but cache the results. This may be feasible for some aspects of reasoning. However, inferring which instances are equivalent takes a prohibitive amount of time. Even if a page only needs to know which instances are equivalent to a given instance the reasoner must find all inverse functional properties and see if any of them imply equivalence between this instance and any other instance. This takes too long to do between receiving a page request and responding.

4.1.3 provenance

Provenance information is important on a web site that pulls information from different sources. A user may need to decide whether they trust the person who made a statement. Tracking provenance information across inferencing is difficult. For example, the querier interface partially abstracts away the triples so that it may appear to a script that (foo, bar, baz) is a triple but really the database contains (foo, sameAs, fnord) and (fnord, bar, baz).

The current website tries to create meaningful contexts for new triples it creates during inferencing. These contexts are not necessarily complete (they do not always contain all source documents used to infer this triple). Whenever the querier code uses RDFLib to query the database it adds any contexts seen to a list in memory. When a page wants to display provenance information it resets that list, performs queries, and then reads the list. This method clearly lacks granularity. However, in our experience on the current website it has been adequate.

Another approach would be to provide a provenance oracle that takes a triple and returns the sources used in inferring that triple.

4.1.4 handling broken source data

Particularly when we began creating the web site, many ontologies available online were malformed. We often ended up using a mishmash of OWL, DAML, and RDFS documents. Because the web site accepts this type of data and attempts to work with it as the author intended, the inferencing code must account for bad data.

On the current site the first thing the inferencer does is try to convert statements about classes, subclasses, etc. to a common form the rest of the inferencer can use. This keeps each function from having to correct for these problems. Where we have encountered more unusual problems with incorrect markup we have modified the code to detect them and avoid them.

Another solution would be to insist that submitted RDF be correct. One possibility is to refuse to load any file that isn't. Also, submitted DAML could be ignored.

4.2 surface features

4.2.1 anonymous nodes

Instances represented by anonymous nodes cause problems with link persistence. Often a page provides a link to another page that will display specific information about an instance. If that instance is anonymous there is no perfect information to include in the link to locate it. Using the node identifier in a link would work until the source file the instance came from was reloaded. This is a problem if another web site (i.e. google) links to ours or if a user bookmarks a page.

Our solution is to use the first option on this list that is available:

- A URI that represents the instance
- An inverse functional property and value
- A list of all properties and values of the instance

Because this is a very common problem, we encapsulated this functionality in two functions in querier. One function creates a string suitable to use in a link to represent a particular instance. The other parses those strings and return instances. Clearly in some cases the parsing function fails because the instance no longer exists. Additionally, the parsing function may fail if the link contains a list of properties and values of the instance and those have changed.

Another option would be to not produce links for any node that does not have a URI associated with it.

4.2.2 representing a DAG on a web page

One challenge we encountered is that the owl class hierarchy is a directed acyclic graph. Many of the traditional ways data is displayed on web pages (such as nested lists, outlines, and different sized headers) lend themselves to displaying trees. It is harder to represent a DAG.

On pages where this became a problem we generally chose to fully display each class only once, and provide a note to "see above" each subsequent time we encountered the class. We also sometimes ignore this problem and display classes and instances multiple times.

Another option would be to produce graphical output that fully displays the relations.

4.2.3 content negotiation precludes proxy caching

One easy way to increase the performance many web sites is to use a HTTP accelerator such as squid. However, common web proxies and http accelerators do not handle content negotiation. They have no easy way to tell that two browsers would have received the same answer to their request. This made it impossible for us to use squid to cache pages and reduce load on the server.

As mentioned above, we chose to provide a way for pages to request being cached in a SQL database instead. This let us conveniently invalidate the page cache whenever the RDF database changed.

Another approach to reducing the load on the web server would be to make the main pages static pages that are recreated each time the database changes. This couldn't be done for every possible page, but might work in approximately the same situations that caching does.

4.3 management

4.3.1 version control

We felt it was important to keep a history of RDF submitted to the site. Mostly this is a safety net in case important data is deleted.

We chose to simply keep all the old versions of files around as numbered backups. This is inefficient but simple. For the amount of data on the current site the inefficiency is not a problem.

Another possibility is to use an independent version control system.

4.3.2 easy submission/editing

We learned that users to do not like entering free form RDF or n3. These both have complicated syntax. Moreover, users are unsure what ontologies are being used and what properties various classes support.

forms One solution we have used is creating custom web forms for different tasks. The paper submission form is an example of this. Most of these forms can only create new data, not modify existing data. These are also very specific solutions. The information about what to ask the user for and how to translate it into RDF markup is hard coded into the script that produces the form and accepts the results.

external tools We have also modified external tools such as PhotoStuff and RIC to submit markup to the site. This is a more general solution but also doesn't allow editing yet.

4.3.3 how to change facts

One major difficulty in editing is finding out what RDF triples must be modified in order to change some fact on the site. For example, if a user "Bob" wanted to change a person's name he would first have to find which files assert that that person has a certain name. Then Bob would have to edit all of those files.

The current web page provides a partial solution to this problem. Provenance information displayed in the RDF instance viewer helps track the location of the files where a statement occurs. The management page allows users to filter by creator and file category. However, this method is cumbersome and won't always work.

One possibility would be to not store RDF in separate files. Then you could delete the offending triple from the store (although it's still possible that that triple is also being inferred).

4.3.4 tracking (meta)metadata

Originally, we stored metadata about files in the files themselves. People frequently found it difficult to remember to do this. We switched to providing an area of the submission page that lets users edit this information in a form. Unfortunately, it's hard to make this form save the information in the RDF files themselves. While it is possible to use RDFLib to add data to the file, the original formatting and comments are lost. Our solution is a second RDF store that contains only this metadata information.

Again, this problem is largely related to storing RDF in individual files.