Enhanced Search and Vocabulary Access for the Shipboard Automated Meteorological and Oceanographic System (SAMOS) using the Semantic Web Stack

Author: Nkemdirim Dockery

Center for Ocean-Atmospheric Prediction Studies (COAPS), Florida State University
2000 Levy Avenue, Building A, Suite 292
Tallahassee, FL 32306-2741

Florida State University, 600 W. College Avenue, Tallahassee, FL 32306

Major Professor: Dr. Xiuwen Liu
Committee: Mr. Shawn Smith, Dr. Peixiang Zhao, Dr. Sonia Haiduc
A project submitted to the Department of Computer Science in partial fulfillment of the requirements for a Master's in Computer Science

December 02, 2014

Contents

Introduction to ODIP and SAMOS

The Ocean Data Interoperability Platform (ODIP) is an international effort to remove the barriers hindering the effective sharing of data across scientific domains and international boundaries. ODIP includes many of the major organizations engaged in ocean data management in EU, US, and Australia. ODIP organizes workshops for building standards, and develops prototypes for evaluation [1]. It essentially allows for the dissemination of best practices and tools for the ocean sciences community.

The Shipboard Automated Meteorological and Oceanographic System (SAMOS) initiative is one of the core programs at Florida State University's (FSU) Center for Ocean-Atmospheric Prediction Studies (COAPS). Participants in the SAMOS initiative collect continuous navigational (position, course, heading, speed), meteorological (winds, pressure, temperature, humidity, radiation), and near-surface oceanographic (sea temperature, salinity) parameters on research vessels while at sea. One-minute interval observations are packaged and transmitted back to COAPS via daily emails, where they undergo standardized formatting and quality control. The SAMOS initiative is one of the programs tapped to contribute to the ODIP prototype addressing data interoperability and discovery [2].

ODIP and SAMOS Objectives alongside the Semantic Web

The objectives of ODIP and SAMOS mesh fairly well in both the short-term and the long-term. The ODIP emphasis on discoverability aligns with a more long-term objective for the COAPS data center. As an existing program with a scientific user-base, SAMOS provides the unique opportunity to mix high-level ODIP goals with interesting use-cases. If we view utilization as a measure of success, then it is important for potential science users to have some way of easily discovering SAMOS data. Any modern search for data will start from established search portals, so any effort towards discoverability should allow data to be indexed by these portals in order to increase exposure to potential users.

The ODIP emphasis on interoperability also easily aligns with long-term COAPS objectives. There are many instances where SAMOS data needs to be presented in different contexts to gain new insights. Let us consider a scenario between the SAMOS initiative and the Rolling Deck to Repository (R2R) program (for which COAPS is a funded collaborator). The R2R program handles another set of data from research vessels that overlap with the set of SAMOS vessels. While SAMOS' data is collected daily, R2R's data is collected per cruise. For those ships that fall under both SAMOS and R2R, it is important to be able to map SAMOS daily files to R2R cruise collections since the different organizations provide different types of quality control and post-processing that scientists are interested in comparing.

The long-term objectives mentioned are fairly typical for scientific organizations and are usually addressed individually through new web services, but what if there was a more generalized approach to interconnected data? The Semantic Web, a World Wide Web Consortium (W3C) effort, aims to be a solution to the demand of generalized, machine readable, more interconnected, yet organizationally separate data. The W3C describes Semantic Web as a technology stack to support a "Web of data." Semantic Web technologies

enable people to create data stores accessible to the Web, build vocabularies, and write rules for handling data. The Semantic Web technology stack includes standards such as RDF, SPARQL, OWL, and SKOS [3] .The technology stack was introduced to SAMOS as an ODIP recommendation. A quick overview of the components follows.

Semantic Web Overview

The Resource Description Framework (RDF) is a standard for data interchange on the Web. RDF has features that facilitate data merging even if the underlying schemas differ, and it specifically supports the evolution of schemas over time without requiring all the data consumers to be changed.

RDF extends the linking structure of the Web to use Uniform Resource Identifiers (URIs) to name the relationship between things as well as the two ends of the link (this is usually referred to as a “triple”). Using this simple model allows structured and semi-structured data to be mixed, exposed, and shared across different applications [3].

SPARQL is an RDF query language that is able to retrieve and manipulate data stored in the RDF format [4].
OWL and SKOS are ontology building tools. Conventional database structure is specified using schemas, but in the Semantic Web world, structural specifications are called ontologies. The term is borrowed from Philosophy and its usage tries to capture the point of view of the Web as an organization of knowledge, primarily by logical relationships. Therefore, in computer science and information science, an ontology is a formal framework for representing knowledge. This framework names and defines the types, properties, and interrelationships of the entities in a subject domain [5]. We use ontologies to limit complexity and to organize otherwise free-form information.

The selection of the Semantic Web stack will also help us satisfy a more short-term objective. Even though we collect and process a large volume of scientific data, our search capabilities are still somewhat limited. We also use a set of parameter names local to SAMOS instead of more standardized, international parameter names. The focus of this project is to enhance our search capabilities for the SAMOS data that we collect and make sure that our parameter names are mapped to internationally recognized names.

Satisfying these short-term objectives will enable us to answer questions like the following: “What records contain sea temperature within a certain geographic footprint when searching using Natural Environment Research Council (NERC) standard parameter names for sea temperature?” We essentially want to be able to search for data by ship, time, observed parameters, and geographic location. We will now revisit some of the components of the Semantic Web stack in more detail.

Technology Background

RDF

RDF is essentially serialized data of the form <URI> <relationship> <URI>. Uniform Resource Identifiers (URIs) are intended to be global references for different objects. They can have the format of web addresses, or they can be some other globally unique identifier like an ISBN number [6]. For example, here are some simple RDF statements serialized in N3 notation:

```
@prefix ex: <http://www.example.org/> .
ex:john      ex:type           ex:Person .
ex:john    ex:hasMother  ex:susan .
ex:john      ex:hasFather  ex:richard .
ex:richard ex:hasBrother ex:luke .
```

Here are the same statements serialized in RDF/XML:

```
<rdf:Description rdf:about="http://abyssal.coaps.fsu.edu/data/john?output=xml">
       <rdfs:label>RDF description of john</rdfs:label>
       <foaf:primaryTopic>
       <rdf:Description rdf:about="http://abyssal.coaps.fsu.edu/john">
            <j.0:type rdf:resource="http://abyssal.coaps.fsu.edu/Person"/>
            <j.0:hasMother rdf:resource="http://abyssal.coaps.fsu.edu/susan"/>
            <j.0:hasFather rdf:resource="http://abyssal.coaps.fsu.edu/richard"/>
       </rdf:Description>
       </foaf:primaryTopic>
</rdf:Description>
```

It is important to note the machine-readable focus of serialized RDF. The various serialization formats capture a large amount of relationships at a time and are not intended to be parsed visually. With serialized RDF you get an actual web document, much like an HTML page. In order to do anything useful with an RDF document, we have to look at the query language component of the Semantic Web Stack called SPARQL.

RDF was introduced above as a way for representing triples of information, but if we consider larger amounts of triples related to each other, the information starts to resemble a graph, as in Fig. 1.

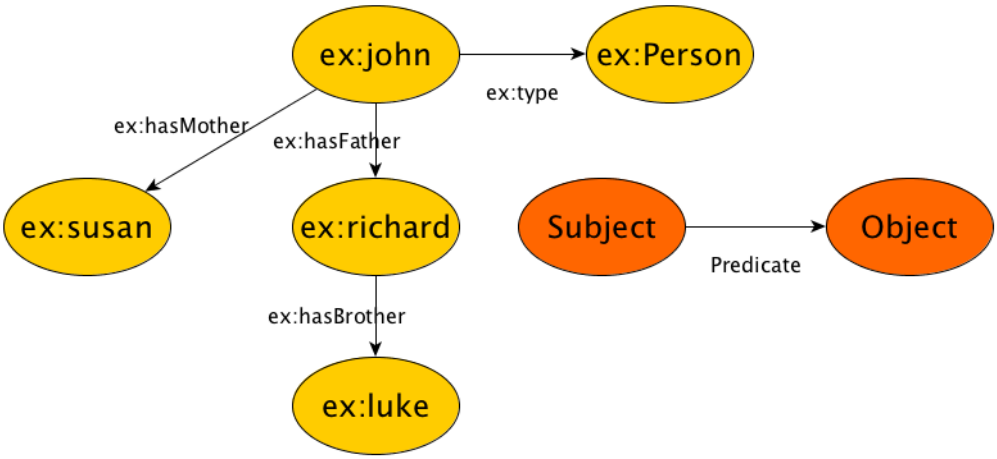
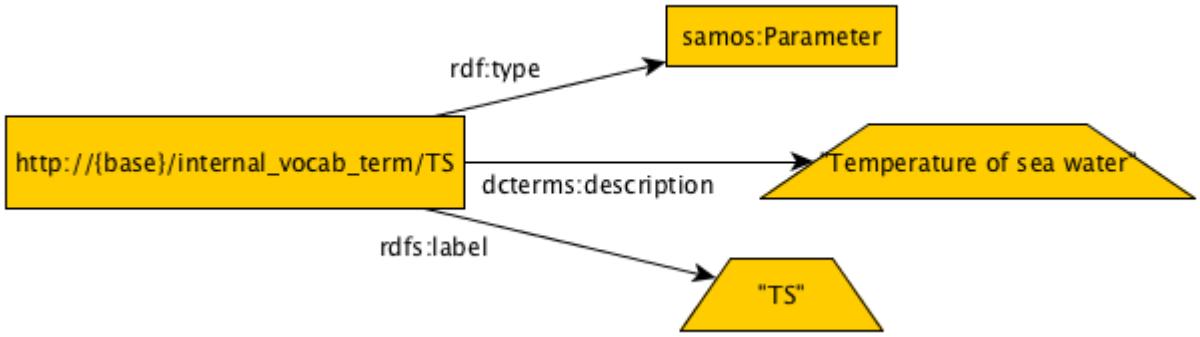


Figure 1. A graph of RDF statements

There are many access APIs for each serialization of the data but none of them are truly geared for querying graph-like data. X-Path, for example, is one of the more common access methods for data serialized in XML, but answering complex questions can require multiple X-Path API calls alongside programming logic to iteratively sift through results. This is because XML as a serialization is geared towards tree structured, or hierarchically structured data.



Consider the graph in Fig. 2. As users, we might ask a graph questions such as “What is the human-readable description for the term TS?” Asking RDF questions requires a query language, so this is where SPARQL comes in.

Another important component of RDF are namespaces. RDF namespaces are very similar to XML namespaces in that they define syntax, resource types and the interactions between these types. They are ultimately expressed using a base namespace defined by the RDF standard. Using namespaces we can express restrictions like “the edge called samos:callsign has a domain restricted to samos:Vessel, and samos:Datafile” meaning that the callsign edge goes from either a vessel resource or a datafile resource to a string literal.

SPARQL

SPARQL serves as the primary means to query graph-like data serialized in RDF. A SPARQL query for the above question might look like:

```
SELECT ?description
WHERE
{
       ?x rdfs:label “TS” .
       ?x dcterms:description ?description .
}
```

It is important to note that this query looks the same regardless of the underlying serialization. The queries are pointed to a SPARQL endpoint, much like a typical database connection. Endpoints are expected to be able to lift the graph-like data out of its concrete serialization to allow for expressive querying.

Most forms of SPARQL query contain a set of triple patterns called a *basic graph pattern*. Triple patterns are like RDF triples except that each of the subject, predicate and object may be a variable. A basic graph pattern *matches* a subgraph of the RDF data when RDF terms from that subgraph may be substituted for the variables and the result is RDF graph equivalent to the subgraph [7]. A query can also have restricting functions like FILTER, which further narrows the solution subgraph. FILTERs can match against string literals using regex, numbers, as well as arbitrary datatypes. The matching functions used for the FILTER constraint are also extensible. There are also keywords like UNION and MINUS to handle interactions between subqueries.

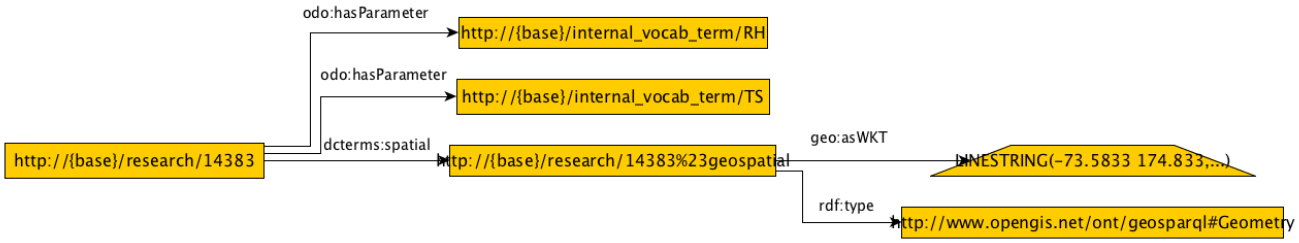


Figure 3. A graph with geospatial predicates and resources.

Let us take a look at a graph that also includes geographic information (Fig. 3). The advantage of a graphical query language and server is that we can more naturally apply graph algorithms when necessary. For example, questions like “Are there any data files that record sea temperature available for the Atlantic ocean?” easily map to a nearest neighbor calculation. An SQL version of such a query would include a large amount of joins and would quickly begin to degrade in performance [8] [9]. While there are geospatial extensions for SQL available with tools like PostgreSQL, they would take us away from the interoperability that the Semantic Web offers.

The geospatial extension for SPARQL is called GeoSPARQL. It is an open standard design by the Open Geospatial Consortium that addresses geospatial concerns in a Semantic Web

context. It provides for a small topological ontology (a set of topological blueprints) for representing geographic data using Geography Markup Language (GML) and well-known text (WKT) literals [10]. We can see an example of well known text in the graph above in the node pointed to by the edge labeled “geo:asWKT”. It also supports Egenhofer topological relationship vocabularies and ontologies for additional reasoning about topological objects [11].
The standard also extends the query language by providing additional FILTER functions. This means that in addition to regular FILTER functions like regex for string matching, we now have functions like “within” or “intersects” to use when building geographical queries.

In order to answer the question above, “Are there any data files that record sea temperature available for the Atlantic ocean?”, we might write a query like this:

```
select  ?x
where
{
       ?x odo:hasParameter ?y .  ?y rdf:label TS .   ?x dcterms:spatial ?sp .
       ?sp geo:asWKT ?g .
       ?z rdf:label “Atlantic Ocean” .
       ?z geo:asWKT ?gz .
       FILTER(geof:sfWithin(?g, ?gz))
}
```

This query assumes that a resource ?z with edges “geo:asWKT” “POLYGON(*[Atlantic Ocean boundaries]*)^^sf:WktLiteral” and “rdf:label” “Atlantic Ocean” is defined.

SPARQL Endpoints

To actually execute these queries, we need some sort of receiving agent. These are called SPARQL endpoints. SPARQL endpoints are essentially services that accept SPARQL queries and return results [12]. They are usually found as the outward facing entry point to an organization’s collection of RDF. They generally support multiple modes of access. They usually provide a web based interface for querying, an RDF/Semantic browser for human inspection or machine-driven scraping, as well as a programmatic interface resembling an actual database connection.

These endpoints can sit on top of the various serializations of collections of RDF documents, but it is important to note that the queries are being executed against the graphs captured from the concrete RDF documents. The translation from a concrete representation to a graph is performed by RDF stores like Apache’s Fuseki [13]. To put it differently, the underlying RDF serialization is hidden from the endpoint component by the RDF store. Instead, the serialization can be thought of as a graph on a higher level.

Components like Fuseki take in RDF serializations and make them available for querying as a graph. However, since the concrete representations can be abstracted away into a graph when it is time to execute queries, we can introduce a more generalized type of RDF store called triple-stores. These triple-stores are basically graph databases for directed, edge-

labeled graphs. Many graph database products are able to support this type of graph. Neo4j, for example, supports property graphs by default, but can easily support a directed edge-labeled graph to support an endpoint [14].

RDF, Discovery and Interoperability

RDF is also designed with automated discovery in mind. Hooks for discovery are tied to the “void” namespace. The void namespace is a standard set of RDF classes and properties that are used to express metadata about RDF datasets. The void standard reserves the URI pattern ‘.well-known/void’ to identify the resource containing metadata about the RDF documents available at that server. This means that any indexing service or web crawler that is interested in the semantic web has a direct hook to use for accessing your metadata.

Interoperability between RDF datasets from different organizations implies a certain amount of component reuse. Reusing well-known predicates and resource classes makes for easier application development between partner organizations. These predicates and classes form a specific namespace. For example, when describing people you can find most of the terms you need by looking at the “friend-of-a-friend”(foaf) namespace. For describing datasets, we can look at the “DCAT” and “void” namespaces. If a namespace does not have the exact term required, a custom term can usually be built as a subclass of terms that are closely related.

In short, the <URI><relationship><URI> structure of RDF documents allow for higher level representation as graphs. SPARQL and SPARQL endpoints allow for querying on this higher level representation. These graphs also leave room for generalized graph databases as endpoint serving tools. In addition, RDF uses a specific set of terms for discoverability. The use of standardized RDF terms allows different data to be more easily meshed. These constructs will be put together to form a SPARQL endpoint serving SAMOS’ file metadata and vocabularies.

SAMOS Enhanced Metadata Interface (SEMI) is our effort to combine the Semantic Web Stack components mentioned above with metadata drawn from SAMOS data quality processing to provide enhanced search and standardized mappings for our vocabularies. The web stack components offer enough functionality to meet ODIP interoperability and discovery goals, while providing enough flexibility for our enhanced search and mapping goals.

Current SAMOS SQL Usage and Data Access

SAMOS' data processing starts with the receipt of emails at a dedicated email address. As the email is processed, it interacts with a MySQL database at certain key processing stages.

1. The first stage of processing is Preliminary Processing. After the Preliminary processing has completed, the MySQL database will contain records describing the conversion of received emails from an ASCII file format to a network common data form (NetCDF). NetCDF is a self-describing, binary file standard for multidimensional scientific data. Additionally, the MySQL database will contain an automated quality control summary which consists of the number of observations flagged for certain conditions alongside other statistics. There may be more that one preliminary NetCDF data file created in the case of data spread over multiple emails.
2. The second stage of processing is Merge Processing. After merging, has completed, the MySQL database will contain records describing the coalescing of multiple stage one files into one NetCDF data file. It will also contain a quality control summary.
3. The third stage of processing is Visual Quality Control (VQC). This also generates a NetCDF file. An analyst uses a graphical user interface tool to flag suspicious or missing data. After VQC has completed, the database contains records tracking the update of a stage two NetCDF file into a stage three file. Another set of quality control statistics is generated.
4. After the second and third stage of processing, data files are passed to a supplemental script that adds a time averaged summary of the file to the database for use by various web services.

We will use these database interactions as data sources for SEMI's SPARQL endpoint.

Currently, SAMOS provides access to its collected and quality controlled NetCDF data files in a couple different ways:

- via A public facing FTP site
- via web forms
- via Thematic Real-time Environmental Distributed Data Services(THREDDS) server

The primary method used is the set of web forms since it is most accessible, and since it offers some search capability. Users can search for data by ship, date, and quality level, or they can search for data by cruise, which is essentially a set of days for which ships are out at sea, as opposed to the default of daily files. The issue for science users, however, is that they would rather use more scientifically relevant metadata as key search terms. Having a listing of ships and dates is not always sufficient. They might ask questions like:

- On which ships are datafiles containing observations of the SAMOS equivalent of NERC standardized sea water temperatures generated?
- Which data files for the 2009 - 2014 date range contain salinity in the Atlantic ocean?

THREDDS servers do have some search capability, but we currently have some compatibility issues with our NetCDF files and THREDDS. THREDDS for data retrieval also falls short of the flexibility and interoperability that ODIP encourages. THREDDS support for new functionality like external vocabulary mapping is lacking due to a slower pace of development and maintenance.

Enhanced Data Access

To enhance SAMOS' search capability, we want SEMI to be able to query on observed parameter and geographical footprint in addition to ship, date, and quality level.
We can answer the questions above by asking SEMI the following SPARQL queries:

- On which ships are datafiles containing observations of the SAMOS equivalent of NERC standardized sea water temperatures generated?

```
select DISTINCT ?z
where
{
?x odo:hasParameter ?y .
?y skos:related <http://vocab.nerc.ac.uk/collection/P07/current/CFSN0036/> .
?x samos:callsign ?z .

}
```

- Which data files for the 2009 - 2014 date range contain salinity (SSPS) in the Atlantic ocean?

```
where
{
?x odo:hasParameter ?y .
?y rdf:label SSPS .
?x dcterms:spatial ?sp .
?sp geo:asWKT ?g .
?z rdf:label "Atlantic Ocean" .
?z geo:asWKT ?gz .
FILTER(geof:sfWithin(?g, ?gz))
}
```

How can we get to a point where we can answer such queries?

Tools Research

In SAMOS' case, where processing is supported by an SQL database, to get to the point of answering these expressive queries, we first have to figure out the tools needed to make the step from SQL tables to an RDF-style graph. We then must do further digging to figure out what tools are available for storing and exposing this new graph through a SPARQL endpoint, all while keeping the requirements for things like geographic search in mind.

D2RQ

My first attempt at going from SQL to RDF was with the tool called D2RQ. D2RQ is a software layer that sits on top of SQL tables, does basic transformations of table-like data into RDF triples via a mapping file, and serves that data over an endpoint. In other words, it offers

RDF-based access to the content of relational databases without having to replicate it into an RDF store.

Mapping files use RDF statements as a declarative mapping language in order to describe the relationship between an SQL data model and a specific graph structure. Mapping files are serialized in Turtle (.ttl). The following example mapping file relates the table ‘Conferences’ in a database to the class ‘Conference’ in an RDF graph [15].

```
# D2RQ Namespace
@prefix d2rq:        <http://www.wiwiss.fu-berlin.de/suhl/bizer/D2RQ/0.1#> .
# Namespace of the ontology
@prefix : <http://annotation.semanticweb.org/iswc/iswc.daml#> .

# Namespace of the mapping file; does not appear in mapped data
@prefix map: <file:///Users/d2r/example.ttl#> .

# Other namespaces
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

map:Database1 a d2rq:Database;
    d2rq:jdbcDSN "jdbc:mysql://localhost/iswc";
    d2rq:jdbcDriver "com.mysql.jdbc.Driver";
    d2rq:username "user";
    d2rq:password "password";
    .
# ------------------------------------------------
# CREATE TABLE Conferences (ConfID int, Name text, Location text);
# maps to:
map:Conference a d2rq:ClassMap;
    d2rq:dataStorage map:Database1;
    d2rq:class :Conference;
    d2rq:uriPattern "http://conferences.org/comp/confno@@Conferences.ConfID@@";
    .
map:eventTitle a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:Conference;
    d2rq:property :eventTitle;
    d2rq:column "Conferences.Name";
    d2rq:datatype xsd:string;
    .
map:location a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:Conference;
    d2rq:property :location;
    d2rq:column "Conferences.Location";
    d2rq:datatype xsd:string;
    .
```

In this example you can get a sense of how relational database columns naturally unfold into a possible graph. A row’s primary key or identifier is turned into the URI of a resource using a template. The other column labels of that row are treated as descriptors or “properties” of the row identifier. Mapping statements are generalized to the entire table so the primary key-to-URI translation effectively represents the creation of a class of RDF resources, while individual primary keys are translated into individual RDF resources.

This idea of unfolding a table into RDF statements is useful, but usually represents the simplest case. As a relational database's schema is normalized, it hides a bit of semantic information. This is easily recoverable by SQL joins, but it means that tables are generally not in a form where they can easily be unfolded into a set of RDF statements. Taking this into account, mapping files also have constructs for more dynamic generation. The "d2rq:PropertyBridge" construct also has functionality for joins, where conditions, and full SQL expressions.

Here is an example of a property bridge using data from another table [15]:

```
map:authorName a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:Papers;
    d2rq:property :authorName;
    d2rq:column "Persons.Name";
    d2rq:join "Papers.PaperID <= Rel_Person_Paper.PaperID";
    d2rq:join "Rel_Person_Paper.PersonID => Persons.PerID";
    d2rq:datatype xsd:string;
    d2rq:propertyDefinitionLabel "name"@en;
    d2rq:propertyDefinitionComment "Name of an author."@en;
    .
```

Here is an example of a property bridge performing some text modification:

```
map:PersonsClassEmail a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:PersonsClassMap;
    d2rq:property :email;
    d2rq:uriPattern "mailto:@@Persons.Email@@";
    .
```

While these constructs extend the functionality of the mapping file, not all the information necessary to perform some of the searches that we would like is stored directly in the database. The SAMOS relational database is primarily geared towards processing. Getting information like the names of the produced files require combining processing data with templates based on particular directory structures. Alongside poor query performance by the D2RQ SPARQL endpoint and a lack of GeoSPARQL support, we gradually realized that the mapping file approach wasn't going to be the best fit for SAMOS.

Parliament

The next tool I came across was the Parliament triple-store. Instead of a mapping file approach, I started looking at graph databases with the idea of storing RDF directly. I searched with the primary criteria of GeoSPARQL support, since geographic search is such an important part of our short-term objective. The primary motivation behind the creation of Parliament was the need for a storage mechanism optimized specifically to the needs of the Semantic Web [16]. They cite a dissatisfaction with the storage of a directed graph in a relational database because "...the straightforward way to store the graph with the required level of generality is to use a single table to store all the triples, and this schema tends to defeat relational query optimizers" [16].

Parliament supports the majority of the Semantic Web stack, with support for advanced OWL features like inferencing (unused in this project) and GeoSPARQL. It is open-source. It has geospatial and temporal indices and uses a SPARQL endpoint as its primary interface. We'll have no built in mechanism for translation from SQL to RDF like a mapping file, instead, we'll construct and write RDF directly to the Parliament graph database. In order to write RDF to Parliament, we'll use the Apache Jena API for Java.

Persistent URLs Using Pubby

Many of RDF's discovery related benefits assume that the resources served by an organization have resolvable URIs. While discoverability is more of a long-term objective, providing resolvable URIs to aid that process is fairly straightforward. This is done with an optional Linked Data layer on top of the SPARQL endpoint using the Pubby Linked Data Frontend. Having a Linked Data layer means that Semantic Web applications focused on the non-SPARQL components of the Semantic Web Stack can still take advantage of SAMOS data [17]. When SAMOS' use of linked data is more established, automated discovery will be a lot easier to set up with this additional layer.

Schema Design

Though graph databases fall under the NoSQL and are quite flexible in terms of graph design, there is still an underlying structural approach for RDF. We organize terms by grouping them under descriptive URI's. These groupings are called namespaces and operate in much the same way that XML namespaces do:

```
@prefix samos: <http://abyssal.coaps.fsu.edu/dataVocab/> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dctype: <http://purl.org/dc/dcmitype/> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
```

We can also define classes that resources can take on as 'types':

```
samos:Quality
       rdf:type     rdfs:Class ;
       rdfs:label   "quality" ;
       dcterms:description "A tag describing level of quality control a SAMOS NetCDF file
has gone through" ;
       void:inDataset <http://abyssal.coaps.fsu.edu/SAMOS> .
```

We also can be specific about the subjects and objects allowed for a property by specifying domain and range classes in the RDF statements that define that property:

```
samos:quality
```

```
       rdf:type                    rdfs:Property ;
       rdfs:domain                 dctype:Distribution ;
       rdfs:range                  samos:Quality ;
       dcterms:description         "The quality control level of a file" .
```

Here are some resources using the definitions given above:

```
<http://abyssal.coaps.fsu.edu/dataVocab/Intermediate>
       rdf:type            samos:Quality ;
       rdfs:label          "intermediate" ;
       dc:type             "controlled_vocabulary_term" ;
       dcterms:description "Describes a second stage SAMOS NetCDF that has undergone
                           additional automated quality control and file merging" ;
       void:inDataset      <http://abyssal.coaps.fsu.edu/SAMOS> .

<http://abyssal.coaps.fsu.edu/intermediate/14355> samos:quality samos:Intermediate .
```

However we could also simply use ad-hoc properties at will. There is no requirement for the property or class definitions above. We could simply link a data-file and a quality string and hope the user can figure out the SAMOS specific meanings:

```
<http://abyssal.coaps.fsu.edu/intermediate/14355> quality “Intermediate” .
```

Certainly there is less effort involved in the ad-hoc approach, but we lose out on the descriptive features that promote reuse and a smoother learning curve for a given graph structure. This section will focus on the SEMI data model and the structural choices made.

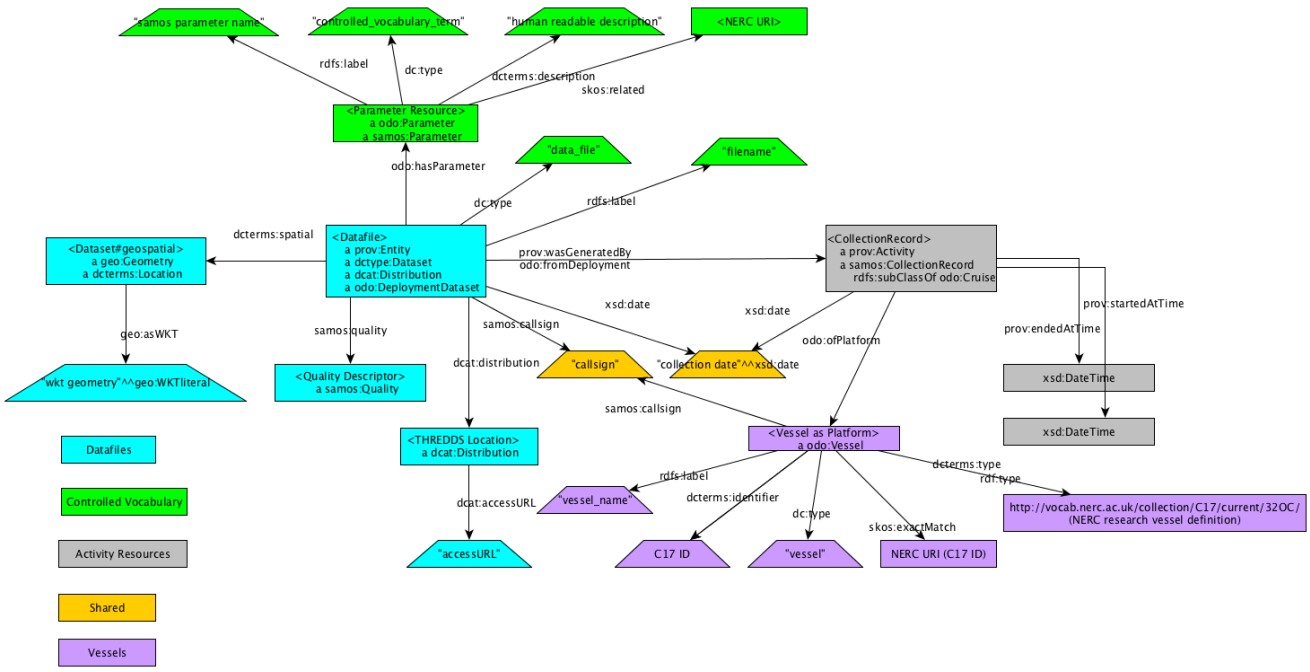


Figure 4. SEMI’s complete data model

I contributed alongside representatives from 5 other ocean sciences organizations at a Semantic Web workshop to the development of a Linked Ocean Data generic graph pattern [18]. The SEMI data model is SAMOS’ adaptation of the results of that workshop.

The graph, seen in Fig. 4, is aimed at describing three primary resources: Datafiles, controlled vocabulary resources, and activity resources.

Data file resources are top level nodes with a few children each that describe the SAMOS data files that we produce in a useful way. Each data file node has edges that point to a human readable label (a filename), a simple coverage date, a searchable geographic coverage, and a quality ranking. Data file nodes also have edges pointing to two other important types of nodes: Parameters and CollectionRecords.

Parameter nodes describe physical measurements. These nodes have edges that point to the SAMOS name for a particular physical measurement, as well as a full description of that measurement. These nodes also have edges pointing to related NERC vocabulary terms. The mappings of parameters to NERC vocabulary terms was done by my colleague Jocelyn Elya.

Collection Records represent a day-long period of data collection. While a date of collection and a ship identifier can be directly accessed from the data file node itself, a Collection Record is a more complete and interoperable way of representing related ship and date information. The domain of an odo:ofPlatform property is an odo:Cruise, but the usage of "cruise" as a descriptor is not very accurate for SAMOS' daily data collection. In an attempt to resolve this inaccuracy, Collection Records are also subclasses of odo:Cruise. Collection Record nodes have edges that point to the collection date as well as a prov style collection date range. They also have edges pointing to full vessel resources as opposed to just callsigns. These vessel resources in turn have C17 identifiers in addition to callsign identifiers. Since C17 ID are International Council for the Exploration of the Sea (ICES) based identifiers, they are more reliable than callsigns.

Namespace List

It is important to note that almost all of the namespaces used are existing namespaces coming from well-known brands.

- dc - Published by the Dublin Core Metadata Initiative (DCMI). This is a broad namespace that contains properties and classes that can describe metadata from a variety of sources. This is an older schema that does not specify the ranges of the various properties.
- dcterms - Published by the Dublin Core Metadata Initiative (DCMI). This is an updated version of the dc namespace that specifies the ranges of various properties. It also adds a large number of properties and classes.
- dctype - Published directly under the dcterms namespace. This namespace defines a broad set of data types for clarity and re-usability.
- dcat - Published by the World Wide Web Consortium (W3C). This is a namespace designed to facilitate interoperability between data catalogs published on the Web. Its properties and classes are closely tied to the DCMI namespaces.

- geo - Published by the Open Geospatial Consortium (OGC). This namespace implements the GeoSPARQL standard, which supports representing and querying geospatial data on the Semantic Web.
- odo - This namespace describes ocean data concepts. For this project, it is being used to relate a datafile to its observed parameters, as well as to the platform from which the data is collected.
- prov - Published by the W3C. This is a broad namespace that provides classes and properties that can describe the origin of data or documents in great detail.
- rdf, rdfs - Published by the W3C. These are broad namespaces that describe RDF itself. For this project, they are the definitive vocabularies for describing classes and properties.
- samos - Published by SAMOS. This is a local namespace used to identify concepts unique to SAMOS. It also serves as a placeholder for concepts that may be replaced by equivalents found in broader, more well-known namespaces.
- sf - Published by the OGC. This namespace is paired with the geo namespace. It describes simple geospatial features that can be operated on by GeoSPARQL.
- skos - Published by the W3C. SKOS stands for Simple Knowledge Organization System. This namespace aims to promote knowledge organization schemes such as classifications and thesauri. For this project, it is mainly used to describe the relationship between local SAMOS vocabulary terms and external NERC vocabulary terms.
- xsd - Published by the W3C. This is a broad namespace describes the XML schema. For this project, it is used because it describes important, basic data types such as dates and times.

I also added a metadata document based on VoID conventions. This metadata document is just a collection of regular RDF statements that use the VoID namespace to express important information. The following schematic describes the document's layout.

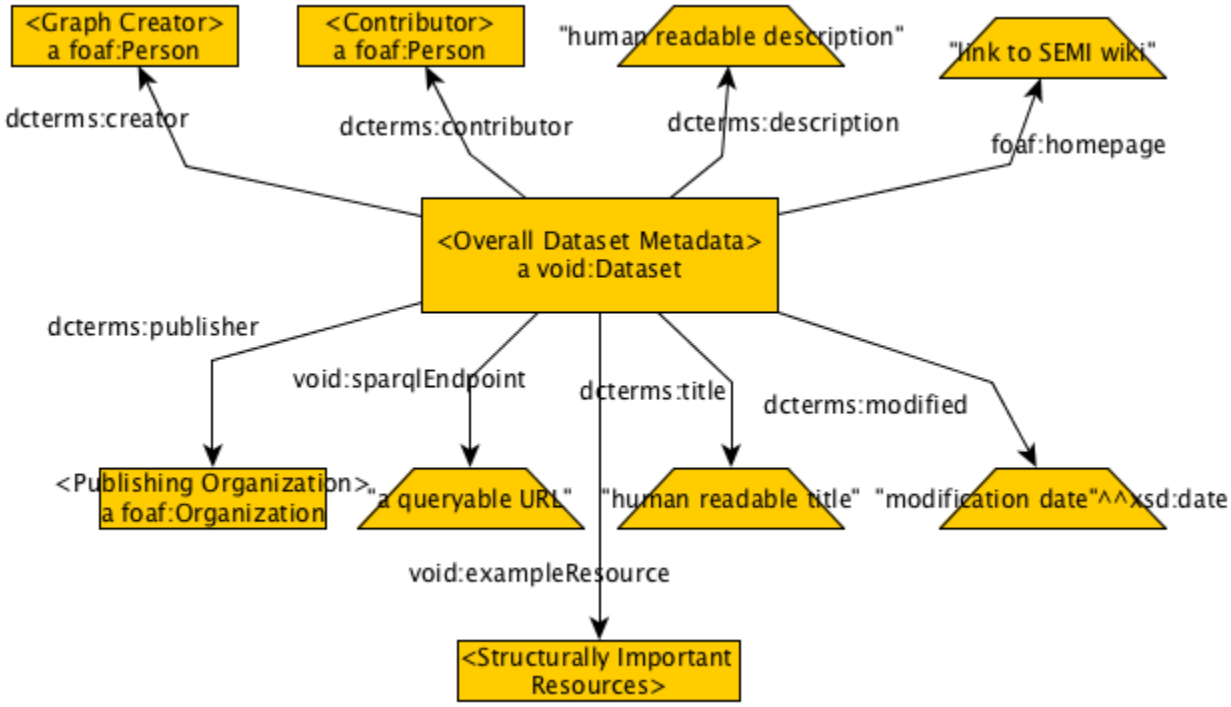


Figure 5. SEMI's VoID metadata layout.

As the schema describes, VoID documents capture basic identifying information as well as information that can help users acquire a basic understanding of the service without outside help (Fig. 5). The void:sparqlEndpoint property points to SEMI's SPARQL endpoint, while multiple void:exampleResource properties point to instances of the core resources of the graph: datafiles, vocabulary terms, and geographic objects. The VoID document currently serves as the base address for the server, but should also be aliased to conform to the well-known convention introduced above.

SQL to RDF Translation

Given that we have storage for our RDF resources, a SPARQL endpoint ready for exposure, as well as a fleshed out data model, we'll need a translation program that takes data scattered across our SQL schema and turns it into an RDF graph ready for writing to the Parliament graph database. The Java program is laid out in 4 major components: a main driver, a light SQL database wrapper, a vocabulary import, and a data import. The SQL to RDF translations are made in the import components

The vocabulary import component takes care of the VoID metadata RDF statements, identifying statements about our ships, and targeted vocabulary terms and their mappings. The database wrapper is a simple class that takes in database connection credentials from a configuration file. It provides basic querying methods, but has room for useful, reusable SAMOS queries that may come up in the future. The data import component

takes care of describing data files and creating any custom properties that we might need. The driver component provides the following command line interface for the program:

```
usage: java MainDriver [-d <arg>] [-p] [-t] -v <arg> [-w]
import_sql Help
   -d,--date_range <arg>     One or two dates for which to perform an import.
                             Date format: yyyyMMdd
   -p,--create_custom        Creates/Refreshes custom properties
   -t,--debug                Constructs RDF graph, but does not commit it to the
                             triple-store. Writes the resulting graph to std_out
   -v,--vessel <arg>         The vessel call sign for which to perform an import
   -w,--import_vocab         Import vocabulary terms and mappings.
```

This allows for incremental updates to the overall graph. The transformation program can therefore be setup as a nightly cronjob with the appropriate date range arguments in order to keep SEMI up-to-date.

Both import components string together API calls of the following form:

```
subject.addProperty(graph.createProperty(graph.expandPrefix("ns:predicate")),
processedValue);
```

Processed values represent a variety of data ranging from geographic serializations to parameter names. Namespaces (ns) and predicates are primarily pre-existing with a few custom created values. These calls result in RDF statements that conform to the data model above.

The addition of future SQL to RDF translations is fairly straightforward. Implement the translation in a new class and then add an option to the main driver along with any expected command line arguments.

Usage

There are currently two implementations of the transformation program available. They only differ in how they communicate with the underlying Parliament graph database. The first implementation builds the SQL to RDF translation completely in memory and then performs a bulk insert to a running Parliament instance's SPARQL endpoint. When performing large imports spanning multiple years and ships, this method does not have good time performance. It does, however have the capability to add statements to the graph while the server is running. This mode is probably more suited for nightly updates to the database.

The alternate implementation creates a graph that is actually backed by a Parliament instance. This means that the graph is modified directly as the SQL statements are translated. This method performs very large imports fairly quickly, but the server has to be turned off in order to have the updates be recorded. The two versions are filed under two separate git branches and are both operational.

Security

With a publicly available service like a SPARQL endpoint, it is important that some security concerns are addressed. A key security requirement of a SPARQL endpoint is the enforcement of limits to external querying. We ultimately want the endpoint to be available for public read-only access while password protecting update access. Another security requirement is user management. We would want to be able to internally manage graph modification based on user specific permissions.

Supported Functionality

Currently, it does not seem as if Parliament has security as a core concern. Remote clients can programmatically perform SPARQL Updates, which include both inserting and deleting triples from graphs [19]. There is also no native functionality available to grant and restrict endpoint usage and updates by user for Parliament.

Mitigation

At a minimum, if adequate security solutions can't be found, the endpoint URL can be password protected by our web container in order to limit access to internal use [20]. Since the APIs that make SPARQL connections do support credentials, secondary services built on the endpoint such as human friendly query builders can still be built by treating the underlying endpoint as a private connection and ensuring credentials are not exposed to the public. Having an endpoint that is only used in private is obviously suboptimal, so further research into SPARQL security is necessary if a secure endpoint is desired. The endpoint is completely separated from the processing database, so any security concerns mentioned here cannot propagate backwards to the processing setup. In the event that a more secure, GeoSPARQL compatible triple-store is found, migrating the SEMI data model would only require changing the connection parameters of the transforming program that identify the underlying RDF storage.

Future Work

- Merge the two import implementations into one command line program.
- Enhance query-building by adding resources that describe geographic features like oceans. Describing common geographic features will enable easier reference to well known terms like "Atlantic Ocean."
- Research possible security solutions for Parliament, as well as alternatives to Parliament that either support GeoSPARQL or at least store their data using the same serialization as the current data model, WKT.
- Find a programmatic source for C17 identifiers. They are currently grabbed from a static file.

Appendix

- Architecture Summary
    - Graphs and component explanations. Recap how it all fits together

Environment Dependencies

- CLASSPATH
- JAVA_HOME
- LD_LIBRARY_PATH - should point to the bin directory in any Parliament distribution
- PARLIAMENT_CONFIG_PATH - should point to the Parliament configuration file being used

Library Dependencies

The following libraries must be included in the classpath environment variable on a linux environment:

- apache-jena-2.11.1/lib
- parliament 2.7.4 /lib
- mysql-connector-java-5.1.29
- (apache) commons-cli-1.2
- (apache) commons-lang3-3.3.2

Acknowledgements

This work was funded primarily by the National Science Foundation (NSF), Oceanographic Instrumentation and Technical Services Program under grant #0917685 via FSU's contribution to the Rolling Deck to Repository project. Additional funding for student contributions to the ODIP platform are provided by NSF as a supplement to grant #0917685. SAMOS activities are base funded by the NOAA Climate Program Office, Climate Observation Division via the Northern Gulf of Mexico Cooperative Institute administered by Mississippi State University. Results and opinions expressed herein are those of the authors and do not necessarily represent the views of NSF or NOAA.

Works Cited

[1] ODIP. (2014) ODIP Welcome. [Online]. http://www.odip.eu/

[2] Center for Ocean-Atmospheric Prediction Studies. (2005) SAMOS Welcome. [Online]. http://samos.coaps.fsu.edu/

[3] W3C. (2013) Semantic Web. [Online]. http://www.w3.org/standards/semanticweb/

[4] Wikipedia. (2014) SPARQL. [Online]. http://en.wikipedia.org/wiki/SPARQL

[5] Wikipedia. (2014) Ontology. [Online]. http://en.wikipedia.org/wiki/Ontology_%28information_science%29

[6] Joshua Tauberer. (2008) Intro to RDF. [Online]. https://github.com/JoshData/rdfabout/blob/gh-pages/intro-to-rdf.md

[7] W3C. (2013) SPARQL 1.1 Query Language. [Online]. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/#basicpatterns

[8] Roberto V. Zicari. (2013) OBDMS Industry Watch. [Online]. http://www.odbms.org/blog/2013/04/graphs-vs-sql-interview-with-michael-blaha/

[9] Wikipedia. (2014) Graph Databse Properties. [Online]. http://en.wikipedia.org/wiki/Graph_database#Properties

[10] Wikipedia. (2014) Well-Known Text. [Online]. http://en.wikipedia.org/wiki/Well-known_text

[11] Wikipedia. (2014) GeoSPARQL. [Online]. http://en.wikipedia.org/wiki/GeoSPARQL

[12] Wikipedia. (2014) SPARQL. [Online]. http://en.wikipedia.org/wiki/SPARQL

[13] The Apache Software Foundation. (2014) Fuseki: serving RDF data over HTTP. [Online]. http://jena.apache.org/documentation/serving_data/

[14] Neo Technology Inc. (2014) Neo4j: Getting Started. [Online]. http://neo4j.com/developer/get-started/

[15] Robert Cyganiak and Chris Bizer. (2012) The D2RQ Mapping Language. [Online]. http://d2rq.org/d2rq-language#examples

[16] Ian Emmons. (2014) Parliament Welcome. [Online]. http://parliament.semwebcentral.org/

[17] Richard Cyganiak and Chris Bizer. (2011) Pubby: A Linked Data Frontend for SPARQL Endpoints. [Online]. http://wifo5-03.informatik.uni-mannheim.de/pubby/

[18] RPI Semantic Workshop. (2014) Linked Ocean Data Results. [Online]. https://github.com/LinkedOceanData/smackdown-results

[19] W3C. (2013) SPARQL Update. [Online]. http://www.w3.org/TR/2013/REC-sparql11-update-20130321/#security

[20] Rahul Kumar. (2013) How to Secure Specific URLs in Apache. [Online]. http://tecadmin.net/how-to-secure-specific-url-in-apache/

[21] W3C. (2014, March) Resource Description Framework. [Online]. http://www.w3.org/RDF/