Summer 8-6-2013

Semantic Assistance for Data Utilization and Curation

Brian J. Becker
bbecker@uno.edu

Semantic Assistance for Data Utilization and Curation

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

by

Brian Jonathan Becker
M.S. University of New Orleans, 2011

August 2013

Acknowledgements

I would like to thank my advisor, Shengru Tu, for much support and direction with this research. I would also like to thank Kevin Shaw from NRL for providing several of the data sets and many use cases relating to bathymetry, hyperspectral, and enterprise data.

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

We propose that most data stores for large organizations are ill-designed for the future, due to limited searchability of the databases. The study of the Semantic Web has been an emerging technology since first proposed by Berners-Lee. New vocabularies have emerged, such as FOAF, Dublin Core, and PROV-O ontologies. These vocabularies, combined, can relate people, places, things, and events. Technologies developed for the Semantic Web, namely the standardized vocabularies for expressing metadata, will make data easier to utilize. We gathered use cases for various data sources, from human resources to big enterprise. Most of our use cases reflect real-world data. We developed a software package for transforming data into these semantic vocabularies, and developed a method of querying via graphical constructs. The development and testing proved itself to be useful. We conclude that data can be preserved or revived through the use of the metadata techniques for the Semantic Web.

Ontology, Graph Databases, RDF, Jena, Provenance, Hyperspectral, Bathymetry, Enterprise Data, Social Networking

1 Introduction

Through a survey on the topics of the Semantic Web and databases, we have observed numerous cases of poor data management. Repeatedly, the mass of data always got out of hand eventually. They were often caused from old methods of archiving data as tapes shoved into a back room in a storage facility. The data eventually becomes impossible to manage, and remains as archives which serve little use aside from providing the curators of such data a little bit of comfort in that their "data is still there." The data may be there, but it has now become inaccessible unless massive efforts are undertaken to analyze and evaluate the utility of said data.

For now, we focus on organizations who have "lost" data in the aforementioned manner, and who are actively attempting to reclaim said data and make it available to researchers. This is mainly due to the fact that very few groups (who are not semantic web and ontology researchers) are actively interested in restructuring their data so that it is easier to manage. This data management is seen as just another expense. A feasible way to make data management widely accepted is to make it something interesting, and something more powerful than what was previously available. Just replicating a relational database system in a semantic web facade is not enough. More powerful queries must be available, and they must be easier to manipulate than with SQL. More relationships between all kinds of objects must be supported by a software system. And data should be stored in an accessible manner, or if not possible, should be referenced in an un-ambiguous way.

The purpose of the development of our framework is providing a way for users to make queries based on ontology vocabularies. While traditional database queries can be used to this purpose, and there are many toolsets available for making such activities easier, our software is designed for the case where the user does not fully understand the schema itself. Some basic field knowledge in a subject area will be required in order to explore the schema. However, our software is designed in such a way that domain experts can easily explore a subject area's ontology as well as other related ontologies. These other ontologies can serve as meta-ontologies to further aid in understanding of the schema.

2 Background

We utilized a number of technologies and toolkits to develop the framework of semantic assistance. This section describes the extent of our use of each of the given technologies. A few of the technologies listed here were not really utilized, though these include only the most notable alternatives.

2.1 Technologies

2.1.1 Resource Description Format and XML

Resource Description Format (RDF) is a data model with similar semantics to Extensible Markup Language (XML). It is a World Wide Web Consortium (W3C) supported format, much like XML, albeit being a much more recent development. Unlike XML, which has a single specified serialization, RDF offers standardized serializations[1] including XML.[2] Like XML, RDF data can be stored in nonstandard formats for various different purposes: making the data more human-readable, and making the data more conducive to random access through traditional database technologies.

While RDF is simply a data model, and does not in itself provide any means to work with the data, a wide array of tools and standards have been developed around it to support semantic data. This is antithesis to XML, which although containing schema support, requires the use of proprietary tools such as Saxon-SA (Schema Aware) to validate. The schema support in RDF is more mature, and is standardized in the form of RDFS.

An RDF data set, whether represented in textual form or in a database, has only a single data structure from which any other can be derived. This is called the triple, by analogy of a tuple from relational calculus. The three components of each triple are the subject, the predicate, and the object. These were named after the concepts of natural languages. A subject is what the metadata is about. A predicate is how the subject relates to another. An object is what the subject relates to, which is in itself another potential

Algorithm 1 Linked List

```
@prefix group: <http://example.info/groups#> .
@prefix example: <http://example.info/example#> .

group:Alice
        group:name "Alice" .

group:Bob
        group:name "Bob" .

group:Carl
        group:name "Carl" .

_:list1
        example:head group:Alice ;
        example:tail _:list2 .

_:list2
        example:head group:Bob ;
        example:tail _:list3 .

_:list3
        example:head group:Carl ;
        example:tail example:Empty .

group:Developers
        group:members _:list1 .
```

subject. This simple data structure does not limit what RDF can be used to represent, as triples can be used to represent any other data structures in many ways.[3]

A Simple Example: Linked Lists

We want to be able to represent lists in RDF, as they are a very common and useful data structure. Let's assume we want a property to represent group members for a hypothetical groups vocabulary.

This demonstrates how flexible RDF can be. However, despite this flexibility, we would probably like a more concise representation of a list. If using a format for serialization such as NTriples, which is simply a list of triples (without any prefix substitution) separated by newlines, we have no choice but to emit data such as this. However, with serializations such as Notation3, we can represent data in a far more compact way.[4]

This is the same exact data, in fact the same representation in low level triples, as the first part of the example. In the first part of the example, we created a list in the same manner as one would be created

Algorithm 2 Notation3 Linked List

```
@prefix group: <http://example.info/groups#>

group:Developers
        group:members
                ( [ group:name "Alice" ], [ group:name "Bob" ],
                        [ group:name "Carl" ] ) .
```

Algorithm 3 Spreadsheet

```
@prefix ss: <http://example.info/spreadsheet#>

ss:Sheet
        ss:tables ( ss:Table1, ss:Table2 ) .

ss:Table1
        ss:name "Table 1" .

ss:CellT1A1
        ss:table ss:Table1 ;
        ss:row 1 ;
        ss:col 1 ;
        ss:data "532"^^xsd:double .

ss:CellT1A2
        ss:table ss:Table1 ;
        ss:row 2 ;
        ss:col 1 ;
        ss:data "123"^^xsd:integer .

ss:CellT1B1
        ss:table ss:Table1 ;
        ss:row 1 ;
        ss:col 2 ;
        ss:data "Example Text" .

ss:CellT1B2
        ss:table ss:Table1 ;
        ss:row 2 ;
        ss:col 2 ;
        ss:data <http://example.info/groups#Developers> .


ss:CellT2A1
        ss:table ss:Table2 .
```

in a functional programming language such as Lisp, as that is the representation that RDF uses internally. This serialization is, nonetheless, far more concise and easier to read. Nonetheless, it takes the same amount of storage space in a database, unless the given database was designed to handle lists in a more optimized way.

A More Complex Example: Spreadsheet

This example is a little bit more complicated, but it will demonstrate that not only default structures available in popular serializations can be produced. We will create a representation of a simple spreadsheet which contains a number of tables (this is achieved through a list), and whose actual cells can contain any type of data.

As one can see, the first table has four cells which are filled-in with data. The second table has no associated cells. The cells are represented as the subjects, with a property representing which table, which row, which column, and which data element the cell contains. This representation of data looks quite cumbersome to actually use, but the query techniques we will explore within this paper demonstrate how we can effectively select and manipulate a range of cells.

This example is not intended to suggest that a valid use for this language is creating a spreadsheet application. This is a quite inefficient and complex way of representing data for something so trivial. However, this does mean a spreadsheet can be very trivially converted into an RDF representation. Of course, one may transform the data in RDF format to make it more usable for developers attempting to implement queries on the data, and for integration with other schemas.

2.1.2 OWL (Web Ontology Language)

An ontology is simply a formal representation of a knowledge domain. There are many technologies, in various stages of development, for representing ontologies. They are used in information systems for applying machine learning concepts to data.

The core concept of an ontology, from which other concepts are derived, is the shared vocabulary. A vocabulary is a predefined set of data structures which represent concepts. These concepts can be tangible, such as real-world objects.

The Web Ontology Language, or OWL, is actually a series of languages: OWL Lite, OWL DL, and OWL Full. These three languages, in ascending order of capabilities, are formal languages representing frame languages.[5] Specialized software designed for parsing and actually using OWL vocabularies are called Inference Engines, and one such library which supports integration with these Inference Engines is called Jena. This powerful library for working with semantic data will be explored further. Inference Engines are powerful tools, which require a lot of processing power due to their usage of machine learning techniques. For this reason, and because defining a set of restrictions and inferences is a complex task which requires understanding of machine learning concepts, the query builder environment which we propose does not initially consider the use of inferencing. Furthermore, these techniques do not scale well for the data we are focusing on, which is more suited to large static database implementations. Instead, we focus on developing customized views for data, which provides another more flexible way for individuals to create inferences for data.

2.1.3 SPARQL

SPARQL is the SPARQL Protocol and RDF Query Language. It is a query language, the a popular and standard query language for RDF.[6] However, it is incredibly powerful, providing most of the syntactical features of SQL. In this project, we focus solely on the **select** and **construct** query semantics, which, unsurprisingly, reflect the SQL **select** query. Select queries return a result table exactly like SQL. Construct queries are similar to select queries, though instead of representing a tabular set of results as SQL does, return an RDF graph (a list of RDF triples) which can be used as a view for another query, returned in a notation such as Turtle for easier reading, or returned as an actual graphical representation of the data.

We used SPARQL to implement the simple type of inferencing which was desired for our framework. SPARQL provided one syntax to do all manipulation of RDF data, making the implementation of the query builder far simpler. Likewise, the single interface developed for the purpose of inferencing and querying reduced the learning curve of the framework significantly. The semantics are also much more familiar for SQL users, further reducing the learning curve. As a construct query can return a graph, it can return data to be treated as inferences. This inferred data can then be left in memory as an RDF database in itself, or materialized by updating the original data. This flexibility allowed us to provide one interface for all desired manipulations on the data, which allows individual freedom in how to generate their own views of the data. Some users of the query engine will prefer simplistic views suited to a tabular notation, while others will certainly need to utilize the graph structure to a fuller extent.

2.2 RDF Data Store

The databases we considered for this framework were, for the most part, limited to RDF triple stores and SPARQL query engines. However, some other databases were considered, notably XML databases for their similar ability to use schemas and tree-like structures. However, the decision to use RDF was reached due to the prevalence of the technology in data aggregation. The ability to use a fully directed graph also afforded a lot more flexibility than a tree-like structure in XML, and RDF allows for more extensibility of schemas with other related schemas.

2.2.1 OpenLink Virtuoso

OpenLink Virtuoso is a powerful RDF triple store, coupled with many data processing utilities for extracting data from XML with GRDDL, as well as RDBMS.[7] User Interface Softwaret files. It is open source software with commercial perks, though the backend server required none of these. Nevertheless, the major issue we had with OpenLink Virtuoso was the fact that it was quite heavy-weight and had no apparent advantages over simple triple stores for the goals we were trying to achieve. It also required a lot more configuration, as it was apparent that this software was designed for large distributed system.

2.2.2 4store

4store is a Triple Store, developed in C++, intended to be a fast and powerful RDF database and SPARQL implementation allegedly capable of scaling up to 15 billion triples.[8] For our first use case, the employee database, we utilized 4store for the backend data storage which was queried by our visual query builder. 4store included its own HTTP backend which interfaced quite well with our frontend. Despite the efficiency gains, for further use cases, we reconsidered the use of 4store as SPARQL 1.1 support was not yet completely implemented.

2.2.3 Jena

Jena is an RDF API for Java, with an embedded triple store and a SPARQL query engine.[9] In addition to standard RDF data models, inferencing models are available which can automatically generate inferenced views based on OWL restrictions. Despite the issues with Jena's performance, we decided that Jena was a far more mature API for implementing the backend. The query builder backend we implemented with Jena was a Java HTTP servlet. For our translation backend, Jena was integrated within the translation software.

2.2.4 Zorba XQuery

The Zorba XQuery engine is a tree-structured database system which is designed for data in XML formats.[10] It uses a binary representation of XML for efficiency purposes, yet exposes interfaces for the XML query languages such as XPath and XQuery, with various different extensions as options. As of 2012, Zorba was a mature XML Database following NoSQL principles. There was no SQL-like query language, as well as an obscure methodology for extending XQuery functions. The standard libraries included for data manipulation were rather limited. It was, however, one of those most powerful XQuery implementations which did not depend on proprietary extensions for necessary features such as XML Schema support. For those reasons, we decided that Zorba was not a viable option for our database backend. In current times, things may have been different, as Zorba has matured to have support for advanced SQL constructs. Additionally, there are now a lot more APIs, such as Data Converters, Formatters, Data Cleaning, far more Excel function analogues, as well as Geological Projections. Many of the tools that were made available in Zorba would have greatly assisted us in the implementation of our framework, though they were not implemented features at the time of implementation.

2.3 User Interface Software

As this project is intended to provide a user-facing application that is both easy to use and powerful, we decided that a major portion of the framework should focus around having a powerful user interface for the development of queries. As queries need to be performed by those who are working in fields other than Computer Science and IT, we came up with a design which did not require a lot of technical knowledge relating to semantic data, or any incredibly steep learning curve for its users to perform most complex tasks.

2.3.1 DHTMLX

For the development of a Javascript Web Application frontend to the visual query builder software, we decided on using a user interface library to provide the necessary visual elements. DHTMLX was one of our options, and was used in the construction of the initial prototype of the frontend. This proved to

be quite inflexible, and required far more time (for library workarounds) than even implementing a simple user interface of our own. Many issues arose from the fact that the library was GPL licensed open source software, however there was a paid commercial version. It turned out that the DHTMLX codebase was heavily obfuscated and any modifications were very tedious. In addition, many features which later turned out to be somewhat necessary were only in the commercial version of the software.[11]

2.3.2 Sencha (ExtJS)

The Sencha toolkit was the second option, and was used to revise the toolkit which was initially prepared for DHTMLX. The development effort with Sencha was far smoother, quickly allowing the placement of all needed visual elements such as the vocabulary tree, instance panel, and main query builder panel. Sencha suggests and provides for a Model View Controller development pattern with Models, Stores, Views, and Controllers.[12] This allowed for simple integration with the query server. We used a custom JSON data format for serializing the vocabulary on the server to be provided to the client, as parsing RDF data in Javascript would have performed poorly and would have required the implementation of a specialized library. The server was also designed to return a list of instances of a given class, implemented via a SPARQL query on the database. Additionally, the server responded to SPARQL queries generated by the user (client-side) explicitly. This simplistic design decision has potential security risks which may need to be reviewed before this project can be considered production-safe.

2.4 Data Manipulation Software

Before this project was truly off the ground, we surveyed numerous different RDF and Data Manipulation technologies. This list describes which applications were seriously considered in the development. We also note the rationale described for using, or not using, any given piece of software.

2.4.1 ARC2 RDF Backend

The ARC2 RDF backend, a now-discontinued PHP library for handling RDF data, was used in the visual query builder backend. It offers a simple model for querying objects for their properties,[13] and was primarily used in the generation of the class hierarchy. Initially used to generate an instance list, it was replaced by the native HTTP SPARQL endpoint which 4store provided. This improved performance quite significantly and removed a dependency to the now deprecated software. The class hierarchy implementation remains in ARC2, but this is a trivial backend to reimplement in either Jena with Java Servlets or another server-side web development language that has a native RDF library. The ARC2 backend uses flat files to be deserialized and converted into a class hierarchy, which would also become a performance bottleneck in large systems.

2.4.2 Protege

One of the initial tools we investigated was Protege, an OWL vocabulary and database editor from Stanford.[14] Protege contains options for editing vocabularies, creating inferences, and defining instances of objects. Many plugins are available for Protege, some of which provide visualizations and others which allow for querying data. We decided that Protege was a good option for initially constructing several test databases on a small scale. However, for larger scale databases the instance editor functionality of Protege is mostly useless, as the database must be stored as a large flat file that is loaded into memory each time it is edited.

2.4.3 HDF-Java

In order to translate HDF data used in the hyperspectral imagery use case, we utilized the HDF-Java library available from the HDF Group.[15] We used a static mapping from the most important components of the HDF document which we believed to be the most suitable for metadata. Likewise, the data within HDF itself was not included, due to considerations that it was not useful for our purposes and would only lead to massive memory usage. As a compromise between availability of data and querying power, we added a set

of statistical functions from the Apache Commons Math library as metadata. As it was a static mapping from two HDF databases from the same field of endeavor, running the translation against a general HDF document may not return the best set of metadata obtainable for that document. Nevertheless, this was a compromise between flexibility and obtainability, as we had a limited subset of HDF data to work with.

2.4.4 Marc4j

For translation of MARC and MARCXML records used in the library database use case, we used Marc4j, an interface for these formats in Java. These formats are machine-readable catalogue formats, developed by the Library of Congress during the 1960s.[16] As expected due to its age, long before linked data became prevalent through the World Wide Web, the data format is flat in nature. Also due to the flat file model, as opposed to HDF-Java, Marc4j's model resembles evented SAX rather than XML DOM. Nevertheless, implementing the translation layer was more trivial than the other translation layers we used. As with our use of HDF-Java, we created a static mapping between the MARC format and the RDF format. However, MARC is specified for only a single domain, therefore the static mapping should equally apply to all data sets using one of these formats. No additional information was added from external sources, due to the nature of the source format being solely metadata.

3 Architecture

The framework developed for this research involves three different major components, as described in Figure (1). The first component developed was a visual query builder written in C++ (4store database backend), PHP (backend glue) and JavaScript (frontend). This necessitated the creation of a database in RDF format, which was initially generated manually in Protege for the Employee Database use case. For the Bathymetry database, we used Taverna to track process provenance information between different steps of data processing, and excluded the entity provenance information because this information was not readily available to use. After these use cases, we decided that different types of data sources were required, for example Excel (CSV), HDF, and XML. Therefore, we developed a translation software which utilized the Apache Jena library to convert tabular and hierarchical data into a temporary schema for further translation. To visualize the results of various queries, we settled on the rapper command line utility to generate directed graphs of query results and the GraphViz graph visualization library.

3.1 The Translation Component

3.1.1 Transitory Mappings

The mapping component of this query is not required for all of the use cases, just the ones which use real-world data which is not native to RDF/XML.

The first part of the translation component of our framework involves a set of transitory mappings between the following formats and RDF:

- HDF (Hierarchical Data Format)
- CSV (Comma Separated Values)
- MARC
- XML

The HDF translation source was utilized in the Hyperspectral data set, in order to convert the metadata and do some slightly more advanced calculation of statistical information. The CSV translation source was used to implement the Enterprise Projects data set. XML was used to implement the Structured Descriptive Data use case.

This part of the translation framework was implemented in Java using the Apache Jena framework for RDF data. The jCSV library was used for iterating through CSV tables generated from Excel documents.

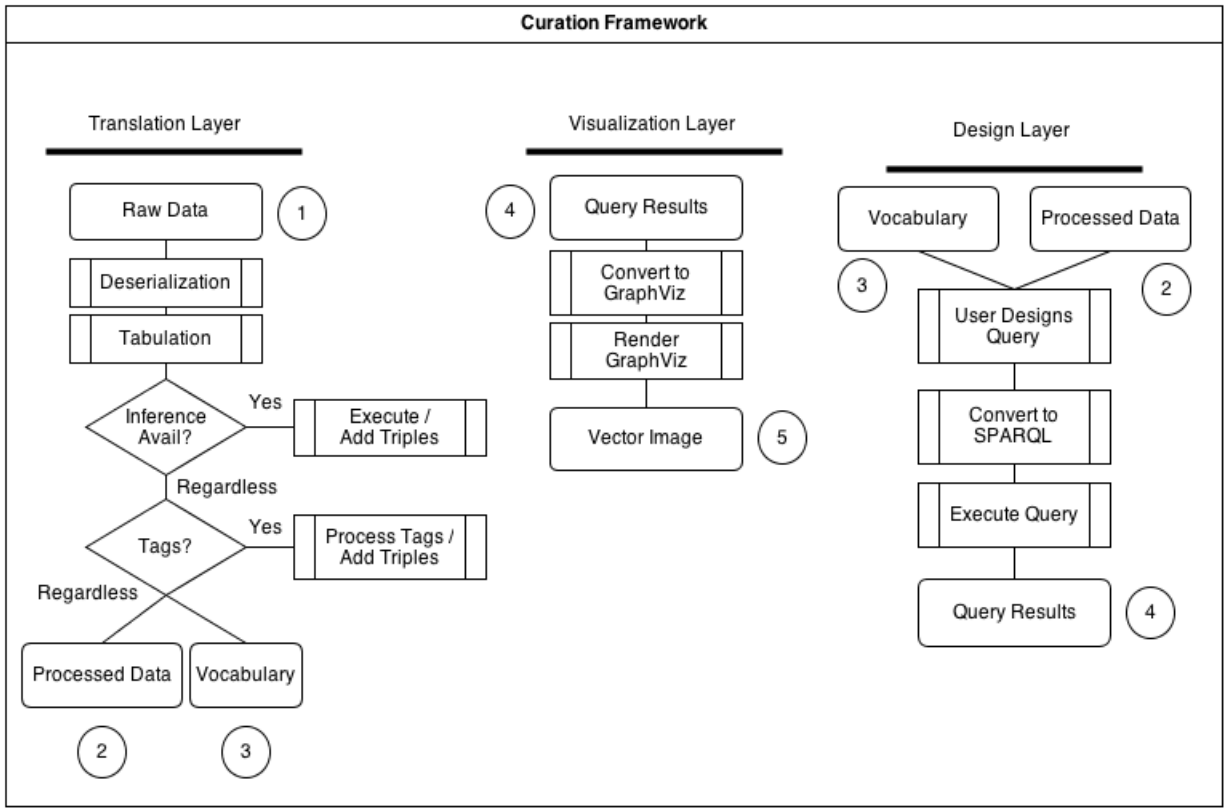

Figure 1: High-Level Architecture Diagram

The Marc4j library was used to deserialize MARC databases for the library database use case. Java's JAXP DOM was used for deserializing XML.

These frameworks were chosen due to previous experience with Web Applications, PHP, and ARC2, as well as the various graphical toolkits often used in these applications. We also evaluated the viability of using various different RDF libraries for different languages, such as the Redland RDF library for C, but Java was chosen because it integrated well with the various libraries which would later be used for mapping different types of data into RDF. Namely, the HDF parser library for Java is very mature, and there are also connectors for using CSV, XLS, MARC, and other high level format parsers. Not only reducing the amount of parsing implementation which was available, Apache Jena took care of the serialization and the deserialization of RDF, both in RDF/XML and RDF/N3. N3 is essentially the Turtle format, and is used throughout information about this project as it is much more human-readable than RDF/XML.

3.1.2 Deepening Mappings

During work with the hyperspectral imaging use case, as well as merging the hyperspectral and bathymetric data sets together through the enterprise data use case, we realized that the massive flat data structures provided were just about as confusing as data subsumed into a relational database schema. Therefore, we wanted ways of making links between different key points of the transitory mappings. Therefore, we would not have to sacrifice the mapping but would also gain a cross-transitory-schema method of querying for basic derived information.

We still experimented with these transitory mappings for some time, as they were much more powerful than working with the data outside of different vocabularies. We integrated many vocabularies, both pre-existing and not, in order to make different sets of data more compatible with each other. However, for example, in the Hyperspectral use case we decided that we were doing far too much text manipulation with regular expressions in our queries. We were being limited with the flat data model, so we wrote several

mapping queries which translated the flat data model into something with a more hierarchical structure. To achieve this, we mainly utilized categorization tags. These categorization tags proved to be so invaluable to building queries, we implemented them in a generic way that could be specified through the use of a very simple tabular text file. This enabled the potential of deepening any other data sources through using sets of text files containing these mapping tuples.

A simple tagging scheme is incapable of providing more than a flat map of tags. These tags are powerful search tools, but provide little more than plain-text search. Additionally, these tags are a lot less likely to be organized in a reasonable manner, and thusly, making normalization of data far more difficult. This tagging scheme reads strings found in textual data, and according to matches of regular expressions, outputs an estimated tag set for any data object. For some types of highly specified data formats, this tagging will be a nearly perfect match, and for others which involve highly variable notations, there will be a lot of visible mistakes. The only current solution we have discovered for this problem is to have mappings for individual data sets.

3.2 The Query Builder Component

The most significant and novel component of our framework is the visual query builder software itself, shown in Figure (2). It was initially developed in JavaScript and PHP, using the user's choice of database with the ARC2 RDF database interface library. Later, the frontend remained in JavaScript but the backend was migrated to an Apache Jena Servlet, as ARC2 was never updated and implementing a SPARQL 1.1 parser in PHP was determined to be much more difficult than simply replacing the PHP glue code. The query builder allows the construction of most types of desired queries. Support for execution and viewing the results of such queries is supported, however, it was not designed to execute the complex translational queries as seen in the mapping component of the framework. This is mainly due to the extreme data volume of such translations, and the design of this component is focused on availability as a web application for quick online operations. Queries used for data translation may be exported from the query builder and run externally, howerver.

The query builder works by providing the user three panels to work in. The first panel is the vocabulary tree, which shows the user the tree view of a given vocabulary and allows him or her to select Classes and Properties. This panel was first suggested while working on the Employee Database use case, as there are many different classes and relations, and a user must not be expected to manually input this level of data.

The second panel is an Instance Viewer, which provides a listing of a given set of instances for a Class. This panel makes it easy to select, for example, a given job position in a database, as it can be selected from the Instances panel.

The third panel is the query tree itself. This tree is the most complex part of the query builder, and is designed to still remain easy to read for most normal users who have some level of skill in the given field. The query tree supports basic graph patterns, unions, and comparisons. It also supports as a literal value anything which can be retrieved through either the first or the second panes, and it does support variables for unknown values to be returned by either a select or a construct query.

3.3 The Visualization Component

Visualization for the query results, subsets of databases, or even queries themselves can be quite difficult to generate due to the very large number of possible nodes and edges. However, we decided that the best possible visualization would be a very small subset of edges and nodes, when it comes to subsets of databases. As far as displaying query results, we attempted to tabularize the outputs to some extent, which makes viewing these particular results somewhat tolerable.

The use of this visualization component over large sets of data, such as entire graphs from one run of the Hyperspectral translation, proved impractical. We generated vector graphics from the RDF data, however there were so many nodes and so many connections between nodes that viewing in an image viewer was not a realistic option. After attempting to format the graphics to print on a large plotting machine, we determined that the space required for such a visualization (not to mention the paper and toner used!) would be crazy. Therefore, we reconsidered visualizing such large databases in this manner. This type of

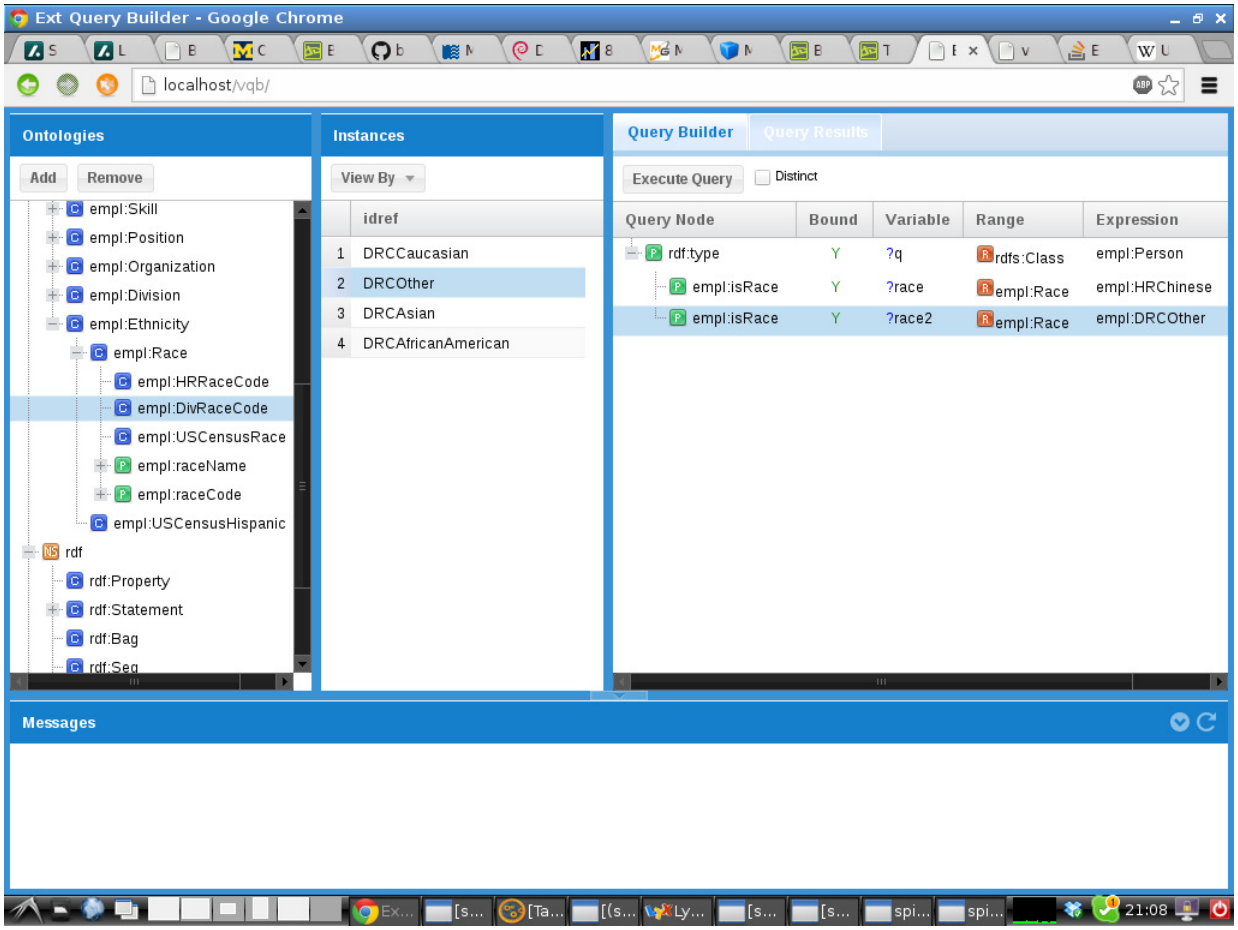

Figure 2: Visual Query Builder Overview

data is more easily and realistically viewed through linked data applications and interactive viewers which only display a small subset of the graph at any given time. Likewise, a query engine also provides a way of cutting back on the amount of data viewed at any given time.

For these reasons, automatic visualization wasn't useful unless they could be generated from the actual query results. Fortunately, this only required a basic implementation of the construct query format in the query builder. This was necessary anyway, as this lacked query type was also required to be used in the hyperspectral use case for presenting the results of the use cases in a textual, hierarchical form.

Probably the simplest component to work with, as we have already generated the query at this stage, as well as executed it and retrieved results. To display the query results on the screen, first we use the Redland RDF tool "rapper" to convert the XML/RDF into *dot* notation. Then, a series of transformations are applied to the data for more readable graphs, if the query designer utilized visualization labeling. Finally, the *dot* graphviz application can be used to draw an undirected graph, which is essentially a more human readable view of the textual SPARQL results.

The visualization component provides a benefit to most users which are not familiar with the SPARQL notation, and that is that they do not need to learn the graph language in order to read the resulting graph.

4 Use Cases, Use Case Realizations, and Implementations

We have worked on a number of use cases in the process of testing our prototype framework. Most of them were developed in-house. We did get expert input for some of the test cases, which greatly improved our ability to debug user issues with the framework and fill in the feature gaps.

The use cases were developed in a variety of ways. The first use case, namely the Employee Database use case, was generated manually via the RDF database editor tool Protege. The Social Networking Database was generated directly from social networking websites' own native RDF/XML providers, and the examined subsets of vocabularies are thus very limited (namely, to FOAF and very little else). The Structured Descriptive Data use case was handled by a automatic conversion process in Java which converted SDD/XML into RDF. With the Library Database, we utilized a similar conversion process with a library designed specifically for MARC catalog handling. Likewise with the Hyperspectral Database, except the source of the data was HDF. The Bathymetry Data use case was produced via Taverna Workbench and manually converting the process model of the original scientists' (likely using Matlab, though it is undetermined) into a new process model which has provenance tracking information. The Enterprise Data use case was produced by combining the Bathymetry, Hyperspectral, and a manually generated database in Excel Spreadsheet format. These three components were used as the basis of the use case.

These test cases were designed to simulate real-world tasks, and demonstrate how much easier they can be achieved than through more traditional database systems. We limited our consideration of scalability to moderately scaled systems, leaving out the possibility of a distributed database system. Nevertheless, it should be quite easy to install a more decentralized database system into the framework as long as it has a SPARQL endpoint. Unsurprisingly, virtually all scalable database backends will have an endpoint available.

In each of these test cases we will also describe several sample databases, in which the Visual Query Builder software will be used in an environment similar to one which would be found in a corresponding real-life organization. Despite being designed for students which are not experienced in a particular field of endeavor, this data set is intended to resemble realistic data as much as possible. The databases which follow include things as disparate as enterprise data to hyperspectral imagery. One single factor unites all of them, the fact that each of the databases are within a single field of endeavor, and are intended on allowing researchers to more easily work together by allowing semantic structures to form naturally.

4.1 Employee Database

This database demonstrates a business application used to solve the issue of resolving conflicting data without excessive manual intervention. This use case implements an employee database which can contain a very large variety of information on their employees. These pieces of data can include, but are not limited to, the following list.

Algorithm 4 Visualization Labeling Pipeline

```
# Get all properties as strings (P1)
PREFIX viz: <http://muckdragon.info/visualization/viz#>
PREFIX fn: <http://www.w3.org/2005/xpath-functions#>

CONSTRUCT {
        ?subject viz:property ?property
        ?property viz:proptext ?proptext
} WHERE {
        ?subject viz:property ?property
        ?property viz:name ?propname
        ?property viz:value ?propvalue
        BIND(fn:concat("<B>", ?propname, "</B>:␣", ?propvalue) AS ?proptext)
}

# Get list of all properties as a property string (P2)
PREFIX viz: <http://muckdragon.info/visualization/viz#>
PREFIX fn: <http://www.w3.org/2005/xpath-functions#>
PREFIX afn: <http://jena.hpl.hp.com/ARQ/function#>

CONSTRUCT {
        ?subject viz:propstring ?propstring
} WHERE {
        ?subject viz:property ?property
        ?property viz:proptext ?proptext
        BIND(fn:concat("{", afn:strjoin("|", ?proptext), "}") AS ?propstring)
}

# Get graphviz final output (P3)
CONSTRUCT {
        ?propstring1 ?predicate ?propstring2
} WHERE {
        ?subject ?predicate ?object
        ?subject viz:propstring ?propstring1
        ?object viz:propstring ?propstring2
}
```

Query Node	*Bound*	*Variable*	*Range*	*Filter Expression*
rdf:type	N	?q	rdfs:Class	empl:Employee
- empl:inDivision	N			empl:Division1
- empl:isEthnicity	N		empl:Ethnicity	empl:NotHispanicLatinoOrSpanishOrigin
- empl:emplId	Y	?id	xsd:int	

Table 1: Query Builder: List all Employees of a Specific Ethnicity

- General contact information
  - Name, address, telephone number, email, etc.
- Worker type
  - e.g. Contract, Salaried
- Citizenship status
- Race and ethnicity
- Disability status
- Authority hierarchy
- Skills and specializations
- Education
- Currently active projects
- Payroll
- Performance reports

4.1.1 List all Employees of a Specific Ethnicity

An enterprise database administrator is working with a database for a large, multidepartmental corporation, which has multiple databases for each department. Many of these databases are updated independently, and thus are never synchronized with each other. This lead to a major issue where some of the departments have created an entirely different schema for representation of certain data elements. For example, there are multiple notations for race and ethnicity provided. The organization has now decided to consolidate many departmental databases into a centralized schema, with the help of our software. This would reduce the workload on database administration.

In this initial demonstration, we want to list all of the employees of a specified ethnicity, under a single division. Here we only want to display the employee ids. This demonstrates the use of a simple query to retrieve a small subset of data. A diagram of the query which can be used in the query builder is in Table (1). Note that the Range and Filter Expression are populated through selection in the user interface from the left and middle panes, the Bound flag is a single yes or no checkbox, and the Variable is set by the user of the query builder. This allows the user to focus on the design of the query rather than the syntax of SPARQL.

4.1.2 Get Metadata Information for Race and Ethnicity

There was initially metadata attached to the race and ethnicity fields, and the database administrator wants to review them to compare this data with the original classifications in the SQL database. For this, he devises a query to show all of the race and ethnicity classes, so that he can do likewise on the SQL database and make a differential comparison. In such test, he discovers that the only difference are the descriptions of

Query Node	*Bound*	*Variable*	*Range*	*Filter Expression*
UNION	N			
rdf:type	Y	?q	rdfs:Class	empl:USCensusRace
- rdfs:comment	Y	?comment	rdfs:Literal	
rdf:type	Y	?q	rdfs:Class	empl:USCensusHispanic
- rdfs:comment	Y	?comment	rdfs:Literal	

Table 2: Query Builder: Get Metadata for Race and Ethnicity

Query Node	*Bound*	*Variable*	*Range*	*Filter Expression*
rdf:type	Y	?q	rdfs:Class	empl:Employee
- empl:hasSkill	Y	?skill	empl:Skill	
- - empl:subSkillOf	N		empl:Skill	empl:Electrician
- empl:emplId	Y	?id	xsd:int	
- empl:first	Y	?first	xsd:string	
- empl:last	Y	?last	xsd:string	

Table 3: Query Builder: List Employees with a Given Skill or Skill Set

the races and ethnicities which were automatically gleaned from Census classifications rather than manually entered.

List all of the races and ethnicities and display their type and detailed comment in the table. This demonstrates use of rdf:* ontology properties and the use of queries with UNIONs. The diagram of the query which was used in the query builder is in Table (2).

4.1.3 List Employees with a Given Skill or Skill Set

The enterprise DBA wants to see how linked data can potentially be a better solution than traditional SQL databases, and is not convinced that the worker skills system is sufficiently better to warrant replacing the entire system. He wanted to be able to keep track of a list of skill hierarchies, so that individuals skilled in a field could have their specializations (generated either through educational transcripts or field experience working for the company) recorded in the database using a class-based inheritance.

This use case demonstrates building a query, which has multiple levels in the query builder tree. The Employees ontology has been extended with information regarding the skill sets of the different employees, so that a member with the proper qualifications can be found for a specific job. In this query, we will search for employees who are qualified as electricians, and we will be able to see their specializations if they have any.

The query description in English is: "List all of the employees who have the Electrician skill. From the list, display their employee id, first and last names, and the specialization of the aforementioned skill. This demonstrates the use of a query where instances have a hierarchy." A diagram of the query in the query builder is in Table (3).

4.1.4 Resolving Data Conflicts in Migration

The organization initially desired a way to keep track of what employees were in what division. This was automatically generated from the individual division's database stores. However, many times an employee who was not terminated but simply moved to another division wasn't removed from a given division's roster, and so still appears in the listing. Therefore, a simple combination of all data stores is not sufficient, and they needed to determine which persons were assigned to more than one division.

The query description in English is: "List all employees who match more than one Division. From the list, display their employee id, first and last names, and each division. This sort of query demonstrates the ability to discover misreported or duplicate data." A representation in the query builder is in Table (4).

Query Node	*Bound*	*Variable*	*Range*	*Expression*
rdf:type	N	?q	rdfs:Class	empl:Employee
- empl:emplId	Y	?id	xsd:int	
- empl:inDivision	Y	?div		
- empl:inDivision	N			(?div2 != ?div)
- empl:firstName	Y	?first	xsd:string	
- empl:lastName	Y	?last	xsd:string	

Table 4: Query Builder: Resolving Data Conflicts in Migration

4.1.5 Detecting Potential Race and Ethnicity Data Conflicts

In the previous example we mentioned data conflicts arising from the collision between multiple divisions. However, the public relations team has noted that they appear to have conflicting race and ethnicity data, which could become an issue for governmental regulation on employment quotas. The database administrator was not sure what was exactly meant by "conflicting" in this case, only that the race and the ethnicity combination were not valid for an individual to possess, such as a Filipino who is not Hispanic. In English: "List all of the valid races for an ethnicity, according to the employees who are currently in the database. From the list, display the race and the linked ethnicity." This type of query demonstrates the avoidance of repeated data with the DISTINCT option, and may be used to find outlying occurrences where some data was misreported. Table (5) presents this query and its results. As you may notice, there is a small result set due to the pruning of duplicate entries.

4.2 Social Networking

Social networking sites, as of recently, have begun releasing a lot of their basic information via RDF/XML as well as other web standards. For this use case, we focused on networks which utilized RDF/XML, rather than other standards such as RSS.

This example demonstrates the usage of data aggregation techniques applied to RDF/XML gleaned from social networking sites such as Facebook and Livejournal which offer the data in a standard semantic format, using such vocabularies as FOAF and DC.

It is possible to make an aggregate data set with other sources, such as DBPedia, which also offers RDF/XML information derived from Wikipedia's "sidebar" style annotations. Geographic information, notable people, statistical information, scientific concepts, and more are all available in a format which is more easily inserted into dynamic query building software such as the aforementioned framework.

This information can be used to infer more information based on location, place of employment, friends' interests, etc. However, some of these tactics will be required simply to resolve conflicting data, such as usernames which are different on different social networking sites as well as usernames which are the same yet refer to completely different people altogether.

4.2.1 Displaying a Person's (Public) Contacts for a Given Site

A social networking site company has decided that, with the advent of RDF for some other social networking sites, that it would be advantageous to experiment with the retrieval of data from this network to determine possible links between individuals. They wanted to include this type of data in their own system, in order to make better guesses between whether two accounts are friends on another network. This would allow them to make better suggestions to the users of potential friends. The company has tasked one data mining expert to create a query which would retrieve a graph consisting of links between individual accounts.

It is possible to traverse a social networking site using solely linked data, to get a person's direct and indirect contacts. For this, we will want to specify a maximum depth to search by chaining queries, as we will not want an entire site database contacts. For each contact, there may be some basic contact information, limited to email addresses, web pages, and instant messenger names. However, in this use case we would like to show only names to reduce visual intensity.

Options: DISTINCT

Query Node	*Bound*	*Variable*	*Range*	*Filter Expression*
rdf:type	N	?q	rdfs:Class	empl:Employee
- empl:isEthnicity	Y	?ethn	empl:Ethnicity	
- empl:isRace	Y	?race	empl:Race	

Race	*Ethnicity*
PuertoRican	White
PuertoRican	BlackAfricanAmerican
OtherHispanicLatinoOrSpanishOrigin	White
OtherHispanicLatinoOrSpanishOrigin	Filipino
NotHispanicLatinoOrSpanishOrigin	Japanese
NotHispanicLatinoOrSpanishOrigin	White
NotHispanicLatinoOrSpanishOrigin	Samoan
NotHispanicLatinoOrSpanishOrigin	BlackAfricanAm
NotHispanicLatinoOrSpanishOrigin	AmericanIndianOrAlaskaNative
NotHispanicLatinoOrSpanishOrigin	OtherAsian
NotHispanicLatinoOrSpanishOrigin	Chinese
NotHispanicLatinoOrSpanishOrigin	Vietnamese
NotHispanicLatinoOrSpanishOrigin	NativeHawaiian
NotHispanicLatinoOrSpanishOrigin	GuamanianOrChamorro
NotHispanicLatinoOrSpanishOrigin	AsianIndian
NotHispanicLatinoOrSpanishOrigin	Filipino
NotHispanicLatinoOrSpanishOrigin	Korean
NotHispanicLatinoOrSpanishOrigin	White

Table 5: Query Builder: Detecting Race/Ethnicity Data Conflicts

Algorithm 5 SPARQL Query: List User Contacts

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX mydata: <http://muckdragon.info/foaf/names#>

CONSTRUCT
{
        ?ljfoaf foaf:nick "arven"
        ?ljfoaf foaf:knows ?level1
        ?level1 foaf:nick ?name1
        ?level1 foaf:knows ?level2
        ?level2 foaf:nick ?name2
} WHERE
{
        ?ljfoaf foaf:nick "arven"
        ?ljfoaf foaf:knows ?level1
        ?level1 foaf:nick ?name1
        ?level1 foaf:knows ?level2
        ?level2 foaf:nick ?name2
}
```

Query Node	*Bound*	*Variable*	*Range*	*Filter Expression*
CONSTRUCT				
foaf:nick	Y	?ljfoaf	xsd:string	"arven"
- foaf:knows	Y	?level1	foaf:Person	
- - foaf:nick	Y	?name1	xsd:string	
- - foaf:knows	Y	?level2	foaf:Person	
- - - foaf:nick	Y	?name2	xsd:string	
WHERE				
foaf:nick	Y	?ljfoaf	xsd:string	"arven"
- foaf:knows	Y	?level1	foaf:Person	
- - foaf:nick	Y	?name1	xsd:string	
- - foaf:knows	Y	?level2	foaf:Person	
- - - foaf:nick	Y	?name2	xsd:string	

Table 6: Query Builder: List User Contacts

Algorithm 6 Command Line: Display Contact Map

```
#!/bin/sh
rapper -i turtle -o dot names.rdf > names.dot
dot names.dot -Tpdf -o names.pdf
```

4.2.2 Displaying a Logical Map of Contacts

The developer in charge of utilizing the framework for their social networking site was tasked with providing a visual representation of what the query he developed in the previous example was doing. They wanted a way to visualize the connections between friends, and provide this sort of visualization ability to their users. This would provide a user-interface feature that was sorely lacking in their platform, and potentially draw more visitors.

Graphical 'mind maps' have gained a lot of popularity on the web over the years, as they are powerful tools for reviewing linked structures. These are very closely related to undirected graphs. This use case demonstrates the display of a 'map' of one particular person's contacts, as in the first use case. Undirected edges link the contacts together, emphasizing the reflexive nature of the relationships seen in the FOAF vocabulary. The output of this use case is a formatted text file which can be read and displayed by the Unix directed graph visualization utility known as "dot."

4.2.3 Lessons Learned in Visualization Component

This was the first use case in which a useful visualization was developed using the Unix utility dot (including in the popular GraphViz package). One may note our use of a specialized VIZ vocabulary that seems to serve no purpose other than to make the output results more complex. On the contrary, we use a vocabulary that has no meaning in RDF, that is processed in a specialized pipeline described in Algorithm (4). After this processing, a form of RDF suitable for viewing but unsuitable for data manipulation is generated. This serves to remove some of the indirection provided by RDF, to provide a more viewer-friendly representation.

4.3 Library Database Aggregation

A group of public libraries has a set of online card catalog databases that they are trying to make more useful to potential users, by linking data with other library databases. The main purpose of this effort was to make library users able to find a resource regardless of whether they had the resource in stock at the current time. They wanted to allow users to look up titles and authors just as they had originally, with the additional feature that they could track where a particular book was available if known.

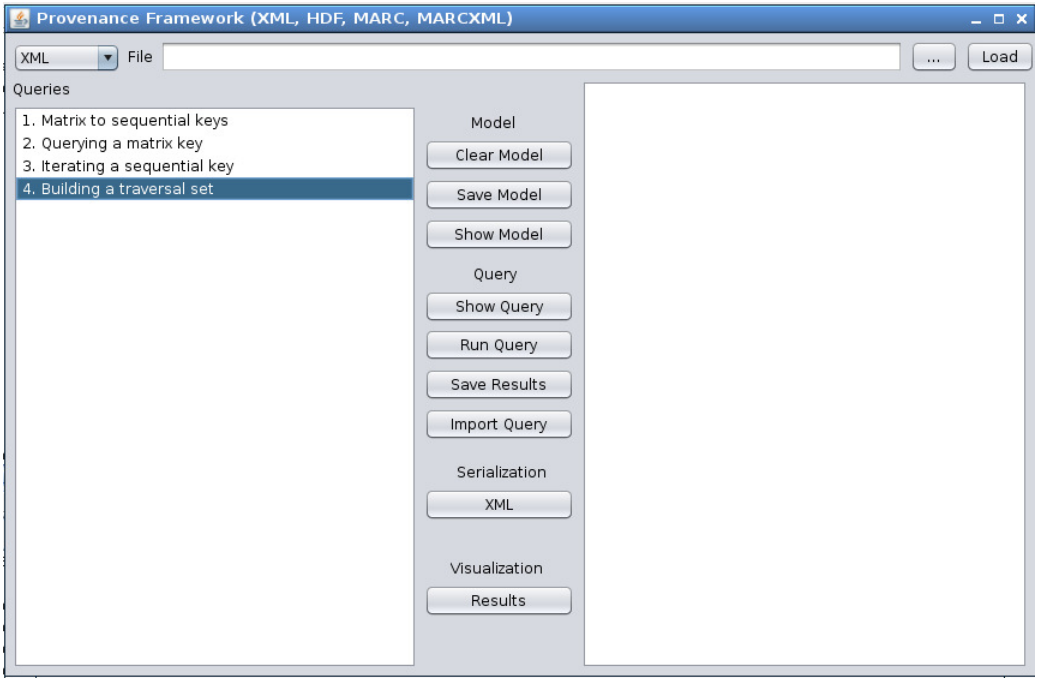

Figure 3: Unified Conversion Software

Many libraries have settled on an ISO-2709 based data format for the storage and retrieval of their bibliographic information called MARC21 (binary representation) and MARCXML (XML representation), developed by the Library of Congress. This is a relatively arcane schema, which uses data fields annotated with tags, indices, and codes. These codes are not obvious without reading the actual MARC schema specification. However, unlike we noticed in the first major use case we implemented, the employee database, these records are an actual standard and should not require any normalization of data before converting into RDF. The database that is returned will be hierarchical, and the vocabulary used will be simple and mostly 'flat'.

4.3.1 Generating Materialized RDF View from XML Documents

This first use case demonstrates the power of the Marc to RDF conversion software, which transforms Marc metadata information into RDF. Initially this conversion is done using a predefined mapping table, which presents a hierarchical database (as the input binary does) using a flat vocabulary model. We are using a Marc4j Java API with the Framework we have developed for conversion (Fig. 3) derived from the Hyperspectral use case. However, this is a far simpler case, intended as a proof of concept that this framework is scalable outward to other fields. We used the Open Access Bibliographic Records created by the University of Michigan totaling 684,597 records. This is a large data set, unlike the earlier Hyperspectral use case which was a small subset of one scientist's work. As the database is licensed under the Creative Commons Zero license, we are able to readily work with the large data set without the hassle of getting permission to use each data set. This way we are more readily able to test the scalability of our framework to large databases. Likewise, as it is a large data set that is pre-normalized, we may eliminate the step of normalization of data and can simply allow the user to provide queries to the query engine directly without inferencing.

4.3.2 Output Bibliographic Listings from a List of Titles

A student interested in several titles relating to archaeology has found some interesting books about the topic on the Internet, but needs to find where they can be located. He goes to his library, and discovers they have implemented a new card catalog system. It is a simplified version of the original query builder, to

Query Node	*Variable*	*Range*	*Expression*
UNION			
rdf:type			bibliographic:Record
- bibliographic:hasDataField			
- - bibliographic:hasDataFieldTag			"245"
- - bibliographic:hasSubField			
- - - bibliographic:hasCode		xsd:string	
- - - bibliographic:hasData	?title		contains("these ruins are inhabited")
- bibiographic:hasDataField			
- - bibiographic:hasSubField			
- - - bibliographic:hasCode	?code	xsd:string	
- - - bibliographic:hasData	?value		
rdf:type			bibliographic:Record
- bibliographic:hasDataField			
- - bibliographic:hasDataFieldTag			"245"
- - bibliographic:hasSubField			
- - - bibliographic:hasCode		xsd:string	
- - - bibliographic:hasData	?title		contains("south asian archaeology")
- bibiographic:hasDataField			
- - bibiographic:hasSubField			
- - - bibliographic:hasCode	?code	xsd:string	
- - - bibliographic:hasData	?value		

Table 7: Query Builder: Bibliographic Listings

make multiple queries easier to perform as well as to make it easier to search for common attributes such as *title* and *author*. From here, the student simply selects that he would like to look up titles, and enters the two titles he was interested in.

This use case demonstrates simple querying operation on the generated view. The user will input queries for individual titles of books, unifying the results together with the SPARQL union operator. A query template in the format of the full-featured query builder is shown in Table (7).

4.3.3 Generating a Bibliography for a Given Author or Characteristic

The student in question now discovers that one such book is available from a nearby library, and none are available at his own. He decides to drive there to check the book out quickly, but first he wants to find out what books are available written by a particular author he favors, "Alice Brown". He finds several books, but none particularly catch his interest as he has read all of the ones available from the library.

Another use of this type of data is generating a bibliographical listing for a given author. In this case, no union is needed as the author remains the same. There are many other possibilities, but we use the author as a criterion simply because it is likely to produce an amount of data suitable for human readability. Another query, slightly modified from the first, was described in the query builder. A representation of it can be seen in Table (8).

4.3.4 Post-Mortem: Performance Issues

Unfortunately and unsurprisingly given the nature of the software in question, Jena was of very poor performance when generating the initial model. In an effort to avoid having to rewrite the entire framework for testing this use case, we have limited the size of the queried database to around 50,000 records. Jena's database, like other triple store databases (and classical relational databases), tends to get slower and slower with the more triples that are added. Quite often, a software application invoking Jena's database features will hang waiting on Jena indefinitely. Figure (4) demonstrates the loading of the entire University of Michigan Library's free bibliographical database, where it appears to stall at loading 96000 records on a

Query Node	*Variable*	*Range*	*Expression*
rdf:type			bibliographic:Record
- bibliographic:hasDataField			
- - bibliographic:hasDataFieldTag			"100"
- - bibliographic:hasSubField			
- - - bibliographic:hasCode		xsd:string	"a"
- - - bibliographic:hasData	?authorl		contains("Brown")
- - - bibliographic:hasData	?authorf		contains("Alice")
- bibiographic:hasDataField			
- - bibiographic:hasSubField			
- - - bibliographic:hasCode	?code	xsd:string	
- - - bibliographic:hasData	?value		

Table 8: Query Builder: Bibliographic Listings

system with 16GB of RAM and a AMD Phenom II X4 955c. There are configuration options which would likely speed up Jena slightly, but for the most part an entirely different triple store infrastructure will be needed. For a more performant and scalable infrastructure, a database system which can handle distributed computing such as OpenLink Virtuoso should be used.

4.4 Ocean Bathymetry

Bathymetry is the study of the depths of ocean floors and other bodies of water, equivalent to topography of land. Through work with the NRL, I learned about the format of actual data real scientists produced. Data relating to bathymetry and other geographic data is very contextual. It is important to know not only what location some data was gathered on, but at what time and date as well as the exact type of data one is dealing with. Data may be in gridded form or it may be a series of points. Normally, this data is stored in a text sequence of floating point values. It may not always be obvious whether data is in gridded form or a series of points, so some level of detection and conversion is necessary. In this use case, we already knew the type of data we were working with as well as when and where the data was collected.

4.4.1 Create a Provenance Workflow for DeLaunay Triangulation

Provenance is the tracking of how events are related to data. These events occur at a specific time and end at a specific time. Events can be started by people, living things, environmental processes, and computer programs. Events can happen at a particular place, be triggered by another event, start other events unfolding, etc. There are many possibilities for describing cause and effect relationships with provenance.

We initially developed a Provenance workflow(5) for a simple task dealing with Bathymetric data, namely parsing a set of data and running DeLaunay Triangulation on it. This data resulted in a three-dimensional colored mesh with vertex coloring. We developed the described Workflow in Taverna Workbench, a Provenance-aware process workflow editor. This works in much the same way as a BPEL (Business Process Execution Language) or BPM (Business Process Management) workflow. The output of Taverna was a flat file representation as is used by the research scientists in this field, as well as a metadata file containing RDF resources.

4.4.2 Data Availability by Date: Bathymetry

Research scientists were interested in converting some of the bathymetry data with attached color heights into three dimensional representations for a simulation. They wanted to find a large area which was scanned fairly recently, as they wanted an interesting simulation that had a good chance of being notable.

The bathymetry data set available to us had some basic temporal information attached, which was appended to the process provenance returned by Taverna. Having this provenance information allows researchers to reuse data which may not even be able to be collected again given certain conditions (e.g.

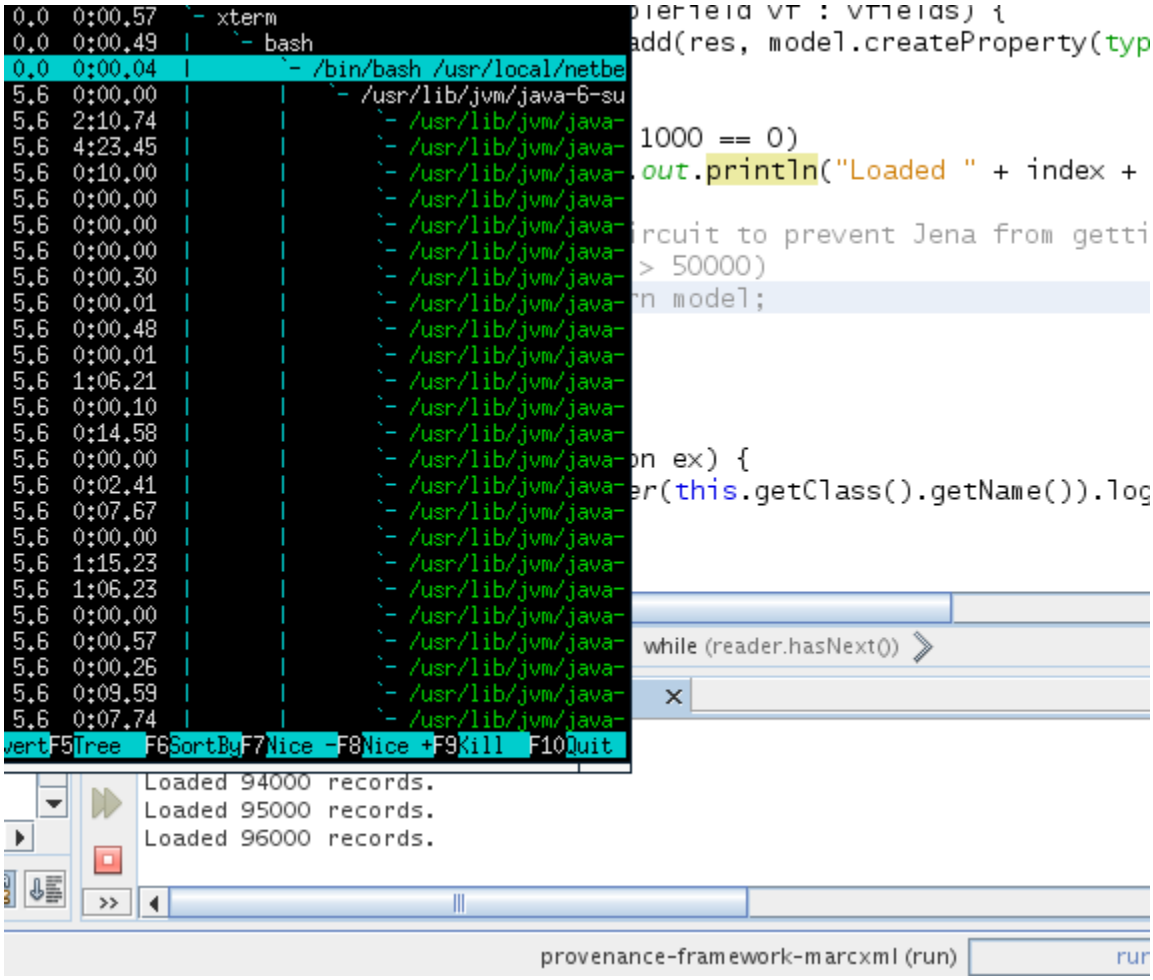

Figure 4: Jena Database Poor Performance

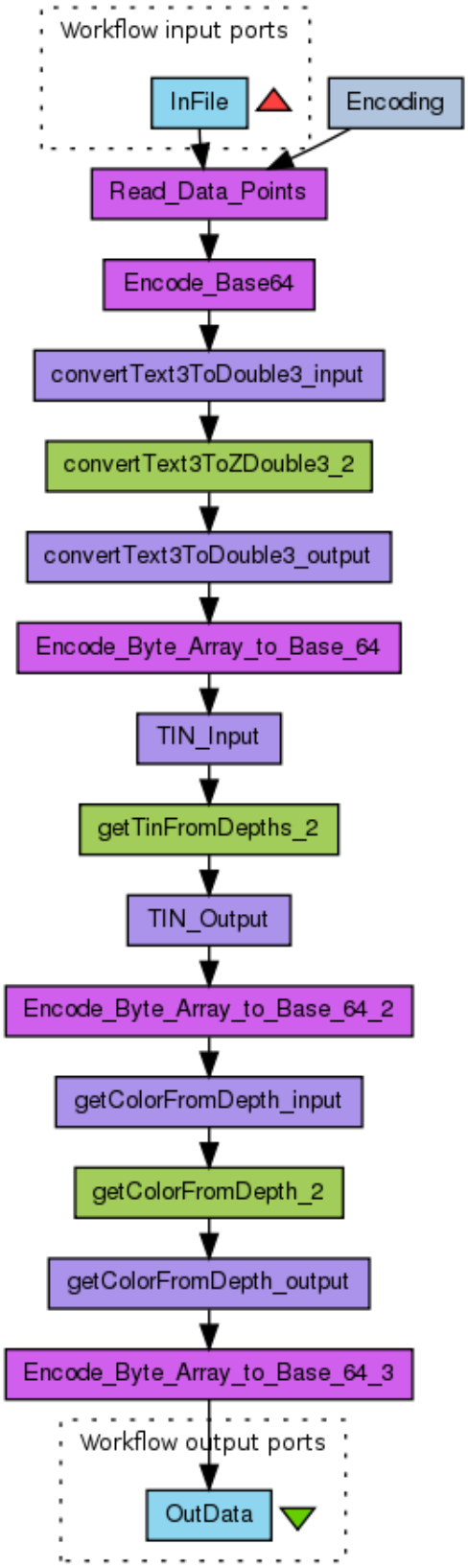

Figure 5: Taverna Workflow for DeLaunay Triangulation

Query Node	*Variable*	*Range*	*Expression*
CONSTRUCT			
category:hasCategoryTag	?q		category:bathymetry
- prov:startedAtTime	?start		
- prov:endedAtTime	?end		
WHERE			
category:hasCategoryTag	?q		category:bathymetry
- measure:hasMeasurementTag			measure:depth
- prov:startedAtTime	?start		
- prov:endedAtTime	?end		

Table 9: Query Builder: Bathymetry Data Timestamps

Query Node	*Variable*	*Range*	*Expression*
CONSTRUCT			
category:hasCategoryTag	?q		category:bathymetry
- space:hasBoundingBox	?bbox		@wgs84:bboxintersect(N1,E1,N2,E2)
WHERE			
category:hasCategoryTag	?q		category:bathymetry
- measure:hasMeasurementTag			measure:depth
- space:hasBoundingBox	?bbox		@wgs84:bboxintersect(N1,E1,N2,E2)

Table 10: Query Builder: Bathymetry Data Timestamps

weather, major geological and oceanographic events, etc.) The cost savings due to not having to collect data again for testing new methods or models would also be very beneficial to the research scientist.

In this example, we have designed a graph query in the visual query builder, which is shown in Table (9).

4.4.3 Data Availability by Place: Bathymetry

The research scientists were not pleased with their simulation, nor was it really visually appealing to anyone not in their field, as all the data was mostly flat. They wanted to be able to show some real underwater terrain and some very deep extremes, so they decided to do a query for some data within a large range containing some relatively interesting terrain. They got back several results, and chose the data set which appeared to have the most accuracy.

Most types of sensor data is useless without the actual location of a study, and the bathymetry data lacked any sort of internal metadata including location information. Therefore, this exercise shows the ease of attaching geographic locations to existing unannotated data. However, once the data is annotated in this manner, it becomes possible to associate bathymetry data with other data sets which may be related. With a much larger bathymetry data set, one could correlate various data sets using computer vision algorithms, if only some contain metadata.

In this example, we have designed a graph query in the visual query builder, shown in Table (10). The @wgs84:bboxintersect expression is an alias to a predefined description of a bounding box in wgs84 format, and returns all partially and fully intersecting bounding boxes.

4.5 Hyperspectral Imagery

Hyperspectral imagery refers to images taken by imaging equipment designed to capture a much wider range of light than the visible spectrum. Also working with the NRL, I got another sample of what real data from research was like there. There was some level of metadata. While the metadata was certainly

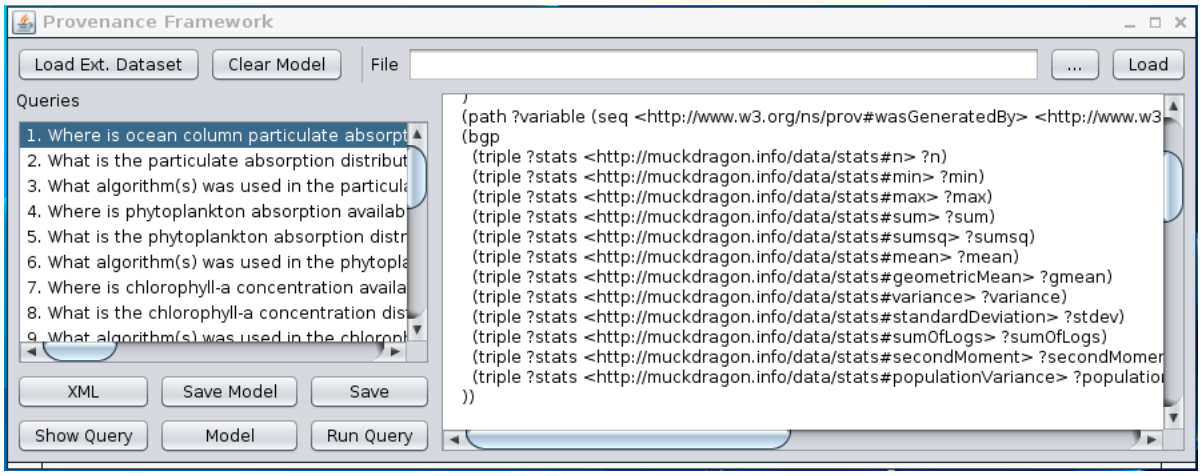

Figure 6: Hyperspectral HDF Metadata

insufficient for querying a vast range of data sets, it was possible to normalize the data to some extent which would make it far more accessible to researchers in the future.

The hyperspectral data, stored in HDF, contains a large amount of metadata which must be transformed into RDF vocabularies. These vocabularies handle data statistics, creating categorization tags, process names and algorithms, as well as data provenance information about the individual data elements. One such vocabulary, which associates variables with product names and algorithms used to produce said data, was generated from an APS product list in Excel format.

4.5.1 Conversion of Hyperspectral Data into RDF Resources

The NRL was interested in curating their enormous data set which was mostly relegated to storage rooms full of external media. Individual researchers had their own data, and this research was not shared well between departments or within a single department well at all. They were interested in reducing research costs by removing the need to duplicate the data acquisition process time and time again. Therefore, they considered a way of extracting metadata and linking it to be a potential solution to their problem.

Initially, we converted the Hyperspectral Data into RDF Resources with the help of a custom piece of Java software(6) which utilized the Java-HDF and Jena Libraries, and is capable of emitting a desired serialization of the formats RDF/XML or the Turtle subset of Notation3 (N3). The software also contained a test framework consisting of numerous different queries, many of which are described in this section along with their related graphical models.

To use the provenance framework conversion utility, one first selects an HDF file (...) to be processed into RDF, and then clicks the Load button. The external datasets are for the purposes of further use cases tying in with the hyperspectral case. The Clear Model button clears the current Jena database for processing of a different file. The Xml button toggles the behavior of the emitter to show either RDF/XML or Turtle. The Save Model button sends the model data to a file. The Save button simply sends the results of a query to a file. The Show Query button shows the currently selected query in the rightmost text area. The Model button shows the current model in the same area, though this is not recommended for very large data sets. Fortunately, the Hyperspectral data set is a single HDF file worth of metadata, so it is not overwhelming. The purpose of the Load External Dataset functionality will be described in the later Enterprise Data use case.

4.5.2 Ocean Column Particulate Absorption

For this use case, we looked at the ocean column particulate absorption locations. We create a bounding box as a search expression, set the measurement category value to "absorption" and the substance category value to "particulate". Then we return the location data via a SPARQL Construct query. This

Algorithm 7 SPARQL Query: Locations for Ocean Column Particulate Absorption

```
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX prov:  <http://www.w3.org/ns/prov#>
PREFIX proc:  <http://muckdragon.info/data/processing#>
PREFIX sets:  <http://muckdragon.info/data/sets#>
PREFIX space: <http://muckdragon.info/data/space#>
PREFIX wgs84: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX measure: <http://muckdragon.info/hyperspectral/tags/measure#>
PREFIX substance: <http://muckdragon.info/hyperspectral/tags/substance#>
CONSTRUCT {
        ?variable space:hasBoundingBox (
                [wgs84:long ?nw_1; wgs84:lat ?nw_2]
                [wgs84:long ?ne_1; wgs84:lat ?ne_2]
                [wgs84:long ?se_1; wgs84:lat ?se_2]
                [wgs84:long ?sw_1; wgs84:lat ?sw_2] ) .
} WHERE {
        ?variable prov:wasGeneratedBy/prov:used/proc:produces ?product ;
                measure:hasMeasurementTag measure:absorption ;
                substance:hasSubstanceTag substance:particulate ;
                sets:isContainedIn ?file .
        ?file space:hasBoundingBox (
                [wgs84:long ?nw_1; wgs84:lat ?nw_2]
                [wgs84:long ?ne_1; wgs84:lat ?ne_2]
                [wgs84:long ?se_1; wgs84:lat ?se_2]
                [wgs84:long ?sw_1; wgs84:lat ?sw_2] ) .
}
```

query is shown in Algorithm (7).

4.5.3 Particulate Absorption Distribution

In the next use case, we looked at the statistical distribution of particulate absorption, for each given data file that was a measurement of "absorption", with a substance of "particulate". All of the statistical data, which was calculated by the Java HDF translation software, is returned associated with each data file. This query is described in Algorithm (8).

4.5.4 Algorithms in Particulate Absorption

Furthermore, we wanted to examine which algorithms were used to compute the particulate absorption values. In this case we were not interested in the input and output products, but solely the algorithms used to generate the final product of particulate absorption.

4.5.5 Phytoplankton Absorption Location

The phytoplankton absorption query was essentially the particulate absorption query, with the only change being the replacement of the substance "particulate" with "phytoplankton" suggesting that the user is looking for information relating to the measurement of a quantifiable amount of phytoplankton. This query is shown in Algorithm (7), with the changes described in this section.

4.5.6 Phytoplankton Absorption Distribution

In this case, the new researcher desired to know the statistical distribution of phytoplankton in the list of data sets. This is the same query as the particulate absorption distribution query, the only changes

Algorithm 8 SPARQL Query: Particulate Absorption Distribution

```
PREFIX proc: <http://muckdragon.info/data/processing#>
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX stat: <http://muckdragon.info/data/stats#>
PREFIX data: <http://muckdragon.info/data/info#>
PREFIX measure: <http://muckdragon.info/hyperspectral/tags/measure#>
PREFIX substance: <http://muckdragon.info/hyperspectral/tags/substance#>
CONSTRUCT {
        ?stats
                stat:n ?n ;
                stat:min ?min ;
                stat:max ?max ;
                stat:sum ?sum ;
                stat:sumsq ?sumsq ;
                stat:mean ?mean ;
                stat:geometricMean ?gmean ;
                stat:variance ?variance ;
                stat:standardDeviation ?stdev ;
                stat:sumOfLogs ?sumOfLogs ;
                stat:secondMoment ?secondMoment ;
                stat:populationVariance ?populationVariance .
} WHERE {
        ?variable prov:wasGeneratedBy/prov:used/proc:produces ?product ;
                data:hasStats ?stats ;
                measure:hasMeasurementTag measure:absorption ;
                substance:hasSubstanceTag substance:particulate ;
                prov:wasGeneratedBy/prov:used ?algorithm .
        ?stats
                stat:n ?n ;
                stat:min ?min ;
                stat:max ?max ;
                stat:sum ?sum ;
                stat:sumsq ?sumsq ;
                stat:mean ?mean ;
                stat:geometricMean ?gmean ;
                stat:variance ?variance ;
                stat:standardDeviation ?stdev ;
                stat:sumOfLogs ?sumOfLogs ;
                stat:secondMoment ?secondMoment ;
                stat:populationVariance ?populationVariance .
}
```

Algorithm 9 SPARQL Query: Particulate Absorption

```
PREFIX proc: <http://muckdragon.info/data/processing#>
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX measure: <http://muckdragon.info/hyperspectral/tags/measure#>
PREFIX substance: <http://muckdragon.info/hyperspectral/tags/substance#>
CONSTRUCT {
        ?variable prov:wasGeneratedBy ?aname
} WHERE {
        ?variable prov:wasGeneratedBy/prov:used/proc:produces ?product ;
                measure:hasMeasurementTag measure:absorption ;
                substance:hasSubstanceTag substance:particulate ;
                prov:wasGeneratedBy/prov:used ?algorithm .
        ?algorithm proc:name ?aname
}
```

being the replacement of the substance "particulate" with "phytoplankton" suggesting that the researcher is looking for information relating to the substance of phytoplankton with the same measured effects - namely, absorption. This query is described in Algorithm (8), with the aforementioned changes.

4.5.7 Algorithms in Phytoplankton Absorption

The new researcher was interested in finding out what algorithms the original researchers employed in the calculation of phytoplankton absorption, to determine if there were any potential accuracy gains that could be made with using alternative algorithms. The only changes to the algorithm listing for particulate absorption is the replacement of "particulate" with "phytoplankton", as in the previous case. This query is described in Algorithm (9), with the aforementioned changes.

4.5.8 Chlorophyll-a Concentration Location

Suspecting incorrectly applied algorithms, the new researcher examined the locations of chlorophyll-a concentration, which is related to the phytoplankton absorption, in order to compare with overlapping regions of phytoplankton absorption. The researcher changed the particulate absorption location query to have a substance of "chlorophyll-a" and a measurement of "concentration". This query is shown in Algorithm (7), with the changes described in this section.

4.5.9 Chlorophyll-a Concentration Distribution

The new researcher used the chlorophyll-a concentration distribution to determine if it correlates well with the phytoplankton absorption. The researcher changed the particulate absorption distribution query to have a substance of "chlorophyll-a" and a measurement of "concentration", as in the previous query. This query is described in Algorithm (8), with the aforementioned changes.

4.5.10 Algorithms in Chlorophyll-a Concentration

To determine what algorithms are used in computing chlorophyll-a concentration, the researcher repeats a similar query to the one which was utilized for phytoplankton absorption and particulate absorption. The researcher changed the particulate absorption concentration query to have a substance of "chlorophyll-a" and a measurement of "concentration", as in the previous query. This query is described in Algorithm (8), with the aforementioned changes.

4.5.11 Sea Surface Temperatures Location

Interested in sea surface temperature information relating to weather patterns, a scientist locates this data set and performs a query on it for what areas of ocean this particular study gathered sensor data.

The researcher changed the particulate absorption locations query to have no substance qualifier and a measurement of "temperature", as in the previous query. This query is shown in Algorithm (7), with the changes described in this section.

4.5.12 Sea Surface Temperatures Distribution

Having discovered the data to be useful, the researcher looks up the statistical information for the region in order to make proper comparisons with other regions. The researcher changed the particulate absorption distribution query to have no substance qualifier and a measurement of "temperature", as in the previous query. This query is described in Algorithm (8), with the aforementioned changes.

4.5.13 Algorithms in Sea Surface Temperatures

Not familiar with the algorithms utilized in derivation of sea surface temperature via hyperspectral imagery, the scientist used the PROV-O vocabulary's wasGeneratedBy/used triple path and specified that the desired value to be returned was simply the algorithm name. The researcher changed the particulate absorption algorithms query to have no substance qualifier and a measurement of "temperature", as in the previous query. This query is described in Algorithm (8), with the aforementioned changes.

4.5.14 Total Suspended Particles Location

The researcher who was interested in particulate absorption ensures that this data set contains all of the areas which data was calculated for particulate absorption. The researcher changed the particulate absorption location query to have two substance qualifiers of "total" and "particulate", as in the previous query. This query is shown in Algorithm (7), with the changes described in this section.

4.5.15 Distribution of Total Suspended Particles

The researcher desired to find statistical information regarding the number of suspended particles in the given area. The researcher changed the particulate absorption distribution query to have two substance qualifiers of "total" and "particulate", as in the previous query. This query is described in Algorithm (8), with the aforementioned changes.

4.5.16 Algorithms in Total Suspended Particles

To verify that the algorithms used were properly well-established, the researcher also checked the algorithms used in generating this data product. The researcher changed the particulate absorption algorithms query to have two substance qualifiers of "total" and "particulate", as in the previous query. This query is described in Algorithm (9), with the aforementioned changes.

4.5.17 Data Availability by Date: Chlorophyll Concentration

A scientist interested in chlorophyll concentration, suspecting a major decrease in phytoplankton after a chemical spill, decides to query all data sets to determine what date data was recorded on, in order to process statistical information. The query is shown in Algorithm (10).

4.6 Enterprise Data

The enterprise data consists of a superset of the bathymetric and hyperspectral use cases. A project list was presented to me, which was manually entered into an Excel spreadsheet. Each of the projects listed was an accepted proposal from a single researcher who was considered the principal investigator. This investigator was in charge of the project, and project information would be attributed to this person. For each project, we were given a list of attributes. The first attribute, a project identifier, was a pair of two separate codes. The first numerical code was a "research level" of the given project, the second numerical code was the two-digit year in which the project was proposed, and the third numerical code was the index of the research proposal in terms of research level and year.

Algorithm 10 SPARQL Query: Chlorophyll Concentration Date

```
PREFIX proc: <http://muckdragon.info/data/processing#>
PREFIX sets: <http://muckdragon.info/data/sets#>
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX measure: <http://muckdragon.info/hyperspectral/tags/measure#>
PREFIX substance: <http://muckdragon.info/hyperspectral/tags/substance#>
CONSTRUCT {
        ?file prov:startedAtTime ?start;
                        prov:endedAtTime ?end
} WHERE {
        ?variable prov:wasGeneratedBy/prov:used/proc:produces ?product ;
                measure:hasMeasurementTag measure:concentration ;
                substance:hasSubstanceTag substance:chlorophyll ;
                sets:isContainedIn ?file .
        ?file prov:endedAtTime ?end ;
                prov:startedAtTime ?start
}
```

- Project identifier
- Scientist identifier
- Type of data (e.g. temperature, bathymetry)
- Sensors used in experiment
- Purpose of data
- Location where data was acquired
- Title of project

4.6.1 Generate Enterprise Data from Spreadsheet

This enterprise data is relatively useless without a way to connect it with the actual data generated by a given project. For this, it was required to convert the spreadsheet into an RDF database, using an autogenerated vocabulary in the RDF translation software. Due to confidentiality reasons, the numerical indices referring to the given scientists could not be dereferenced into names or other identifying information. Therefore, the generated enterprise data was integrated manually with a small subset of actual project data (namely, the bathymetry and hyperspectral use cases). This small subset of linking information was stored in enterprise-link.ttl, and the real converted enterprise data was stored in enterprise.ttl. An example of the converted data is shown in Table (11).

4.6.2 Merge Enterprise Data Together with Provenance Framework

After we have translated the enterprise data into the RDF format we wish to use, we needed to merge the Enterprise data with the Bathymetry data set as well as the Hyperspectral data set, otherwise we would not be able to make any sensible queries. We could do trivial tasks such as look up titles of projects, years project was achieved, or sensors used in a project, but this would be simple enough to implement in a conventional database. We wished to integrate data, the main goal of the semantic web, and tie the enterprise data into the other two research projects which we had concrete results from.

To this effect, we utilized the same Provenance Framework Application(6) which we also used in the Hyperspectral use case. We simply added a few new queries to test our integration efforts. We also added a Load External Dataset button for this use case. If this Enterprise Data dataset is made available to it, then all of the queries listed in this use case will be available to be tested.

```
<http://muckdragon.info/arv/request/6.1/07/1>
        rdf:type        arv:Data;
        arv:scientist   <http://muckdragon.info/arv/scientist/1>;
        arv:dataType    "Hyperspectral imagery", "IR imagery", "topographic LIDAR",
                "bathymetric LIDAR", "thermal imagery";
        arv:sensor      "hyperspectral", "NASA LIDAR";
        arv:purpose "Understand the hydrodynamic sources of features detectable in
                IR imagery of water surface";
        arv:location "Virginia Coast Reserve";
        arv:title "Joint LIDAR/Hyperspectral Data Collection".
```

Table 11: Conversion Example: Enterprise Data

Algorithm 11 SPARQL Query: Bathymetry Research Enterprise Query

```
PREFIX : <http://muckdragon.info/temp/enterprise-query0001#>
PREFIX proc: <http://muckdragon.info/data/processing#>
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX sets: <http://muckdragon.info/data/sets#>
PREFIX arv:  <http://muckdragon.info/arv/schema/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>
PREFIX measure: <http://muckdragon.info/hyperspectral/tags/measure#>
PREFIX substance: <http://muckdragon.info/hyperspectral/tags/substance#>
PREFIX category: <http://muckdragon.info/hyperspectral/tags/category#>

CONSTRUCT {
    ?s1 :usedSensor ?sens1 .
} WHERE {
        OPTIONAL {
                ?v1 prov:wasAttributedTo ?s1 .
                ?r1 arv:scientist ?s1 .
                ?r1 arv:sensor ?sens1 .
                ?v1 category:hasCategoryTag category:bathymetry .
                ?v1 measure:hasMeasurementTag measure:depth .
        }
}
```

4.6.3 Find Scientists: Bathymetry Research, Sensors Used

A researcher studying bathymetry wants to find any and all bathymetry related research which has previously been conducted, and what sensors were used in the research. Therefore, the researcher queries for the scientists who have been attached via enterprise data to actual projects. The query is shown in Algorithm (11).

4.6.4 List Scientists' Data: Bathymetry

A researcher knows of a particular scientist's work, for instance they have read a publication by them and would like to see what actual data sets they used in their research. This query is shown in Algorithm (12).

Algorithm 12 SPARQL Query: List Bathymetry Data by Scientist

```
PREFIX : <http://muckdragon.info/temp/enterprise-query0001#>
PREFIX proc: <http://muckdragon.info/data/processing#>
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX sets: <http://muckdragon.info/data/sets#>
PREFIX arv:  <http://muckdragon.info/arv/schema/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>
PREFIX measure: <http://muckdragon.info/hyperspectral/tags/measure#>
PREFIX subst: <http://muckdragon.info/hyperspectral/tags/substance#>
PREFIX category: <http://muckdragon.info/hyperspectral/tags/category#>
CONSTRUCT {
    ?v1 prov:wasAttributedTo ?s1 .
} WHERE {
    OPTIONAL {
        ?v1 prov:wasAttributedTo ?s1 .
        ?r1 arv:scientist ?s1 .
        ?r1 arv:sensor ?sens1 .
        ?v1 category:hasCategoryTag category:bathymetry .
        ?v1 measure:hasMeasurementTag measure:depth .
    }
}
```

4.6.5 Find Scientists: Hyperspectral Research, and Sensors Used

A researcher in the hyperspectral field wants to locate all scientists who performed hyperspectral research on a given data set, and also wants to know what sensors are associated with his or her studies. This is equivalent to the Enterprise query described in Algorithm (11), but is more complex as there are three different categorizations that we are looking for, phytoplankton absorption, particulate absorption, and sea surface temperatures.

4.6.6 List Scientists' Data: Hyperspectral

Noting some of the sensor types, the researcher queries the scientists' data which is related to the hyperspectral imaging. This allows the researcher to decide which, if any, of the listed sensors actually apply to the data being studied. This is the same as Algorithm(13), with the line in the CONSTRUCT section "?v1 prov:wasAttributedTo ?s1 ." rather than "?s1 :usedSensor ?sens1".

4.6.7 Review: Enterprise Data Schema

The enterprise data use case was one of the most successful use cases, as it integrated the Bathymetry and Hyperspectral use cases with an additional Enterprise schema together under one schema. This schema became automatically apparent from the union of the three schemas, and a visual representation of such schema is presented as Figure (7). From here, we are able to see the interaction between the Tags (Substance, Category, Algorithm, Sensor, Measurement) from the Hyperspectral data set, the Project Requests in the Enterprise data set, as well as the PROV-O Activities, Entities, and Agents available from both Hyperspectral and Bathymetry data sets.

5 Discussion

As this framework is for the most part a prototype which has proven itself through real-world use, there are many design considerations in this system which may change. For instance, the user interface is

Algorithm 13 SPARQL Query: Hyperspectral Research Enterprise Query

```
PREFIX : <http://muckdragon.info/temp/enterprise-query0001#>
PREFIX proc: <http://muckdragon.info/data/processing#>
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX sets: <http://muckdragon.info/data/sets#>
PREFIX arv:  <http://muckdragon.info/arv/schema/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>
PREFIX measure: <http://muckdragon.info/hyperspectral/tags/measure#>
PREFIX substance: <http://muckdragon.info/hyperspectral/tags/substance#>
CONSTRUCT { ?s1 :usedSensor ?sens1 . } WHERE {
    OPTIONAL {
        ?v1 prov:wasGeneratedBy/prov:used/proc:produces ?p1 ;
        sets:isContainedIn ?f1 .
        ?f1 prov:wasAttributedTo ?s1 .
        ?r1 arv:scientist ?s1 .
        ?r1 arv:sensor ?sens1 .
        ?v1 measure:hasMeasurementTag measure:absorption .
        ?v1 substance:hasSubstanceTag substance:phytoplankton .
    } OPTIONAL {
        ?v1 prov:wasGeneratedBy/prov:used/proc:produces ?p1 ;
        sets:isContainedIn ?f1 .
        ?f1 prov:wasAttributedTo ?s1 .
        ?r1 arv:scientist ?s1 .
        ?r1 arv:sensor ?sens1 .
        ?v1 measure:hasMeasurementTag measure:absorption .
        ?v1 substance:hasSubstanceTag substance:particulate .
    } OPTIONAL {
        ?v1 prov:wasGeneratedBy/prov:used/proc:produces ?p1 ;
        sets:isContainedIn ?f1 .
        ?f1 prov:wasAttributedTo ?s1 .
        ?r1 arv:scientist ?s1 .
        ?r1 arv:sensor ?sens1 .
        ?v1 measure:hasMeasurementTag measure:temperature .
    }
}
```

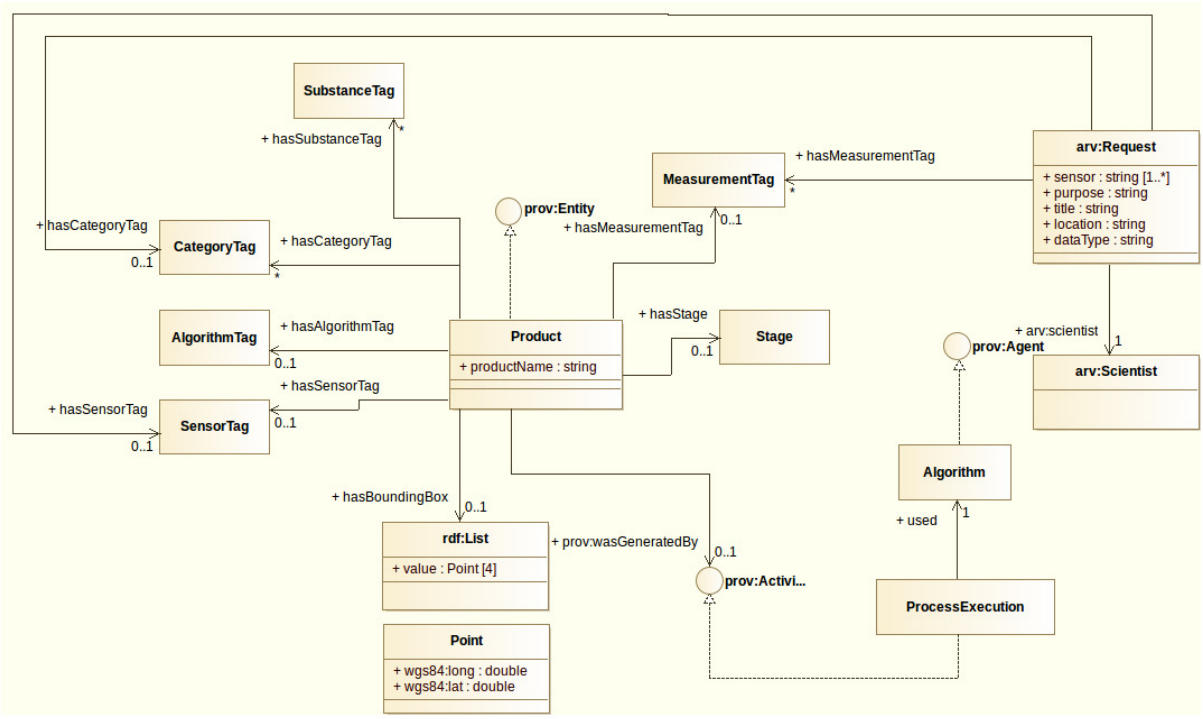

Figure 7: Enterprise Data Schema in Modelio

still not as powerful and intuitive as possible. The user still has to specify variable names for individual results in a query, and manually type literals which are not string or integer literals.

The design also contained no reference to good security practices, and there are few SPARQL APIs which provide anything along the lines of SQL Bound Parameters. Another security consideration is the lack of authentication between the client and the server. For a production system used by a real organization, authentication as well as user permissions is a must, but for this prototype the user account system would likely be too large of an undertaking to make into a useful feature. There is also a lack of security on the database backend itself, with a simple HTTP request all that is necessary to read all of the data contained within the triple store.

The translation component of the framework would be best suited to be redesigned at least somewhat. The current method of hardcoding mappings between data formats and RDF is inflexible and will certainly not work for every user's purposes. A more flexible design would likely include a very simple domain-specific language for specifying direct mappings as well as categorization tags for deepening data structures. While the initial components (such as the categorization tag mapper) are developed, further development would be required to make things as seamless to the user as possible.

There are quite a few improvements which could be made to the user interface. More varied operations on return sets from subqueries should be available (full/inner/outer joins, etc), decent drag-and-drop support in the user interface, and a method of inserting URI literals without a user manually typing them. A more powerful result set visualization system, rather than the limited options of tabular data or serialized RDF in Turtle format, would certainly be desired.

5.1 Further Utilization

There are further potential uses for a curation framework such as this. Any collection of data which is difficult to navigate in traditional hierarchies may be aided by these methods, and automated ways of attaching metadata (EXIF, etc) may be utilized as well. The current state-of-the-art desktop environments such as KDE4 are already starting to use these methods to help users organize their own data. In the case

of KDE4, they are using OpenLink Virtuoso as well as a basic Qt4 GUI to provide a query builder interface of their own. This is a good choice for a database server, as Apache Jena would not integrate due to its Java API, and may lack the performance required for even a single heavy data user's file management purposes.

6 Conclusions

Through our research into the semantic web, we conclude that there are indeed a very large number of issues with data traceability and reusability, especially with large organizations that have a mass of data. This is especially evident in the case of the bathymetry and hyperspectral data, which are real data sets that were developed in a traditional research setting. There was no framework for cataloging the resources even in a very primitive way. Bathymetry data had practically no associated metadata and it was all retrieved manually. Likewise, the enterprise data set was developed over a short period by manual data entry. Despite the fact that data tends to become unmanageable over time, we have demonstrated methods which can be used, in part, to mitigate the effects of poor data management practices by providing a small amount of data management in manageable sizes to users, without the requirement of manually creating a comprehensive schema and database.

References

[1] David Beckett and Tim Berners-Lee. Turtle - terse rdf triple language. Team submission, W3C, http://www.w3.org/TeamSubmission/2011/SUBM-turtle-20110328/, 2011.

[2] Brian McBride. Rdf/xml syntax specification (revised). Recommendation, http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/, February 2004.

[3] Brian McBride. Rdf vocabulary description language 1.0: Rdf schema. Recommendation, http://www.w3.org/TR/2004/REC-rdf-schema-20040210/, February 2004.

[4] Dan Connolly and Tim Berners-Lee. Notation3 (n3): A readable rdf syntax. Team submission, W3C, http://www.w3.org/TeamSubmission/n3/, March 2011 2011.

[5] W3C OWL Working Group. Owl 2 web ontology language document overview (second edition). Recommendation, W3C, http://www.w3.org/TR/owl2-overview/, December 2012. 2012.

[6] Steve Harris, Eric Prud'hommeaux, and Andy Seaborne. Sparql 1.1 query language. Recommendation, W3C, http://www.w3.org/TR/2013/REC-sparql11-query-20130321/, March 2013.

[7] Openlink virtuoso. Web, February 2012.

[8] 4store - scalable rdf storage. Web, 2009.

[9] Apache Software Foundation. Apache jena - an introduction to rdf and the jena rdf api. 2013.

[10] *Zorba 2.9.1 Documentation*. http://www.zorba.io/documentation/2.9/zorba/indexpage.html, 2013.

[11] DHTMLX, http://docs.dhtmlx.com/doku.php. *DHTMLX 3.6 Manual*, 2013.

[12] Sencha, http://docs.sencha.com/extjs/4.2.1/. *ExtJS 4.2.1 Manual*, May 2013.

[13] semsol and MattiSG. Getting started with arc2, 2012.

[14] Stanford Protege, http://protege.stanford.edu/doc/faq.html. *Protege FAQ*, 2013.

[15] The HDF Group, http://www.hdfgroup.org/hdf-java-html/hdf-object/index.html. *HDF Object Package*, May 2011.

[16] Library of Congress, http://www.loc.gov/standards/marcxml/. *MARC21 XML Schema*, 2012.

Vita

The author Brian Becker was born in Merced, CA, moving early in his childhood to Northwestern Louisiana. He attended the University of New Orleans and received his Bachelor's degree in 2011, continuing his studies as a Masters student. His research was conducted under the supervision of his advisor, Shengru Tu.