Ontology based Querying and Integration of Heterogeneous Flat Files

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Rohit Dinakar

Graduate Program in Computer Science and Engineering

The Ohio State University

2010

Master's Examination Committee:

Dr. Gagan Agrawal, Advisor

Dr. Hakan Ferhatosmanoglu
Abstract

In scientific domains, most of the measurements collected during observation periods are stored in flat files. In many cases, especially when different scientists come together from different fields to draw comprehensive conclusions, the formats of the files vary from one group to another. Integrating, querying, and retrieving data from such heterogeneous data files present a challenge. Semantic interoperability is essential in order to harmonize these datasets.

In this thesis, we describe an ontology-based system that parses, summarizes, represents and integrates heterogeneous data files stored as flat files. The test bed dataset is from the Episodic Events Great Lakes Experiment (EEGLE) project which collected over 500 MB of data in more than 1,500 objects. Existing works on querying hydrological data involve the use of relational databases and do not provide ways to query within the flat files. Hence efficient ways are required to eliminate the overhead associated with relational databases and still provide the flexibility and ease of querying that relational databases offer.

We develop an intuitive approach using ontologies to integrate and query the semi-structured data present in the flat files. The crawled data from the flat files is
represented in XML using resource scripts to provide a structure and schema to it. We then create ontologies with rules using Protégé-OWL editor to semantically represent the data being observed. The ontologies are mapped with the XML data to generate records similar to relational database records. Finally, these mapped records can be queried from a custom-built interface to get the desired results.

Currently, the system that we have developed supports simple column queries, range queries and similarity queries. There is also support for keyword-based semantic queries through the Protégé-OWL editor. Our system makes use of the right tools to integrate and represent the data semantically since we intend to provide as much semantic support as possible through the use of ontologies. Since we deal with domain-specific data, the robustness of the system can only be determined by how well we support normal and semantic querying. The ontologies can be enriched semantically to extend support for complex queries.
This work is dedicated to my parents, Mrs. Girija Dinakar and Mr. S. Dinakar and my motherland, India.
Acknowledgments

First and foremost, I would like to thank my advisor Prof. Hakan Ferhatosmanoglu, for his patience and constant support throughout the course of my Masters. His guidance and encouragement helped me remain focused on my work. I would also like to thank Prof. Gagan Agrawal for agreeing to be on my Master’s Thesis Committee and for all the valuable suggestions he has made.

Special thanks to Dr. Guadalupe Canahuate and Yogesh Pendharkar for initiating this project and for all the technical inputs I received.

I extend my sincere gratitude to the University Libraries for supporting me financially during my Masters program, and helping me to achieve my academic goals.

Thanks to the entire team at the Database Research lab for their constant support, suggestions and making my stay a memorable experience.

Finally, this journey would not have been possible without the support and love of my parents. I am forever indebted to them for all the values and principles they have imbibed in me.
Vita

2002 – 2006…………………………………….B.Tech, Information Technology, Anna University, Chennai, India

2006 – 2008……………………………………Software Engineer, Intel India

2009 - 2010 .................................Graduate Administrative Associate, University Libraries, The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering
Table of Contents

Abstract................................................................................................................................. ii
Dedication .............................................................................................................................. iv
Acknowledgments.................................................................................................................. v
Vita........................................................................................................................................ vi
List of Figures ........................................................................................................................ x
Chapter 1: Introduction......................................................................................................... 1
  1.1 Introduction to Ontologies .............................................................................................. 3
     1.1.1 Domain Ontologies.................................................................................................... 4
  1.2 Ontology Components ................................................................................................... 5
     1.2.1 Individuals ................................................................................................................. 6
     1.2.2 Classes ....................................................................................................................... 6
     1.2.3 Attributes .................................................................................................................. 8
     1.2.4 Relationships ............................................................................................................ 9
  1.3 Related Work.................................................................................................................. 9
  1.4 System Architecture ..................................................................................................... 12
Chapter 2: Conversion to XML ...................................................................................... 14
  2.1 EEGLE Dataset ................................................................................................. 14
  2.2 ServingXML .................................................................................................... 18
    2.2.1 Resource Scripts ......................................................................................... 21
Chapter 3: Building Ontologies using OWL ............................................................ 23
  3.1 OWL Overview ............................................................................................... 23
    3.1.1 The Three Species of OWL ....................................................................... 24
    3.1.2 OWL Header ............................................................................................ 26
    3.1.3 OWL Instances ........................................................................................ 27
    3.1.4 OWL Properties ....................................................................................... 27
    3.1.5 OWL Classes ........................................................................................... 28
  3.2 XML to OWL Mapping ..................................................................................... 30
    3.2.1 Mapping XML Nodes to OWL Classes ...................................................... 32
    3.2.2 Mapping XML Nodes to OWL Properties .................................................. 32
    3.2.3 Generating Class Instances ..................................................................... 33
    3.3 JXML2OWL ................................................................................................ 34
Chapter 4: Query Execution and Front End ............................................................... 37
  4.1 Supported Queries ............................................................................................ 37
    4.1.1 Column Queries ........................................................................................ 37
List of Figures

Figure 1.1 Illustration of Classes in Ontology ................................................................. 8
Figure 1.2 System Architecture ..................................................................................... 12
Figure 2.1 Sample of text file formats produced by the EEGLE project ..................... 16
Figure 2.2 Snapshot of the EEGLE project data interface ........................................... 17
Figure 2.3 ServingXML Input file .................................................................................. 19
Figure 2.4 ServingXML Output file ................................................................................ 20
Figure 2.5 ServingXML Resource Script file ............................................................... 21
Figure 3.1 Subclass relationships between OWL and RDF/RDFS ............................... 24
Figure 3.2 Sample OWL header .................................................................................... 26
Figure 3.3 Representation of OWL Instances ............................................................... 27
Figure 3.4 Representation of OWL Properties ............................................................... 28
Figure 3.5 Creation of an OWL Datatype property ....................................................... 28
Figure 3.6 Representation of OWL Classes (containing individuals) ......................... 29
Figure 3.7 Creation of an OWL Class ............................................................................ 29
Figure 3.8 Correspondence between XML schema and OWL classes/properties ........ 30
Figure 3.9 JXML2OWL Conversion process ................................................................. 35
Figure 4.1 An Example of a Column query on the ontology ........................................ 38
Figure 4.2 An Example of a max query on the ontology .............................................. 40
Figure 4.3 Euclidean distance ................................................................. 41
Figure 4.4 An example of a Similarity Query on the ontology ...................... 42
Figure 4.5 An example of a DL query in Protégé-OWL ............................ 44
Chapter 1: Introduction

Interdisciplinary efforts to derive novel conclusions about certain phenomena are becoming not only desirable but necessary. The goal usually is to perform comprehensive analysis involving all aspects of the phenomenon. In many scientific and engineering areas, a particular challenge arises because most of the measurements/experiments collected during observation periods are stored in flat files. Flat files are the easiest way to compile data as they offer great flexibility, portability. There is no expense or training required, in clear contrast to what will be involved in using modern databases or other advanced software. Some sensors do not have better capability than producing low level flat files. In addition, since the data is collected in a distributed fashion, files are managed by their owners. They are the ones that know how to interpret the data and analyze it. Typically, some metadata file is associated with the flat files to explain its content and layout. However, these metadata files are written by the owner(s)/producer(s) of the data to help other humans in understanding the content of the data.

As an example of a large effort involving such challenges, consider the ongoing WATERS (WATer and Environmental Research Systems) Network effort [18], whose goal is to use Networked sensors, assimilation of high-frequency data and interdisciplinary experimentation data to enable forecasting and management of critical
water processes affecting and affected by human activities. This effort is trying to integrate data collected at more than 1200 sites. The specific challenges include sharing of data collected in different formats, comprising different attributes, and measured using different units, and processing of this data by various services and programs developed by different research groups.

In such cases, the net output from data collection is heterogeneous datasets whose integration possesses numerous technical challenges. Even when standards are specified within each group of experts, adopting a worldwide standard is difficult. This is because tools and systems may already exist that require/produce data in a given format, and a change is either not possible or is prohibitively expensive. This is why the integration problem, though critical for scientific advances, is almost never addressed till the very end of the project. In many cases, the integration of such data is replaced by the exchange of partial findings among groups. However, in this scenario, it is not possible to present a comprehensive view of the system or query the collected data across files, and much of the effort and costs associated with data collection is wasted.

Semantic interoperability is essential in order to harmonize these datasets. Reading a flat-file is easy, but understanding its content and integrating it across domains is only possible by interpreting the content and deriving a meaningful representation that can be combined at a semantic level. There is a need to support semantic queries over the dataset since the people who use the system may not be proficient in complex query languages. A simple natural language query support is possible only by semantic representation of the data.
In this thesis, we present an ontology-based system that parses heterogeneous flat files, summarizes them, and semantically represents the data to integrate heterogeneous data files into a single ontology. An open source tool called ServingXML [19] is used to crawl the flat files and convert them to XML documents. Custom ontologies are created in OWL (Web Ontology Language) [20] using the Protégé-OWL editor for semantic representation of data. The mapping between the XML documents and the ontologies is done using JXML2OWL API. Finally, the integrated ontology can be queried using a simple interface built in Java and also by semantic reasoners available as part of the Protégé-OWL editor.

1.1 Introduction to Ontologies

In computer science and information science, ontology is a formal representation of the knowledge by a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain, and may be used to describe the domain. In theory, ontology is a "formal, explicit specification of a shared conceptualization" [1]. An ontology provides a shared vocabulary, which can be used to model a domain — that is, the type of objects and/or concepts that exist, and their properties and relations. Ontologies are used in artificial intelligence, the semantic web, systems engineering, software engineering, biomedical informatics, library science, enterprise bookmarking, and information architecture as a form of knowledge representation about the world or some part of it.

The core meaning within computer science is a model for describing the world that consists of a set of types, properties, and relationship types. Exactly what is provided
around these varies, but they are the essentials of an ontology. There is also generally an expectation that there be a close resemblance between the real world and the features of the model in an ontology. According to Gruber (1993) [1], ontologies are often equated with taxonomic hierarchies of classes, class definitions, and the subsumption relation, but ontologies need not be limited to these forms. Ontologies are also not limited to conservative definitions – that is, definitions in the traditional logic sense that only introduce terminology and do not add any knowledge about the world. To specify a conceptualization, one needs to state axioms that do constrain the possible interpretations for the defined terms.

1.1.1 Domain Ontologies

A domain ontology (or domain-specific ontology) models a specific domain, or part of the world. It represents the particular meanings of terms as they apply to that domain. For example the word *card* has many different meanings. An ontology about the domain of poker would model the "playing card" meaning of the word, while an ontology about the domain of computer hardware would model the "punched card" and "video card" meanings.

Since domain ontologies represent concepts in very specific and often eclectic ways, they are often incompatible. As systems that rely on domain ontologies expand, they often need to merge domain ontologies into a more general representation [21]. This presents a challenge to the ontology designer. Different ontologies in the same domain
can also arise due to different perceptions of the domain based on cultural background, education, ideology, or because a different representation language was chosen.

At present, merging ontologies that are not developed from a common foundation ontology is a largely manual process and therefore time-consuming and expensive. Domain ontologies that use the same foundation ontology to provide a set of basic elements with which to specify the meanings of the domain ontology elements can be merged automatically. There are studies on generalized techniques for merging ontologies, but this area of research is still largely theoretical.

1.2 Ontology Components

Contemporary ontologies share many structural similarities, regardless of the language in which they are expressed. Most ontologies describe individuals (instances), classes (concepts), attributes, and relations [22]. Common components of ontologies include:

- **Individuals**: instances or objects (the basic or "ground level" objects)
- **Classes**: sets, collections, concepts, types of objects, or kinds of things.
- **Attributes**: aspects, properties, features, characteristics, or parameters that objects (and classes) can have
- **Relations**: ways in which classes and individuals can be related to one another
- **Function terms**: complex structures formed from certain relations that can be used in place of an individual term in a statement
- **Restrictions**: formally stated descriptions of what must be true in order for some assertion to be accepted as input

- **Rules**: statements in the form of an if-then (antecedent-consequent) sentence that describe the logical inferences that can be drawn from an assertion in a particular form

- **Axioms**: assertions (including rules) in a logical form that together comprise the overall theory that the ontology describes in its domain of application.

- **Events**: the changing of attributes or relations

Ontologies are commonly encoded using ontology languages.

1.2.1 **Individuals**

Individuals (instances) are the basic, "ground level" components of an ontology. The individuals in an ontology may include concrete objects such as people, animals, tables, automobiles, molecules, and planets, as well as abstract individuals such as numbers and words. Strictly speaking, an ontology need not include any individuals, but one of the general purposes of an ontology is to provide a means of classifying individuals, even if those individuals are not explicitly part of the ontology.

1.2.2 **Classes**

Classes – concepts that are also called *type, sort, category, and kind* – can be defined as an extension or an intension. According to an extensional definition, they are abstract groups, sets, or collections of objects. According to an intensional definition,
they are abstract objects that are defined by values of aspects that are constraints for being member of the class. The first definition of class results in ontologies in which a class is a subclass of collection. The second definition of class results in ontologies in which collections and classes are more fundamentally different. Classes may classify individuals, other classes, or a combination of both. Some examples of classes:

- **Person**, the class of all people, or the abstract object that can be described by the criteria for being a person.
- **Vehicle**, the class of all vehicles, or the abstract object that can be described by the criteria for being a vehicle.
- **Car**, the class of all cars, or the abstract object that can be described by the criteria for being a car.
- **Thing**, representing the class of all things, or the abstract object that can be described by the criteria for being a thing.

Importantly, a class can subsume or be subsumed by other classes as illustrated in Figure 1.1; a class subsumed by another is called a subclass (or subtype) of the subsuming class (or supertype). For example, **Vehicle** subsumes **Car**, since (necessarily) anything that is a member of the latter class is a member of the former. The subsumption relation is used to create a hierarchy of classes, typically with a maximally general class like **Anything** at the top, and very specific classes like **2002 Ford Explorer** at the bottom. The critically important consequence of the subsumption relation is the inheritance of properties from the parent (subsuming) class to the child (subsumed) class. Thus, anything that is necessarily true of a parent class is also necessarily true of all of its
subsumed child classes. In some ontologies, a class is only allowed to have one parent (*single inheritance*), but in most ontologies, classes are allowed to have any number of parents (*multiple inheritance*), and in the latter case all necessary properties of each parent are inherited by the subsumed child class. Thus a particular class of animal (*HouseCat*) may be a child of the class *Cat* and also a child of the class *Pet*.

![Figure 1.1: Illustration of Classes in Ontology](image)

### 1.2.3 Attributes

Objects in an ontology can be described by relating them to other things, typically aspects or parts. These related things are often called *attributes*, although they may be independent things. Each attribute can be a class or an individual. The kind of object and the kind of attribute determine the kind of relation between them. A relation between an
object and an attribute express a fact that is specific to the object to which it is related.

For example the Ford Explorer object has attributes such as:

- <has as name> Ford Explorer
- <has by definition as part> door (with as minimum and maximum cardinality: 4)
- <has by definition as part one of> \{4.0L engine, 4.6L engine\}
- <has by definition as part> 6-speed transmission

1.2.4 Relationships

Relationships (also known as relations) between objects in an ontology specify how objects are related to other objects. Typically a relation is of a particular type (or class) that specifies in what sense the object is related to the other object in the ontology. For example in the ontology that contains the concept Ford Explorer and the concept Ford Bronco might be related by a relation of type <is defined as a successor of>. The full expression of that fact then becomes:

- Ford Explorer is defined as a successor of : Ford Bronco

The most important type of relation is the subsumption relation (is-a-superclass-of, the converse of is-a, is-a-subtype-of or is-a-subclass-of). This defines which objects are classified by which class. For example the class Ford Explorer is-a-subclass-of 4-Wheel Drive Car, which in turn is-a-subclass-of Car.
1.3 Related Work

Ontologies are used in various domains to achieve semantic interoperability. Gruber [1] addressed the problem of portability for common ontologies over heterogeneous systems proposing the translation approach where the ontologies are represented into common, system independent form and then translated into system specific representation. Sheth [2] discussed the need of focusing on semantics while answering the user query requests. The first ontological approach in information systems was proposed by Guarino [3] in an Ontology-driven Information System (ODIS) architecture where the use of ontologies at development time as well as run time is discussed. Frank [4] discusses the use of ontologies in geographical domain assuming the ontologies are already available. Fonseca [5] make use of ontologies for the integration of geographical data sets proposing ODGIS (Ontology-driven Geographic Information System) architecture that acts as a system integrator.

In [6], Chaves et al. discuss the data model called Geographic Knowledge Base (GKB) for integrating geographic data from multiple domains and generating ontologies from them. Other works such as [7] and [8] discuss the use of ontologies towards the disambiguation of geographical names. In [9], Viegas et al. discuss the ontology based approach for querying the geographical database by mapping the relationships between various classes.

For the mapping between XML and OWL, Bohring and Auer [10] propose a ready-to-use XSLT framework approach for generating ontologies automatically out of existing XML data with relational origins. Xiao and Cruz [11] provide a formal model for
the mappings between XML schemas and local RDFS ontologies and those between local ontologies and the global RDFS ontology. In [12], a tool called X2OWL that aims at building OWL ontology from an XML data source is presented. This method is based on XML schema to automatically generate the ontology structure, as well as, a set of mapping bridges. Rodrigues [13] et al. present a framework, JXML2OWL that allows organizations to automatically convert their XML data sources to a semantic model defined in OWL. We use the APIs of JXML2OWL for our mapping.

In conclusion, related works on ontology based query processing are too domain specific and the ontologies need to be modeled based on the needs of the domain and the types of the queries that are intended to be supported. For example, the Pharos Environmental Software [26] developed for the Texas Coastal Ocean Observation Network furnishes data collected by automated equipment “as is”. Consequently, the scope of querying is limited in such cases. We compile the available elements, apply them in our system design, and build on top of existing components. The result is a flexible architecture that allows for advanced query processing over heterogeneous flat files.

The rest of the thesis is organized as follows. In Section 1.4, we provide an overall system architecture. In Chapter 2, we discuss the process of conversion to XML and also about the EEGLE dataset being used. In Chapter 3, we provide an overview of the OWL language and discuss how the actual mapping is done from the XML data to OWL ontologies. Chapter 4 contains details about the query processing and execution
part. We discuss the types of queries supported and also present examples with results. In Chapter 5, we conclude with the future work.

1.4 System Architecture

The aim of the proposed system is to integrate heterogeneous datasets stored as flat files and enable query processing over the integrated data. Figure 1.2 depicts the system architecture.

Figure 1.2: System Architecture
There are 3 main components to the system, namely

- **XML Conversion** – The first step is to convert all the flat files to XML data, thereby having one common schema for files containing same measured attributes.

- **Ontology Building process** - Assuming we have the XML files containing all instance data from the flat files, the next step would be to map this data into OWL ontologies.

- **Query Execution and Front end** – In the Query Execution phase, we build a simple interface which can be used to query for data within the flat files. Querying can be done at attribute level as well as record level.

These components are described in detail in the coming chapters.
Chapter 2: Conversion to XML

XML is well suited for the interchange of data, since XML documents are self-describing, easily parsed and can represent complex data structures. Also, there is a wide variety of high-quality, inexpensive tools for parsing and transforming XML documents. We chose to convert all flat files to XML since OWL, which is described in the next chapter, can be encoded using XML.

2.1 EEGLE Dataset

The datasets refer to heterogeneous data stored in different formats as low level flat files containing data that can be combined at a semantic level. Scientific domains provide abundant data with these characteristics. As a test bed we use the data produced by the Episodic Events Great Lakes Experiment (EEGLE) project.

A team of over 40 environmental scientists from federal agencies and universities put together a comprehensive interdisciplinary research program to study the Lake Michigan plume supported by NSF and NOAA. Teams of specialists came together to conduct experiments focusing on the same region, to provide a unique opportunity for new insights into coupling between biological, chemical, and physical processes.

The Episodic Events Great Lakes Experiment (EEGLE) [14] project includes three years of intensive data collection. Some moorings measured water velocities and
temperatures throughout the water column, while others collected the plume materials as they sank towards the lake bottom. Also, radar sites were installed to study surface currents, winds, and wind waves. During the plume, multiple shipboard surveys, cruised over the lake to collect data and samples. The collected data is used to create hydrodynamic models (currents, temperature, wave, and ice), sediment transport data, and lower food web simulations. The goal is to measure and assess the impact of episodic events on the nearshore-offshore transport and transformation of biochemically important materials in the Great Lakes. The findings are used to evaluate future lake management options, and provide a more realistic assessment of how nutrients and contaminants in the sediments continue to recycle within the lake.

As expected, the heterogeneity in scientific fields also produced heterogeneity in the data collected. The data is stored as flat-files. Figure 2.1 shows some of the text file formats produced by the EEGLE project. As can be seen, some files have a header to identify the field names, others have some metadata information such as the location (lat and long) and the station name (in one line or one per line), while others have no header at all. The common element in all the files is that several measurements are collected (data lines) and the data fields are tab delimited. This is the only assumption we made about the data format in the conversion phase. Metadata files also exist and most of them consist of unstructured text describing the experiment.
Figure 2.1: Sample of text file formats produced by the EEGLE project

A web interface was developed to enable browsing the data using different criteria. Figure 2.2 shows a snapshot of the main data page. The query interface offers
limited functionality as the results are shown as the data file in its original format and it is not possible to query the actual data contained in the files. However, this system is what makes the EEGLE project such a good candidate to test and evaluate the proposed system. Our results are validated against the information compiled in this system and our ontology is partly populated using the object relationships already established on this webpage.

Figure 2.2: Snapshot of the EEGLE project data interface
2.2 ServingXML

ServingXML is an open source, Apache 2.0 licensed, framework for flat/XML data transformations. It defines an extensible markup vocabulary for expressing flat-XML, XML-flat, flat-flat, and XML-XML processing in pipelines.

The tool requires us to create resource scripts specifying the attributes contained in the flat files that should become the tags in the XML file. The syntax for running the console application is:

```
sevingxml [-options] service [param=value...]
```

The options are as follows.

- `-r` - Identifies the resources script.
- `-i` - Identifies the default input file (if not specified it defaults to the "standard" input stream.)
- `-o` - Identifies the default output file (if not specified it defaults to the "standard" output stream.)
- `-c` - Identifies an optional configuration file. If this option is not specified, ServingXML looks for a configuration file called servingxml.xml in the classpath. If none is found, default values are used.
- `-T` - Show exception stack trace.
- `-help` - Print help.
- `-version` - Print the version

Consider a tab-delimited snapshot of an input file from the actual dataset as shown in Figure 2.3. As we can see, this particular flat file measures various parameters
like Depth, Temperature, Oxygen content, conductivity, pH etc. of the water body. All values measured are in floating point notation. Also, this particular flat file contains few lines of header information. Typically, the header contains the latitudinal and longitudinal information of the region where the measurements were collected. It may also contain information about the cruise and ship.

![Figure 2.3: ServingXML Input file](image)

The output XML file should have the attributes measured in these flat files as element tags with the values inside them. We structure each line of the measured input as a record element in the XML file. The root element would be an identifier that denotes the scientist who made the observations. Figure 2.4 gives a snapshot of the desired output XML file for the input flat file described above.
Figure 2.4: ServingXML Output file

This conversion is possible through the use of resource scripts which we will see in the next section.
2.2.1 Resource Scripts

The resource script file is required by ServingXML to do the transformation. It is like a configuration file where in we need to specify the structure of the flat file that is being transformed. Resource scripts typically use XSLT transformers to do the transformation. ServingXML responds to requests by invoking a service, which in turn reads content and subjects it to a number of transformations, and finally writes output. We can specify which transformer to use in the configuration file. Figure 2.5 shows a snapshot of the resource script required to transform the input file shown in Figure 2.3 to the output XML file shown in Figure 2.4.

```xml
<sx:flatFile id="ctd_lansing-file">
  <sx:flatFileHeader lineCount="5"/>
  <sx:flatFileBody>
    <sx:fieldDelimiter>
      <sx:whitespaceSeparator/>
    </sx:fieldDelimiter>
    <sx:flatRecordType name="record">
      <sx:delimitedField name="DATE" start="2"/>
      <sx:delimitedField name="DEPTH"/>
      <sx:delimitedField name="TEMP"/>
      <sx:delimitedField name="OXY"/>
      <sx:delimitedField name="COND"/>
      <sx:delimitedField name="PAR"/>
      <sx:delimitedField name="ATTEN"/>
      <sx:delimitedField name="pH"/>
      <sx:delimitedField name="FLUOR"/>
    </sx:flatRecordType>
  </sx:flatFileBody>
</sx:flatFile>
```

Figure 2.5: ServingXML Resource Script file
Some of the tags used in the resource script are as follows:

- `<sx:flatFile>` - Specifies the flat file that is given as input to ServingXML. The input file can be specified at the command line as well.

- `<sx:flatFileHeader>` - Specifies that the input flat file has a header associated with it. Optionally, we can also specify the number of lines in the header and the total length of the header records, including record delimiter, if any.

- `<sx:fieldDelimiter>` - Specifies what character has been used as a delimiter between fields. In our case, the field delimiters are either a tab space or a whitespace character.

- `<sx:delimitedField>` - Specifies the fields that are present in the flat file. After transformation, these fields become element tags in the output XML file. Optionally, we can also specify the starting position of the field in the record. If it is not specified, it defaults to the end of the previous field.
Chapter 3: Building Ontologies Using OWL

As already mentioned in the background, after we do the conversion to XML, the next step is to map the XML instance files to OWL ontologies. We chose to create our own ontologies specific to the data being processed rather than extending existing ontologies like SWEET ontology [15].

3.1 OWL Overview

Ontologies are used to capture knowledge about some domain of interest. An ontology describes the concepts in the domain and also the relationships that hold between those concepts. Different ontology languages provide different facilities. The most recent development in standard ontology languages is OWL from the World Wide Web Consortium (W3C) [16]. OWL makes it possible to describe concepts but it also provides new facilities. It has a richer set of operators and it is based on a different logical model which makes it possible for concepts to be defined as well as described. Complex concepts can therefore be built up in definitions out of simpler concepts. Furthermore, the logical model allows the use of a reasoner which can check whether or not all of the statements and definitions in the ontology are mutually consistent and can also recognize
which concepts fit under which definitions. The reasoner can therefore help to maintain the hierarchy correctly.

Ideally, OWL would be an extension of RDF Schema, in the sense that OWL would use the RDF meaning of classes and properties (rdfs:Class, rdfs:subClassOf, etc), and would add language primitives to support the richer expressiveness. Figure 3.1 shows the subclass relationships between some modeling primitives of OWL and RDF/RDFS.

![Subclass relationships between OWL and RDF/RDFS](image)

Figure 3.1: Subclass relationships between OWL and RDF/RDFS

### 3.1.1 The Three Species of OWL

OWL ontologies may be categorized into three species or sub-languages: OWL-Lite, OWL-DL and OWL-Full [17]. A defining feature of each sub-language is its expressiveness. OWL-Lite is the least expressive sub-language. OWL-Full is the most expressive sub-language. The expressiveness of OWL-DL falls between that of OWL-Lite and OWL-Full. OWL-DL may be considered as an extension of OWL-Lite and OWL-Full an extension of OWL-DL. All varieties of OWL use RDF for their syntax.
Instances are declared as in RDF, using RDF descriptions and typing information. OWL constructors are also specializations of their RDF counterparts.

**OWL-Lite**

OWL-Lite is the syntactically simplest sub-language. It is intended to be used in situations where only a simple class hierarchy and simple constraints are needed. For example, it is envisaged that OWL-Lite will provide a quick migration path for existing thesauri and other conceptually simple hierarchies.

**OWL-DL**

OWL-DL is much more expressive than OWL-Lite and is based on Description Logics (hence the suffix DL). Description Logics are a decidable fragment of First Order Logic and are therefore amenable to automated reasoning. It is therefore possible to automatically compute the classification hierarchy and check for inconsistencies in an ontology that conforms to OWL-DL. We use OWL-DL for creating our ontologies.

**OWL-Full**

OWL-Full is the most expressive OWL sub-language. It is intended to be used in situations where very high expressiveness is more important than being able to guarantee the decidability or computational completeness of the language. It is therefore not possible to perform automated reasoning on OWL-Full ontologies.
3.1.2 OWL Header

OWL documents are usually called OWL ontologies, and are RDF documents. So the root element of an OWL ontology is an rdf:RDF element which also specifies a number of namespaces. An OWL ontology may start with a collection of assertions for house-keeping purposes. These assertions are grouped under an owl:Ontology element which contains comments, version control and inclusion of other ontologies. Figure 3.2 shows a sample header section of the OWL ontology.

![Sample OWL Header](image)

Figure 3.2: Sample OWL Header

The only assertion which has any consequences for the logical meaning of the ontology is owl:imports: this lists other ontologies whose content is assumed to be part of the current. Notice that while namespaces are used for disambiguation purposes, imported ontologies provide definitions that can be used. Usually there will be an import element for each used namespace, but it is possible to import additional ontologies, for example ontologies that provide definitions without introducing any new names.
3.1.3 OWL Instances

Instances/Individuals represent objects in the domain that we are interested in. Instances of classes are declared as in RDF. Unlike typical database systems, OWL does not adopt the unique names assumption, thus: just because two instances have a different name (or: ID), that does not imply that they are indeed different individuals. Figure 3.3 shows a representation of some individuals in some domain. We represent individuals as diamonds.

![Diagram of individuals and their connections]

Figure 3.3: Representation of OWL Instances

3.1.4 OWL Properties

Properties are binary relations on individuals, i.e. properties link two individuals together. In OWL there are two kinds of properties:

- *Object properties* which relate objects to other objects.
  
  Examples are isTaughtBy, supervises etc.

- *Datatype properties* which relate objects to datatype values.

  Examples are phone, title, age etc. OWL does not have any predefined
data types, nor does it provide special definition facilities. Instead it allows
one to use XML Schema data types, thus making use of the layered
architecture of the Semantic Web

Properties can have inverses. Properties can be limited to having a single value – i.e. to
being functional. They can also be either transitive or symmetric.

Figure 3.4: Representation of OWL Properties

Figure 3.5 illustrates the creation of a datatype property using OWL. The domain and
range of the property is also specified.

Figure 3.5: Creation of an OWL datatype property

3.1.5 OWL Classes

OWL classes are interpreted as sets that contain individuals. They are described
using formal (mathematical) descriptions that state precisely the requirements for
membership of the class. Figure 3.5 shows a representation of some classes containing individuals – classes are represented as circles or ovals, rather like sets in Venn diagrams.

Figure 3.5: A representation of some classes containing individuals

For example, the class Cat would contain all the individuals that are cats in our domain of interest. Classes may be organized into a superclass-subclass hierarchy, which is also known as a taxonomy. Subclasses specialize (‘are subsumed by’) their superclasses. One of the key features of OWL-DL is that these superclass-subclass relationships (subsumption relationships) can be computed automatically by a reasoner. In OWL classes are built up of descriptions that specify the conditions that must be satisfied by an individual for it to be a member of the class. Figure 3.7 illustrates the creation of a Class using OWL.

```
<owl:Class rdf:about="#record">
  <rdfs:subClassOf rdf:resource="#ctd_lansing_1997"/>
</owl:Class>
```

Figure 3.7: Creation of an OWL Class

29
3.2 XML to OWL Mapping

In order to fully understand the transformation process of instances between XML and OWL schema we have to understand the differences of these two data models. XML’s data model describes a node labeled tree (independently of using XML Schema or DTD to define the model), while OWL’s data model is based upon the subject-predicate-object triples from RDF. RDF schema defines a vocabulary for creating class hierarchies, attaching properties to classes and adding instance data.

Since the main characteristic of XML Schema and DTDs is to define a tree structure for the data, the transformation of instances from one data model to the other consists to simply map the XML tree structure to a class hierarchy. Therefore, when creating mappings between elements of an XML schema and an OWL schema, we need to consider, in the one hand, the tree structure of XML, and, in the other hand, the class structure of an OWL ontology. Figure 3.6 shows a possible mapping based on the correspondences between XML schema elements and OWL classes and properties.

<table>
<thead>
<tr>
<th>XSD</th>
<th>OWL</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:elements, containing other elements or having at least one attribute</td>
<td>owl:Class, coupled with owl:ObjectProperties</td>
</tr>
<tr>
<td>xsd:elements, with neither sub-elements nor attributes</td>
<td>owl:DatatypeProperties</td>
</tr>
<tr>
<td>named xsd:complexType</td>
<td>owl:Class</td>
</tr>
<tr>
<td>named xsd:SimpleType</td>
<td>owl:DatatypeProperties</td>
</tr>
<tr>
<td>xsd:minOccurs, xsd:maxOccurs</td>
<td>owl:minCardinality, owl:maxCardinality</td>
</tr>
<tr>
<td>xsd:sequence, xsd:all</td>
<td>owl:intersectionOf</td>
</tr>
<tr>
<td>xsd:choice</td>
<td>combination of owl:intersectionOf, owl:unionOf and owl:complementOf</td>
</tr>
</tbody>
</table>

Figure 3.8: Correspondence between XML schema and OWL classes/properties
The nodes of the tree structure can be easily identified and referenced using an XPath expression. Since all the nodes have the same syntactic representation, no more considerations need to be drawn with respect to XML. Dealing with OWL is more involved, since depending on the semantics of an XML node it can be mapped to different OWL elements. Having a particular XML node, we need to consider three possible mapping scenarios to OWL:

- Map an XML node to an OWL concept.
- Map an XML node to an OWL datatype property.
- Relate an XML node to an OWL object property.

The challenge is to formally specify under which conditions an XML node need to be mapped to an OWL class, datatype or object property.

One solution is to identify the XML nodes with an XPath expression. For instance, the XML nodes representing electronics are referenced with /products/electronics/product and can be mapped to the appropriate concept defined by the ontology. Similarly, computers are addressed by /products/computers/product and mapped to the corresponding ontological concept. XPath expressions have other advantages: XML attributes can be easily addressed prefixing the attribute name with the ‘@’ symbol. For instance, /products/computers/product/@price could address the price of a computer. Also, XPath predicates could be used to support conditional mappings.
3.2.1 Mapping XML Nodes to OWL Classes

XML nodes are referenced with XPath expressions while OWL classes are referenced with their URIs. The pair (OWL class URI, XPath expression) identifies a mapping and means that an instance of the OWL class identified by the URI reference is created for each XML node matching the specified XPath expression.

For example, the following pair \( (\text{product:eletronicsProduct, /products/computers/product}) \) indicates that an instance of \text{electronicsProduct} is created for each XML node matching the XPath expression \text{/products/computers/product}.

3.2.2 Mapping XML Nodes to OWL properties

The W3C OWL recommendation defines two kinds of OWL properties: \emph{datatype} and \emph{object properties}. Both properties have a domain and a range. The domain of a property is not always a single class. For instance, it is possible to define the domain of a property as the union of several classes. The range of a property varies according to the type of the property. Datatype properties are properties for which the value is a data literal, such as \text{xs:integer} (where \text{xs} is a prefix associated to the namespace http://www.w3.org/2001/XMLSchema), while object properties take individuals of a particular class as range.

To create a datatype property mapping, the property as well as both its domain and its range must be specified. The OWL datatype property, which is an OWL resource, is addressed by its URI reference or by its prefix and local name. The value of a datatype
property range is a data literal such as xs:integer or xs:string. Such a value can be specified with an XPath expression to indicate the XML element, attribute or node containing the value used to fill the property value.

Mapping OWL object properties is very similar to the mapping of datatype properties. The difference occurs in the range of the property. While the range of datatype properties takes literal values, the range of object properties takes instances of OWL classes. The OWL object property is addressed like any other OWL resource. The domain is specified like the domain of datatype properties. For the same reasons of the domain of properties, the range of object properties is also referenced with a class mapping.

3.2.3 Generating Class instances

OWL instances are generated from the mappings created between the XML schema and the OWL ontology. Instances of OWL classes are characterized by having unique identifiers. When creating the OWL instances document, it must be ensured that unique identifiers are generated for each individual. Another important task is to detect duplicate instances on the XML document.

With the support of many-to-one mappings, several XML nodes identified by the different or even by the same XPath expressions may refer to the same individual. Based on their unique identifier, duplicate instances (instances with the same ID) must be detected and filtered so that only one instance is created. By default, the ID is generated by sequentially concatenating the underscore symbol ‘_’ with the prefix of the mapped class, with its local name and with the string-value of the mapped XML node.
3.3 JXML2OWL

The JXML2OWL API is a generic and reusable open source library for mapping XML schemas to OWL ontologies for the Java platform. More precisely, the mapping tool supports mappings between any XML schema (XSD and DTD) to concepts (classes and properties) of any OWL ontology.

According to the mapping performed, the tool generates mapping rules as an XSL document that allows the automatic transformation of any XML data, that is, any XML document validating against the mapped schema, into instances of the mapped ontology. Generated mapping rules are wrapped in an XSL document to easily support instances transformation. XSLT is the used standard to transform XML documents. The XSL document generated by JXML2OWL can be used by any XSLT processor to automatically transform instances of the mapped schema into instances of the ontology. Figure 3.7 illustrates at a high level how the JXML2OWL tool performs the conversion from XML to OWL.
The mapping process requires the following steps:

i. The first step consists of creating a new mapping project and loading both the XML schema related file (XSD or DTD) and the OWL ontology. If an XML schema is not available, it is possible to load an XML document. In this case, JXML2OWL extracts a possible schema.

ii. In the second step, we create class mapping between elements of the loaded XML schema and concepts of the ontology. Once these mappings are created, it is possible to relate them to each other with the intent of creating object property
mappings, or to relate them with elements of the XML schema to create datatype property mappings.

iii. Finally, in the last step, it is possible to export the transformation rules, generated according to the mapping performed, as an XSL document. With this XSL document it is possible to transform any XML document which validates against the mapped XML schema into individuals of the mapped OWL ontology.

JXML2OWL supports one-to-one, many-to-one, one-to-many and many-to-many mappings. This means that an element of the loaded XML schema can be mapped to several OWL classes and several elements of the schema can correspond to the same OWL class.
Chapter 4: Query Execution and Front End

In the previous chapter, we explained how the mapping between XML and OWL is done using the JXML2OWL API and how the OWL instances are generated. Now, this populated OWL instance file can be queried or reasoned to get the desired results. We use the Protégé-OWL API to manipulate the populated OWL instances programmatically. The Protégé-OWL API is an open-source Java library for the Web Ontology Language and RDF(S). The API provides classes and methods to load and save OWL files, to query and manipulate OWL data models, and to perform reasoning. Furthermore, the API is optimized for the implementation of graphical user interfaces. A standalone Java Swing interface is used for testing the created OWL ontologies and to perform querying.

4.1 Supported Queries

4.1.1 Column Queries

The flat files created by the scientists performing the experiments may contain many attributes and parameters that are being measured. In such cases, he/she might only be interested in studying the values of certain attributes based on which meaningful conclusions can be drawn. For example, if a scientist is interested in studying only about
the attenuation measurements collected from a particular region of the lake, he/she can do so by querying only for that particular attribute within the OWL instance file.

The Protégé-OWL API [23] provides a function by name `getOWLDatatypeProperty()` which returns a collection of all individuals containing the property name specified as a parameter to the function call. Recall from chapter 3 that all attributes measured and stored in the flat files are mapped as OWL Datatype properties in the ontology. Each individual property value can also be accessed by the following way: - `<individual>.getPropertyValue()`

Figure 4.1: An Example of a Column query on the ontology

Figure 4.1 shows the Protégé-OWL editor interface with the user choosing to view all the attribute values of a particular record of the flat file. Likewise, using the test interface developed, we can browse through the values of a particular attribute as well.
4.1.2 Range Queries

Range queries can be considered an extension of column queries with the additional ability to query through a range of values. A scientist may be interested in knowing the lowest depth of the lake from where measurements were taken or the largest pH value observed within the readings from a particular region. These high/low values would help them make meaningful observations regarding the data collected. We support min and max queries for this type of need.

As with column queries, all individuals with the specified property name are got with a call to getOWLDatatypeProperty(). These are then saved to a collection and the collection can be searched to find the maximum or minimum element from it.

Figure 4.2 shows a max query for the attribute attenuation on the ontology. This indicates that 24.2509 is the highest value of attenuation measured from that particular region. Optionally, while creating the ontologies, we can use the OWL constructs, owl:minCardinality and owl:maxCardinality to specify the lower bound and upper bound respectively. In combination, the two can be used to limit the property’s cardinality to a numeric interval.
4.1.3 Similarity Queries

Similarity queries are useful when we need to compare two sets of data. In our case, the scientist performing the experiments in one region would be interested in observing similar regions which show more or less the same characteristics as the region being experimented. More specifically, interesting observations could be made by finding similar attribute values from the same data set. Ontologies help us to execute similarity queries on semi-structured data also.

Each line of observation is considered a record and corresponds to an OWL instance. So, the goal is to find the most similar record for a particular line of observation. There has to be some distance metric associated to define the term ‘similarity’. Since the observations are all real values, we chose to use ‘Euclidean
distance\[^4\] [24] as the similarity metric. So, for each selected record, Euclidean distances are calculated for every other record and the record with the least distance is considered to be the most similar record.

In Cartesian coordinates, if \( \mathbf{p} = (p_1, p_2, ..., p_n) \) and \( \mathbf{q} = (q_1, q_2, ..., q_n) \) are two points in Euclidean \( n\)-space, then the distance from \( \mathbf{p} \) to \( \mathbf{q} \) is given by:

\[
d(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}
\]

Figure 4.3 Euclidean Distance

Here, the points \( p_1, p_2, p_3 \) etc. correspond to the different attribute values measured within a record.
Figure 4.4: An example of a Similarity Query on the ontology

Figure 4.4 shows a similar record that has been computed based on the attribute values of attenuation, conductivity and fluorescence for the record selected.

4.2 DL Queries using Protégé-OWL

Description Logic (DL) queries can also be supported for our ontologies through the Protégé-OWL editor. The query language (class expression) supported by the plugin is based on the Manchester OWL syntax, a user-friendly syntax for OWL DL that is
fundamentally based on collecting all information about a particular class, property, or individual into a single construct, called a frame.

DL queries can only be executed on a classified ontology. So, we use reasoners provided with Protégé-OWL to do the classification. A reasoner is a piece of software able to infer logical consequences from a set of asserted facts or axioms. The notion of a semantic reasoner generalizes that of an inference engine, by providing a richer set of mechanisms to work with. The inference rules are commonly specified by means of an ontology language, and often a description language. Many reasoners use first-order predicate logic to perform reasoning.

Suppose we have an ontology like this:

- Class:
  - Record

- Data Properties:
  - Depth
  - Attenuation
  - Conductivity

And suppose also that we have several hundred instances of class Record in our ontology. To find an instance with a particular value of “Depth”, we enter the following query: `- Depth Value 0.5310`. Any instances found will then be displayed in the query results.
Figure 4.5: An example of a DL query in Protégé-OWL

Figure 4.5 displays the instances from the ontology that have the attribute value of \textit{depth} as 0.5310.

4.3 Complexity

Since various modules and tools are involved, it is not possible to determine the complexity of the whole approach as such. There is more pre-processing involved than the actual execution. For example, the conversion to XML step can be done as a pre-processing step when we have all the input data that we need. The ServingXML tool can be run as a batch file to convert all the flat files required. However, we may have to intervene to modify the resource scripts when the attributes measured in the flat files change.
The ontology building step is an offline process. Creating ontologies specific to the data being processed requires prior knowledge of the data set. Also, it may require some domain knowledge.

Once the ontologies are available, the mapping from XML to OWL and query execution can be done as a single step. The maximum time spent is in loading and parsing the ontology. This step takes less than a second with the Protege OWL parser. Once the ontologies are loaded, the query execution time is negligible.

Similarly, when we load the ontologies onto the Protege-OWL editor, we have to synchronize the reasoner in order to be able to run description logic queries. The synchronization step takes around a second. The time taken for other steps involved in classification like checking concept consistency, computing the inferred hierarchy, computing equivalent classes is negligible.
Chapter 5: Conclusions and Future Work

5.1 Conclusions

In this thesis, we describe an ontology-based system that parses, summarizes, represents and integrates heterogeneous data stored as flat files. We provide a means for semantically querying the EEGLE (Episodic Events Great Lakes Experiment) data present within the flat files without the need for a relational database by representation of the data using ontologies. The system described in this thesis uses the semantics and flexibility provided by XML and ontologies to achieve this.

In Chapter 1, we described the concept of ontology and described its components in detail. Also, we provided overall system architecture. In Chapter 2, we gave an overview of the EEGLE dataset and the different formats of data to deal with. We also described an open source tool, ServingXML, which we use to convert the semi-structured data present in flat files to XML formats, thereby giving a schema to it. In Chapter 3, we described how we create ontologies using the Web Ontology Language (OWL). Also, the techniques involved in mapping XML to OWL using the JXML2OWL API were discussed. In Chapter 4, we presented how queries can be executed over the populated ontology instances and the types of queries that the system supports.
To demonstrate these capabilities, we built a simple interface using Java Swing to test the ontologies. Column queries, range queries such as max and min queries, and similarity queries can be run on these ontologies. Also, the created ontologies can be loaded onto Protégé-OWL editor to support Description logic queries as well.

5.2 Future Work

The current system can be enhanced to support a wide variety of features including the following:-

- If the files contain different attributes, currently the system requires that the resource script be modified each time to accommodate the changes. The challenge here is to automate the XML generation step so that new data can be seamlessly converted to XML without requiring manual intervention.

- We chose to create our own ontologies rather than extending existing ontologies since the ontologies are dependent on the data being measured. Alternatively, we can use ontologies provided by SWEET (Semantic Web for Earth and Environmental Terminology) that provide a common semantic framework for various Earth science initiatives.

- Ontologies can be enriched by adding more classes and classifying instances against those classes. For example, we could create a class such as HighAtten which would contain all instances that have an attenuation value of 2.0 and above. Classifying instances into various classes gives ample scope for reasoning and semantic queries.
Currently, we only support simple queries like column queries, min/max queries, similarity queries, DL-queries. SPARQL (SPARQL Protocol and RDF Query Language) [25] is the emerging standard for semantic queries. The system can be extended to support complex queries through SPARQL. Also, to support natural language queries, query translation and rewriting to SPARQL can be done.

The query results can be displayed using visualization tools such as Google Maps so that location based queries can be supported as well. Output visualization would help the scientists derive meaningful conclusions from the results.

A web interface can be developed so that all these APIs can be exposed as web services.
References


[18] WATERS project http://www.watersnet.org/


[23] Protégé-OWL API http://protege.stanford.edu/plugins/owl/api/


[26] Pharos Environmental Software query page http://lighthouse.tamucc.edu/pq