Bridging the Ontological Gap between Semantic Web and the RESTful Web Services

Yuan Jin

School of Computer Science
McGill University
Montreal, Canada

August 2010

A thesis submitted to McGill University in partial fulfilment of the requirements of the degree of Master of Science.

Dedication

To My Parents

My Support and My Love

Acknowledgements

First, I'd like to express appreciation to my two supervisors – Prof. Xue Liu and Prof. Renee Sieber. Prof. Liu offered me many research chances in Computer Sciences and taught me how to do research from the very beginning. The research team Prof. Liu created in the Cyber-Physical Lab is powerful, in which many members become my good friends in research and life. Prof. Sieber offered me the precious chance to do proof-of-concept Semantic Web project with her and her team. She's very patient with her research and her students. And she has very good management skills with which she could coordinate students from Computer Science, Geography, and East Asian Studies to work together perfectly, in different phases of the project. She helped me find the research problem of this project, which, as I believe, resolves some realistic problems for the Semantic Web community.

I'd also like to thank Chris Wellen, Jimmy Li, and Jin Xing for their contribution to this project. Without them and their work, this project could never realize what it expects to achieve. Chris Wellen designs and maintains the ontology, which creates concepts and relations between those concepts for geographical, historical, biographical and literary knowledge related to China. It heralds similar studies and will continue to serve as a good example of interdisciplinary studies for the humanities. Jimmy Li joined the project quite late but he is very creative in user interface designing. The ontology-driven user interface is surely a valuable way to combine the findings in Semantic Web with the user interface designing. Jin Xing helps me to optimize my architecture of the project and provides critical suggestions to my documentation of the coding. His knowledge of Servlet and Servlet coding adds more flexibility to the project.

I'm also very grateful to Bin Chen, PLA major and visiting scholar from China's National University of Defense Technology. He is a very experienced developer who helped me solve many technical problems, e.g., different encodings between the program and the database server. He's a good cook as well (specialized in mixing Chinese northern and southern foods) and I'll always remember his cooking, before I met him again.

Zhe Chen and Guofei Zhou all used to be my roommates. I'm sure they are the most tolerant roommates you could find in Montreal now. Zhe's aspiration for world's first-class Ph.D education has stimulated my sluggish thesis writing. His knowledge and wisdom in Mathematics, as well as his persistence in doing researches surely would bring his dreams true before long. Guofei, one of my best friends in Canada, shares his luxurious apartment with me at the time I need it. The quietness of his laboratory, the two warm-hearted Bangladeshi researchers in his lab and his cooking skills all impressed me. I also want to thank him for his precious time spent on the teachings in the Integrated Circuit design. Even though I cannot remember them now.

Last but not least, I owe my sincere gratitude to Miss Li Xinyi, who recently becomes my mental masseuse, before I get too depressed with my writing. She is the one you can hardly find these days who is characterized by her cleverness, humor, and patience.

Abstract

Data are produced in large quantities and in various forms around the globe everyday. Researchers advance their research depending on the availability of necessary data and the discovery of them. As people's demand to manage the data grows, however, three problems appear to hinder the attempts to effectively leverage the data. One is the semantic heterogeneity found in linking different data sources. Database designers create data with different semantics; even data within the same domain may differ in meaning. If users want to acquire all the obtainable information, they have to write different queries with different semantics. One solution to such a problem is the use of ontology. An ontology is defined as a specification for the concepts of an agent (or a community of agents) and the relationships between them (Gruber 1995). Concepts and relationships between concepts are extracted from the data to form knowledge network. Other parties wishing to connect their data to the knowledge network could share, enrich and distribute the vocabulary of the ontology. Users could also write queries to the ontology by any RDF query language (Brickly 2004). The use of ontology is part of the Web 3.0's effort to provide a semantic-sensitive global knowledge network.

A second problem is about new ways to access data resources with ontology information. People used to build application-specific user interfaces to databases, which were offline. Now many choose to expose data in Web Services. Web services are a system to provide HTTP-based remote request calling services that are described in a machine-readable format (Haas and Brown 2004). They usually provide application (or web) programming interfaces to manage data. The question is Web Services are born in a world of applications relying on conventional ways to connect

to data sources. For example, D2RQ (Bizer and Seaborne 2004) translates queries against ontology to SQL queries and it depends on JDBC to read from relational databases. Now the interfaces for these data sources are going to be changed. The Semantic Web world faces the challenge to lose data sources. If Web Services were going to spread over the Internet one day, this lack of connection would hold back me from applying the ontology to connect to heterogeneous data sources.

A third problem (or constraint) is working within the specific project domain. I embed this within a humanities cyberinfrastructure that integrates Chinese biographical, historical and geographical data. The data sources come in various forms – local and remote relational databases and, RESTful Web Services. Working with both legacy databases and the new web application interfaces narrowed down my choice of solutions. Commercial products provide ways to "ontologicalize" the Web Services. I argue that they are heavyweight (e.g. unnecessary components bound with the product) and cost-prohibitive for small-scale projects like ours. Several mature open source solutions featuring working with relational databases provide no or very limited access to Web Services. For example, no clue is found in D2RQ to join Web Services into their system, while OpenLink Virtuoso answers calls for SOAP but cannot manage data from RESTful Web Services.

I propose to build a connection between ontologies and Web Services. I devise the metadata to represent non-RDF Web Services in ontology, and I revise the code and create new data structures in D2RQ to support ontology queries to data from RESTful Web Services.

Abstrait

Les données sont produites en grandes quantités et sous diverses formes dans le monde et tous les jours. Les chercheurs avancer leurs recherches en fonction de la disponibilité des données nécessaires et la découverte de leur. Comme la demande des gens pour gérer les données croît, toutefois, trois problèmes semblent entraver les tentatives d'exploiter efficacement les données. La première est l'hétérogénéité sémantique dans reliant différentes sources de données. Concepteurs de créer des données de base de données avec une sémantique différente; même les données dans le même domaine peuvent avoir une signification différente. Si les utilisateurs souhaitent obtenir toute l'information obtenue, ils doivent écrire des requêtes différentes avec une sémantique différente. Une solution à ce problème est l'utilisation de l'ontologie. Une ontologie est définie comme une spécification pour les concepts d'un agent (ou d'une communauté d'agents) et les relations entre eux (Gruber 1995). Concepts et les relations entre les concepts sont extraites des données pour former réseau de connaissances. Les autres parties qui souhaitent se connecter leurs données au réseau de connaissances pourraient partager, enrichir et diffuser le vocabulaire de l'ontologie. Les utilisateurs peuvent aussi écrire des requêtes à l'ontologie par une requête RDF langue (Brickley 2004). L'utilisation de l'ontologie est une partie de l'effort de Web 3.0 pour fournir un réseau de connaissances sémantiques sensibles mondiale.

Un deuxième problème est sur le point de nouvelles façons d'accéder aux données des ressources de l'information ontologie. Les gens de construire des interfaces utilisateur des applications spécifiques aux bases de données, qui ont été mises hors. Maintenant, de nombreux fournisseurs de données choisir pour exposer les données des services

web. Les services web sont un système pour fournir la demande HTTP à distance d'appeler les services qui sont décrits dans un format lisible par machine (Haas and Brown 2004). Ils fournissent généralement l'application (ou web) interfaces de programmation pour gérer les données. La question est des services web sont nés dans un monde d'applications s'appuyant sur les moyens classiques pour se connecter à des sources de données. Par exemple, D2RQ (Bizer and Seaborne 2004) se traduit par des requêtes sur l'ontologie de requêtes SQL, et cela dépend de JDBC pour lire à partir des bases de données relationnelles. Maintenant, les interfaces de ces sources de données vont être modifiées. Le monde du web sémantique doit relever le défi de perdre des sources de données. Si les services web ont été va se répandre sur Internet, un jour, ce manque de connexion tiendrait nous ramène de l'application de l'ontologie de se connecter à des sources de données hétérogènes.

Un troisième problème (ou contrainte) est travailler dans le domaine des projets spécifiques. Nous incorporer cela dans une cyber-infrastructure qui intègre les sciences humaines chinois biographiques, des données historiques et géographiques. Les sources de données prennent des formes diverses - bases de données locales et distantes relationnelles et, les services web RESTful. Travailler avec les anciennes bases de données à la fois et l'application web de nouvelles interfaces rétréci vers le bas notre choix de solutions. Produits commerciaux offrent des moyens à ontologicalize les services web. Nous soutenons qu'ils sont lourds (par exemple, les composants inutiles liés au produit) et ils sont coûteuse pour les projets à petite échelle, comme notre projet. Plusieurs solutions open source mature offrant de travailler avec des bases de données relationnelles ne fournissent pas ou peu accès aux services Web. Par exemple, aucun indice se trouve dans D2RQ se joindre aux services

web dans leur système, tandis que OpenLink Virtuoso répond aux appels de savon, mais ne peut pas gérer les données provenant des services web RESTful.

Nous proposons de construire un lien entre les ontologies et les services web. Nous trouver les métadonnées pour représenter les non-RDF services web dans l'ontologie, et nous revoir le code et créer de nouvelles structures de données en D2RQ à l'appui des requêtes ontologie à partir des données des services web RESTful.

Contents

Chapter 1 Introduction

The quest to translate data into knowledge, an eternal challenge to the Computer Science researchers, has achieved some milestones. Unstructured data, stored in documents and web pages, are crawled and tokenized to form inverted indexes. Boolean queries are parsed and compared to inverted indexes to produce a suitable result. Structured data, on the other hand, are usually stored in formatted databases. The database usually comes with a schema and/or metadata to give tables and attributes semantically meaning. Following the semantics of the schema, structured queries like Structured Query Language (SQL) are parsed and matched up to values of specific attributes. Two problems, nevertheless, will arise to the structured data, as a result of people's demand to manage the data. One problem is the semantic heterogeneity found while linking several data sources. For example, if users want to discover something from a number of important academic databases, which are schematically distinctive, it is not too complicated to create a satisfactory query scheme if similar terms of these databases have the similar meaning. However, semantic diversity among the database schemas multiplies the complexity of linking these data sources. Comparative effectiveness researchers in healthcare find this problem quite disturbing because they may learn of different interpretations of the meaning of the data from several sources or different terms actually refer to very similar meanings (El-Gayar 2010).

The idea of ontologies might offer an optimal solution to this problem. In computer science an ontology extracts a set of concepts and the relationships between those concepts within a knowledge domain (or across domains) (Gruber 1995). Concepts

are defined as Classes; whereas relationships are labeled as object properties. Data properties characterize features of the class, for example, an "age" data property of a "Person" class. Ontology is a formal representation of knowledge and provides a shared vocabulary to model that domain (or several domains). Particular meanings of terms may apply to that domain (or several domains). For example, in Figure 1.1 the "foaf:interest" property is defined as "A page about a topic of interest to this person". If users choose to impose a *Friend of a Friend* (FOAF) ontology (Brickley and Miller 2004) on their database, they agree to use foaf meanings and not to define the interest as something else (e.g., interest rate). If experts from a domain could work out a domain ontology and build the connection between databases and the ontology, then users only need the vocabulary from the domain ontology to query all the connected databases.

There are several formal languages to encode ontology. Resource Description Framework (RDF) provides the metadata for those formal languages. One of the most popular is the Web Ontology Language (OWL) (McGuinness and Harmelen 2004), which is also a descendant of RDF and RDFS (RDF Schema)--both are Semantic Web data models.

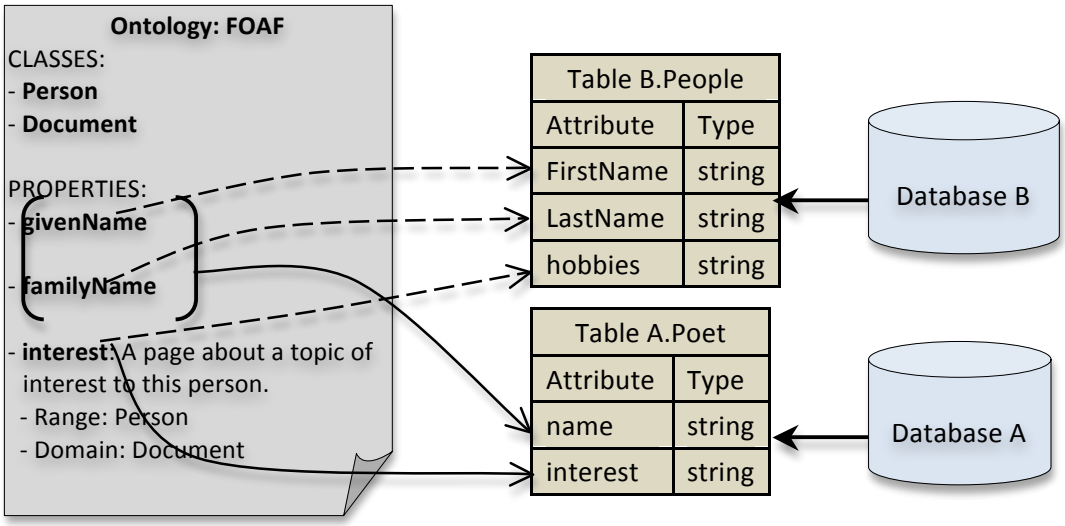


Figure 1.1. Mapping between databases and the ontology

Note that the FOAF ontology's two name properties – "givenName" and "familyName" are distinctively mapped to two equivalent name attributes in the People table of database B, whereas both of these two name properties have to be grouped to form the "name" attribute of the Poet table of database A. The issue is that even with a common framework, there is a lot of middleware that must be constructed to get the framework to speak to the databases themselves. And the "interest" property is actually mapped to a semantically similar attribute "hobbies" of database B.

Users query RDF/OWL with an ontology query language. There are many of these as well. The World Wide Web Consortium (W3C) recommends the *SPARQL Protocol and RDF Query Language* (SPARQL) (Prud'hommeaux and Seaborne 2008). SPARQL is defined an RDF query language that can write globally unambiguous queries and will be described in the next chapter. SPARQL is to the ontology (or in a greater sense, the Semantic Web) what SQL is to the relational databases. That means SPARQL is able to query ontologies with structured languages. However, SPARQL queries are based on triple patterns. Triple pattern represents a relation in the ontology. The subject and object of a triple are usually two connected concepts, while the predicate is the literal representation of the relation. The reasons SPARQL is

preferred by many Semantic Web groups and by me are its abilities to extract data from a giant collection of data structures and formats, including extracting data from other ontologies, RSS feeds, RDF, and XML (Bray et al. 2008).

Apart from the problem with linking different data sources, another problem regards the representation of data. People used to keep records of data in text and later in the databases. Now many database administrators choose to represent their data in Web Services, programming interfaces that are accessed by HTTP and processed at remote servers hosting the demanded services. There are mainly two kinds of Web Services – SOAP (Gudgin et al. 2007) and REST (Fielding 2000). SOAP is a protocol. WSDL is an interface definition that describes the content of the messages; the messages can be described within WSDL using XSD and thus structured. Both SOAP and REST based web services can use the HTTP protocol. REST based web services use the HTTP syntax to describe operations; however, the payload (messages) is not defined in a formal manner.

In comparison to the traditional connecting methods such as JDBC (Crawford et al. 2002), Web Services emphasize the programmability of the services provided by the data sources, instead of simply creating a direct connection to the data source and grabbing out the data. This new representation of data, coming with an increasing growth of deployment, has rendered many traditional applications obsolete. Researchers now need to reinvent new approaches to connect to the Web Services-based data sources and to utilize the new capacities of them.

My research question joins these two problems together – when the new method of linking different data sources meets the new representation of data sources, is it

possible to allow for queries of Web Services with the ontology-based query language? I want to create a bridge so that SPARQL queries could find their ways to HTTP Get-based RESTful Web Services. My question spurs from the project – Integrating across Space, Time and Gender in the Humanities for Chinese Literary, Historical and Geographical Databases (shortened to Integrating Chinese Historical Databases or ICHD) (Fong 2007). The data sources include Chinese Biographical Database (CBDB), the largest online relational database in recording Chinese biographical information regard officials and their kin; Ming Qing Women's Writings Database (MQWW), a pioneering online database on historical Chinese women's writings; and a RESTful Web Services-supported Chinese Historical Geographical Information System (CHGIS). Data queries cross historical, geographical, literary and cultural domains. In the larger project I created my own ontology that can answer questions to these domains. In this thesis I want to set up connections from the ontology-based query language to databases as well as RESTful Web Services.

As I will discuss in Chapter 2, existing tools like the D2RQ and OpenLink Virtuoso (OpenLink) do support connections from the ontology to relational databases. These tools either transform SPARQL queries into equivalent SQL queries or, turn relational databases into application ontology (i.e., an ontology which is derived from a database, and which resembles the database schema) so people could query it with a RDF query language. None of easily available or open source current software possesses the ability to include Web Services as a data source and to generate equivalent Web Services requests for SPARQL queries. Commercial vendors do support connecting SOAP and RESTful Web Services to ontology; however, they are far from my real need - I work within the realm of humanities research, which is

underfunded in terms of computational resources (Unsworth 2006; Short 2006), I need an inexpensive as well as an extensible solution. In conclusion, there is no easy way for people to create ontology directly from Web Services. Considering the spreading of Web Services on the Internet, this lack of connection would hold me back from applying the ontology to connect to heterogeneous data sources.

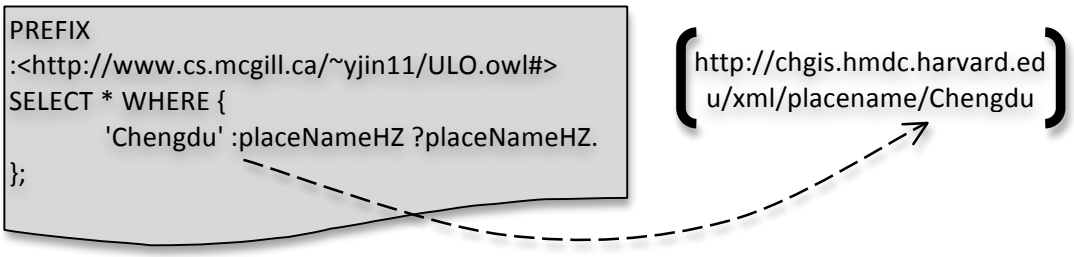


Figure 1.2. Sample input SPARQL query (left) and the output request for the CHGIS Web Services.

Figure 1.2 shows the product of what I am proposing, a new approach to bridging the gap, by transforming SPARQL queries into Web Services requests. Considering the complexity to effecting results from an ontology, for example, creating ontology models and parsing the SPARQL queries, I based my research on existing open source tools like the Jena Semantic Web Framework, D2RQ library, which is a tools to convert SPARQL queries to SQL queries, as well as some XML parsers. The Jena framework provides an SPARQL query engine and a programmatic context for RDF, RDFS and OWL. SPARQL queries thus could be easily parsed, modeled and saved in memory. D2RQ provides "a declarative language to describe mappings between relational database schema and OWL/RDFS ontologies" (Bizer 2003). The D2RQ platform also provides interfaces so that Jena APIs and SPARQL protocol could be embedded in the code. I take advantage of these tools to create a scheme that maps between the SPARQL queries and RESTful Web Services.

The rest of the thesis is organized as follows. Chapter 2 summarizes previous works in linking Semantic Web with different data sources. Chapter 3 describes the proposed system architecture, major open source tools, and the methodology I used to link Web Services. Chapter 4 gives the implementation details. Chapter 5 lists some running examples. The conclusion section features lessoned I learned from the building and thoughts I find helpful to continue the research.

Chapter 2 Literature Review

2.1 Overview of the Previous Researches

After the concept of ontologies was introduced in the Semantic Web world to integrate knowledge pertinent to domains, many researchers started to working on integrating the ontology with the existing or legacy data sources. Of this research, considerable effort was focused on semantics in text-based searches (Foltz 1996; Zhao and Grosky 2002; Cuenca-Acuna and Nguyen 2002; Mack and Hehenberger 2002; Short 2006), particularly in the field of humanities where my research is based (Borgman 2007; Barnard and Ide 1997; McCarty 2003; Gietz 2006).

My research question is how to design and implement a system that enables people to query Web Services with SPARQL queries. Specifically, how do I generate an equivalent RESTful Web Services request that could answer the questions in that SPARQL query? There are many ways to do this. I could transform the SPARQL query by some rules into Web Services requests. So for example, Zhao et al. (2008) tried to create specific rules that can replace triples in SPARQL queries to Web Services-related instructions, before transforming them into Web Services requests. Or I can "ontologicalize" Web Services so that by comparison with the query (which is written by the vocabulary of another ontology) I could find the shared parts. For instance, this thesis leverages the shared triples between the query and the ontologicalized Web Services so that triples in the query would be converted to Web Services requests. The Web Services ontology should then help transform the shared parts into Web Services requests. I can gain these ideas by studying related literature or those from similar areas.

The other alternative of my research focus is to turn SPARQL query into a standardized XML message, which is commonly acceptable by Web Services (e.g., SOAP). They do it this way because programs written in different languages on different platforms can communicate with each other in a standard way.

There are reasons I am not using XML. First, in either RESTful or SOAP Web Services, service providers only accept structured form of the web method and its input parameters (along with other service information, e.g., security configuration parameters). For example, any query regarding a region's name sent to the CHGIS RESTful Web Services should conform to http://chgis.hmdc.harvard.edu/xml/placename/QUERY-STRING with QUERY-STRING replaced by a specific value. Requests sent to SOAP server must also follow a structured form to include remote method and input values information (specified by a WSDL file), whether they are in the text, HTTP or the XML. What matters to the services server are the method's name and values. Second, the research question seems to me therefore, is how I should magically infer information from an SPARQL query (which has nothing to do with the Web Services yet) and choose the appropriate web method (among other methods provided by the Web Services provider) based on the previously acquired information. This information could be wrapped in an XML message and sent to the server, but they could also be loaded in a simple HTTP GET request. I believe the HTTP request is sufficient enough for the project, because it is simpler to implement than the XML option.

I begin by reviewing papers that connect databases to the Semantic Web. Databases serve the main data sources for storing data in the world now. Many advanced studies started from the database research, so do the Semantic Web technology (Broekstra et

al 2002; Chebotko et al. 2006; Cyganiak 2005; Pan and Heflin 2003). And if compared with research papers in Web Services in Semantic Web, literature in database is quite affluent in number and styles. For example, some research (Bizer 2009; Pan and Heflin 2003) considers the use of RDF to create semantics for databases. This shapes my research methodology. I then discuss some papers (e.g., Battle and Benson 2008) about including Web Services in Semantic Web. These researches are short in number but are very enlightening for their novelty in design. Creating links between geospatial Web Services and ontology forms the last part of my literature review. Even if geospatial Web Services (e.g., Web Feature Service (WFS) (Vretanos 2005) or Web Map Service (WMS) (Beaujardiere 2004)) are quite different from the Computer Sciences, they present good research methodology and are very helpful in thinking about ours.

2.2 Research on Databases

My investigation on databases focuses on the question – if I am presented an SPARQL query, what can I learn from the conversion of equivalent SQL queries for the database? There are numerous ways to do conversion and I focus on four (Broekstra et al 2002; Chebotko et al. 2006; Cyganiak 2005; Pan and Heflin 2003). Broekstra et al. (2002) coined the Sesame system. Sesame is RDF and RDFS based database engine. Users send RQL queries (RDFS Query Language) to the parser, which transforms queries to calls of SAIL APIs. The SAIL APIs are a set of Java interfaces that store and retrieve RDFS-based information. The APIs could be used on many data sources, for example, relational databases, file systems or in-memory storage. This is good in terms of its design and extensibility. It separates RDFS management from other data structures; the only computational work is the

transformation from RQL queries to the SAIL APIs. Since the APIs are consistent, it is theoretically possible to include any data source. However, the problem is the implementation – creating Java classes that follow the API standard from any data source. If I have dozens of different Web Services, it would be complex to develop all the implementations.

Chebotko et al. (2006)'s paper introduced two things – a basic and efficient algorithm to translate an SPARQL query's basic graph pattern (BGP) to SQL. BGP is the triple pattern in the WHERE clause of a SPARQL query without any other modifiers (e.g., GROUP or OPTIONAL). The algorithm is then escalated to process queries with OPTIONAL graph patterns. This paper provides useful details about SPARQL transformation. It points to the circumstances in which I should consider a triple's subject or object as the value for an SQL condition and how I should correlate a predicate in a triple with its graph as like a condition and its database table. Although the paper is about the conversion to SQL, it's implication on my research, of the use of subject and object (of a triple) as the condition for a database table is indispensable. I begin to think about the use of these in Web Services. However, RESTful Web Services do not have a standard query language like SQL and so this paper stops short of providing answers about SPARQL-to-Web Services translation.

Cyganiak (2005) proposed a SPARQL-to-SQL transformation algorithm that is based on relational operators. The semantics of SPARQL is used firstly to find out the relations in an SPARQL query (e.g., UNION, Projections, and SELECT). Triples and graphs of a query are all labeled with relations. Triples of the query are then replaced by relational operators attached to the relations, followed by graphs of the query. This algorithm is good because of its accuracy in translation (all the relations in the triples

are found and then translated to SQL). But it depends on a very relational perspective. RESTful Web Services is resource-oriented, which means it highlights the structure of the resources but not the relations between resources. If I can easily find the relations between resources then the semantics of the web services could be easily constructed. But it is not. What I learned from this paper is to design the SPARQL-to-Web Services from the point of view of the resources Web Services provide, instead of seeking relations. Can I relate triples in an SPARQL query to the resource of Web Services? The answer is yes.

Pan and Heflin (2003) take a quite distinctive angle at the use of databases in Semantic Web. They proposed to extend relational databases with the capability to do RDF storage. Users could query the database by applying inference rules on the RDF part of the database. The inference rules work by changing SPARQL queries to the RDF part of the database. This paper gives me some insight – I probably could reconstruct Web Services in RDF. which means I could create an RDF-based view for Web Services. The problem for me is, how I should design such an RDF-based view to express all the details of the web service? And because I have my own RDF – the ontology for the project, it is possible I could query the Web Services by querying the combination of the project ontology and the RDF version of Web Services. However, it didn't happen because the project ontology is too abstract while the RDF-based Web Services are full of dependencies on the Web Services.

These papers together contribute to the invention of mature solutions to bringing databases into the Semantic World. Among the popular, there are OpenLink Virtuoso (OpenLink Software 2010), BBN (Fisher et al. 2008) and D2RQ (Bizer and Seaborne 2009). And each one has its pros and cons. OpenLink Virtuoso is a virtual database

engine that "implement Web, File, and Database server functionality alongside Native XML Storage, and Universal Data Access Middleware, as a single server solution." (OpenLink Software 2010). OpenLink Virtuoso supports transformation of SPARQL queries to Web Services, partially. It is also capable of processing ontology-based queries against the database. Because the OpenLink Virtuoso is a native quad store, its strength is in its scalability and performance. Scalability is an important evaluation factor for the project because the number of my data could get quite large. According to OpenLink (*Ibid.*) its performance is good because it is "uniquely architected to address today's escalating Data Access and Integration challenges without compromising performance, security, or platform independence." (*Ibid.*) I don't use it because I am underfunded to employ such a commercial tool that provides many functions I don't need. And the project is proof-of-concept which means it does not place the performance as the first priority.

BBN's Semantic Distributed Query architecture is a similar application-ontology-for-database style (Fisher et al. 2008). It provides a lightweight structure in terms of the algorithm's simplicity – the computational difficulty is writing and parsing SWRL inference rules of ontology. SWRL inference rules are popular tools to create inference ability for OWL files. The so-called Automapper of this paper creates mapping ontologies of a data source with SWRL rules. It is however, integrated in BBN's proprietary software. For many small-scale projects with limited funding, BBN's product is too heavyweight and expensive. And well-supported cyberinfrastructure projects expect to have experimental functionalities, and are usually composed of many distributed servers. Shortage of tools to deal with these requirements is the striking weakness of BBN's product.

The third is D2RQ (Bizer and Seaborne 2009). Like Virtuoso and BBN, D2RQ is designed to provide RDF view over non-RDF databases. It generates "mapping files" from relational databases. A mapping file is application ontology. It creates an RDF version for a database schema. Each table in the database is shadowed as an ontology class in the mapping file, attribute of a tuple as a property of that class. The mapping file reconstructs primary keys and foreign keys, and it automatically creates linking classes for tables that can be joined. The general idea of D2RQ is to compare the input SPARQL query with the mapping file, which provides mapping information between ontology (vocabulary of which is used by the query) and the database. The result of the comparison, in addition with SPARQL-SQL transformation rules, would turn the query into SQL. D2RQ is good because not only it successfully distinguish itself as a reliable Semantic open source tools towards database, but also it sets up an example to connect ontology to other things, for example, Web Services. However, there are things D2RQ does not do well, for example, to support multiple database integration, which means, the system cannot intelligently link any two attributes from two databases even if they are semantically relevant. This is acknowledged as a failing in an evaluation work by the designer (Bizer and Cyganiak 2007).

These platforms have various connectivity to Web Services. Virtuoso supports Web Services but it only supports SOAP, not RESTful Web Services. D2RQ does not support any transformation of SPARQL queries to Web Services. If I am to complete this connection to Web Services, I have to look for additional research.

2.3 Research on Web Services

As more database and website administrators decide to provide Web Services, some researchers consider how to connect the Semantic Web and Web Services. In these cases, ontology is used to represent the concepts in Web Services. DAML-S is a semantic markup language that was proposed to marshal the concepts of ontology to describe contents and functions of Web Services (Paolucci and Sycara 2003). Some people thought WSDL (Christensen et al. 2001), a Web Services description language, is probably a good place for DAML-S to get a sense of the contents of Web Service. So, Ankolekar et al. (2000) proposed DAML-S to convert WSDL into ontology by describing what a service can do. This relates to my research because, WSDL determines what services SOAP would offer, and converting WSDL into ontology gives me some insight – is it possible to convert RESTful Web Services into ontology in this way? Unfortunately the answer is no. RESTful Web Services do not rely on some descriptive files for their services. OWL-S was developed to replace DAML-S. In OWL-S, Martin et al. (2004) proposed the use of three ontologies - a profile ontology, used to describe what the service does; a process ontology and corresponding presentation syntax, used to describe how the service is used; and a grounding ontology, used to describe how to interact with the service. I gain from these papers approaches used to convert SOAP to ontology extend my knowledge about methods to bring Semantic Web and Web Services, but those cannot help me in "ontologicalizing" RESTful Web Services.

Battle and Benson (2008) linked the Semantic Web and Web Services at the level of access to data and services. In their paper, they develop two infrastructure elements that will use REST to implement Semantic Web applications on top of existing or

new services. REST is a useful architecture because it offers equivalent services as SOAP but usually cost less in communications. And because of the lack of an explicit contract (as in an interface definition) in REST there is a certain degree of implicit coupling between the provider and consumer. This can lead to a degree of brittleness. The first element, the Semantic Bridge for Web Services, enables Semantic Web developers to execute SPARQL queries against existing web services. It does so by wrapping the WSDL and OWL-S or Web Application Description Language (WADL)(Hadley, 2006) description of Web Services operations, and translating the results returned into the SPARQL query result format. WADL is an XML-based format to provide a machine-readable description for HTTP-based REST Web Services (*Ibid*.). The second element, Semantic REST, is a protocol intended for new Semantic Web applications that need to support REST-style access to query and data manipulation functionality. Battle and Benson (*Ibid.*) show how to extend the capabilities of the existing SPARQL protocol by supporting updates and deletes.

This approach is a good example considering I know the semantics of the CHGIS Web Services from by working with Berman et al.(2008). The issues for creating an ontology of Web Services are then how I derive semantics from Web Services and, how to design a ontology schema that describes all the semantics of the CHGIS Web Services. The semantics of Web Services are derived from the knowledge domain not IT infrastructure. That includes D2RQ, which technically is building a mapping file and not an ontology. Unless I have a really good AI (which does not exist) I cannot determine the semantics of a web service (either WSDL or REST based) just from the web service interface. So the semantic information is introduced by humans. The solution to the second question will be elaborated in Chapter 3.

2.4 Research on Geospatial Web Services

The computational geography community, as pioneers in working with geospatial Web Services (Andrews 2007), could bring me some thoughts. Few years ago, the Open Geospatial Consortium (OGC) has developed its own standard of Web Services to retrieve raster map images and/or vector based geographic features from geospatial data sources - Web Map Service (WMS) and Web Feature Service (WFS). They are the most popular Web Services employed by the geospatial Semantic Web projects (Hobona et al. 2007). One of my datasets is geographic. A few research papers describe how SPARQL queries are transformed to geospatial Web Services requests, although most do not offer more details than the documentation of the project. For example, Paul and Ghosh (2006) argued that domain ontology could be used to provide shared vocabulary for the schemas of WFS servers. Domain ontology is a specification of terms used to model a specific domain. Queries to a service broker, which maintains a list of services, can be translated to specific WFS getFeature requests to different service providers such as WFS servers. Details are not provided, though. Li and Yang (2008) proposed a semantic spatial search engine architecture that can crawl WMS automatically. The GeoBridge is the most important component in the proposal. It parses queries from the search engine, and assigns the query to specific WMS. The WMS are provided by multiple geospatial data sources. How the system is built is still unknown. The SPIRIT spatial search engine (Jones et al. 2004) has shown ontology to be useful in searching web documents with spatial content. User queries can include a subject, a place name, and a spatial relation to the place name. Results are a list of documents and their positions on a map. The search engine uses geographical and domain ontologies to distinguish and expand user queries, to rank documents based on the relevance to the query, and to extract metadata from

web documents. I do not really learn too much knowledge related to my research focus from these papers, however, they show the fact that even though the geospatial community is an eager fan of Web Services but they have shown more interests in the consumption of this technology than in exploring research questions in this field.

Some researchers in computational geography are conducting studies with a similar purpose to ours – to query Geospatial Web Services. Zhao et al. (2008) proposed the query rewriting techniques to refactor the SPARQL query, so that part of a query could be transformed to WFS getFeature requests during parsing. The rewriting rules, also called inference rules, are defined in the ontology that provides the one of the vocabularies for the SPARQL query. When the parser identifies several (or a single) triples of a query that match a pattern in the ontology, it replaces these triples with a corresponding WFS getFeature request. They also mentioned the ability of their system to query relational databases was based on D2RQ. So in conclusion, they created a system to handle queries to both relational databases and WFS. This query rewriting approach might be feasible theoretically, but it is subjected to two problems. One, such a proposal cannot be deployed to large-scale projects. The manual writing of inference rules would take massive time. Also, developers have to be trained to be knowledgeable in the structure of databases, WFS, as well as ontologies in the first place, and then they can begin to work. In addition, changes to the structure of any data sources would possibly render these inference rules obsolete. The second problem is about its integration with D2RQ. D2RQ offers parsing queries to relational databases, but in a very different style from Zhao et al. (*Ibid.*). Unlike the inference rules-based query rewriting, D2RQ transforms SPARQL to SQL by comparing application ontology generated from the database with the query. If Zhao et al. hope

to bind D2RQ with their own proposal; they have to change either their code or D2RQ's dramatically. A system like this is technically difficult to prove applicable in reality.

Even though WFS/WMS prevails in the geospatial community and I utilize the Web Services of a geospatial database, I am interested in creating a connection between ontology and the RESTful Web Services. First, RESTful Web Services combine certain advantages from both WFS and WMS. RESTful Web Services enables any feature queries (WFS) based on a structured URL (WMS). And it reduces the deficiencies. WFS suffers from a rigid form of request information. For example, one could get geographic feature information via its regulated XML format. But it is difficult to carry extra filter information with the getFeature, especially in terms of temporal perspective. RESTful works as WMS to allow for the addition of extra filter parameters to optimize queries. Second, many geospatial application developers find WMS and WFS difficult to use. The creator of Open Street Map, which is a volunteer driven street map of the world (www.openstreetmap.org), has commented, "WFS has some good points, but overall, it is overly complicated for a lot of use cases. The primary transport being GML is probably one of them. Complete lack of examples of how to interact with it, the complexity of write support, etc. all adds to it" (Schmidt 2009). Third, WFS requests are not a resource request but a remote request procedure call. The design of WFS regards resources as in abstract parcels. Several resources should be integrated in one function if they serve the same purpose. Developers may not be able to understand the structure or the relationships of the resource. RESTful Web Services features a clear structure of resources. People could easily get a sense of what their resource is and where the expected resource will be, from the semantics

of the URLs. Finally, it should be argued that WMS and WFS were not designed to follow best practices in web principles, and "now unfortunately is harnessed with so much momentum (vendor, industry, 'architecture enforcement') that it will be difficult to resolve" (Turner, pers. Comm..). I believe a focus on RESTful Web Services for geographical data will help researchers keen on this new technology build their own applications.

In this chapter, I have studied the literature on associating databases, Web Services as well as geospatial Web Services with the Semantic Web world. This literature review leads me to look for a relevant approach to moving from an ontology-based SPARQL query to an HTTP GET request of RESTful Web Services.

Chapter 3 Methodology

The objective of this methodology section is to guide readers through the process in which I made effort to bring the Semantic Web and Web Services together. This chapter concentrates on the design issues. The implementation portion of this effort is in Chapter 4.

I start by modeling the system architecture. The model will be outlined with discussions on the functions of each component, the use of open source tools, the design of the schema file that drives the parsing of SPARQL queries, as well as data structures that hold parsed information and facilitate the transformation work.

This work is part of the Integrating Chinese Historical Data (ICHD) Project. The goals of the larger project are to bridge the gap between computing and Chinese humanities, to create new research possibilities to study historical Chinese women writers and to encourage the use of geographical, biographical, literary and computing knowledge in the humanities domain (Sieber et al. forthcoming). The project can currently be found at http://linuxdev01.geog.mcgill.ca/gui/draft9/.

3.1 System Architecture

Before I begin, I want to outline the steps that need to be taken. Initially, the system is supposed to work in the following way - the user writes his/her question as an SPARQL query and sends it to the system, which transfers it right into RESTful Web Services requests. The service provider then responds to the requests with structured results.

However, except for well-trained researchers, ordinary users have to rely on the SPARQRL generator to create queries for their input. Both of the knowledge from Web Services and other databases (that are parts of the project's data sources) should be conceptualized to form a project ontology, which again provides the vocabulary for the SPARQL generator to work. The Web Services has its own application ontology that includes information about mapping ontology terms to Web Services ones. Application ontology is a specification of Web Services with RDF terms. The application ontology instructs the Query Broker, which receives SPARQL queries from the SPARQL generator, how to translate queries to corresponding requests.

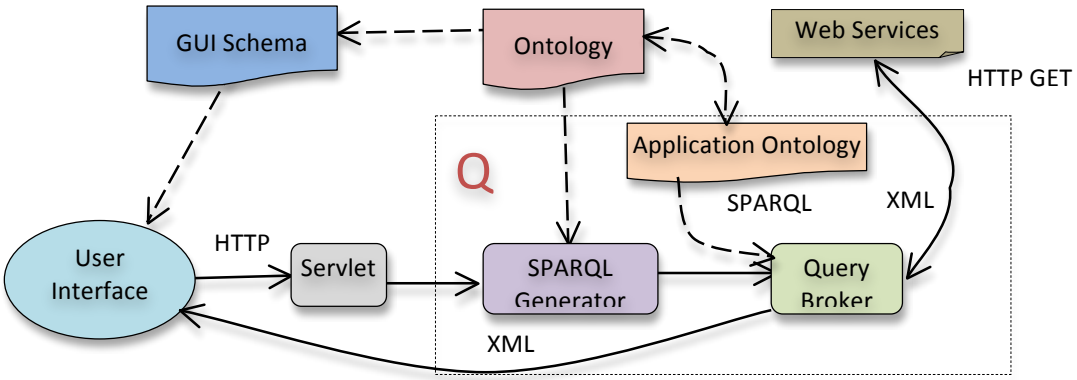


Figure 3.1 Abstract model to build the bond between ontology and Web Services.

Figure 3.1 shows some basic components of my model. My research focus is in the Rectangle Q. The other components are mentioned towards the end of the chapter. The core of the design of this architecture derives from two key points in Figure 3.1 - the ontology in the top middle and the Query Broker in the right bottom.

As I mentioned previously in this section and the Introduction section, ontology is composed of concepts and the relations between them found in the Web Services and databases of the project. It serves three purposes. First, the content of the Graphical User Interface (GUI) is dependent on the classes (i.e. concepts) and properties (i.e.

relations) of the ontology. The tree-like interface is iteratively constructed in Flex from the structure found in the ontology. This explains, to some extent, the logic of the Ontology-GUI-Schema-User's Request line. On the project's portal website (http://linuxdev01.geog.mcgill.ca/gui/draft9/), users are prompted to select concepts (e.g. "Person", "Region" and etc.), which are provided by the GUI schema file. A GUI schema file tells the user interface which concepts in the ontology are searchable. (There are non-searchable concepts, which are abstract terms like "Birth" or "Marriage". Users find these kinds of information by searchable concepts. For example, you could find "Birth" information by query a "Person"s "hasBirthDate" and "hasBirthPlace" properties.) The schema file is constructed to include the structure and contents of the ontology. It differs from the ontology at the schematic information that allows the user interface to compose a structured message to the Servlet (and the SPARQL generator). The Servlet shares the schema file, reads and regroup user's questions so that the SPARQL generator can use them. This is not within my research focus.

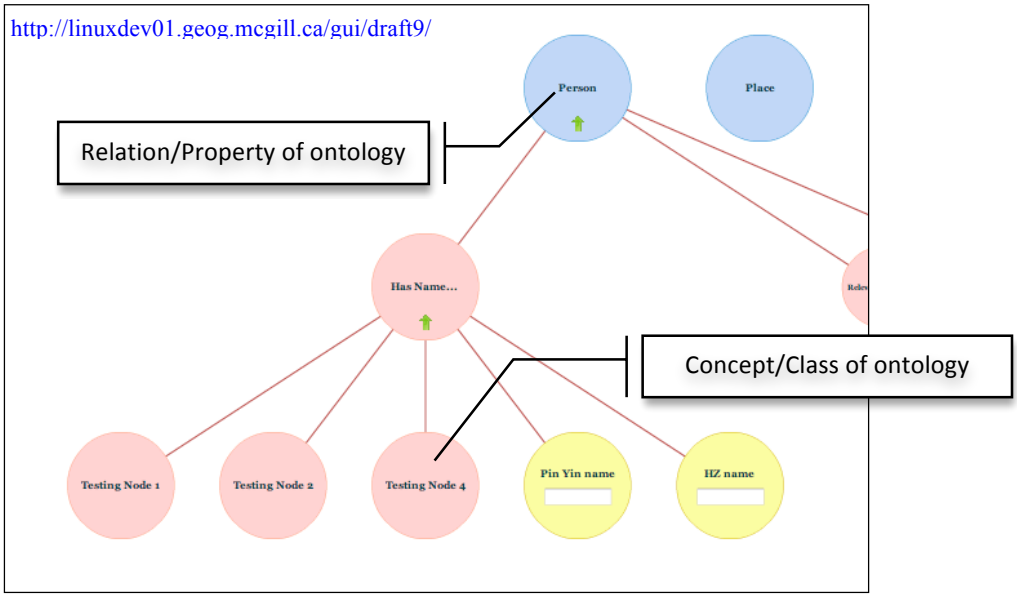


Figure 3.2. A simple ontology-driven user interface.

The blue bubble indicates the concept “Person” from the ontology; the red bubbles represent properties, e.g. “Name” of a “Person”; the yellow ones are specific properties of the “Person” concept.

The second purpose that ontology serves is to help the SPARQL generator in query making. First, when the Servlet reads user’s information from the user interface, it stores the data in an appropriate data structure, because later on the SPARQL generator could find the necessary data quickly. This needs the assistance of the GUI schema file. And then the SPARQL Generator would translate the data structure to SPARQL queries because the Query Broker that handles the query-to-Web Services request work, is designed to accept SPARQL queries. This is aided by ontology. For example, when a user chooses “Person” by “hasNamePY”, that is internally translated into, “Give me all the information related to this person X” by the ontology file. Ontology decides what information should be returned to the user. The ontology has a “Person” class (or concept, interchangeably), for example, and it will identify all the properties (or relations, interchangeably) that are used to describe the person. This explains the Ontology-SPARQL generator and Servlet-SPARQL generator lines. Nevertheless, there are other issues on how to do the query generation. I will discuss it in Section 4.1.

The third point of ontology in the project is to update application ontologies. And this somehow explains my design of the system. Application ontologies are created as a specification for all the details in a web service, for example, the connection string, the username, the web method’s name and parameters, and so on. It simulates the Web Services with RDF terms (I will explain this later) but it is short of the sense of abstraction. The ontology file, in comparison, is so abstract of concepts and relations

that the database and Web Services cannot understand it anymore. For example, the ontology file has a “Place” concept, which could mean quite a few more things than a “location”. Web Services have a “region” function (or method, interchangeably), which I think they are semantically equivalent. So I have to update application ontology with such equation information. The Query Broker would then know that the “Place” concept used in an SPARQL query refers to the region function. This is essentially the logic of my paper to connect ontology and Web Services. This explains the Ontology-Application Ontology-Query Broker line in Figure 3.1.

The other key point in the design of this system is the Query Broker. The Query Broker decides how an SPARQL query is finally transformed to Web Services request. In the ICHD project, the Query Broker utilizes D2RQ and Jena libraries and thus is able to transform an SPARQL query to SQL queries as well. The SPARQL queries are the output of the SPARQL generator and are the input of the Query Broker. The Query Broker completes the transformation with the help of application ontologies. The expected outputs of the Query Broker are two things. One is the result of the transformation – an HTTP GET request that will be sent to the service provider. The other is structured result that will be used by the user interface. Usually the Web Services provider will return the service clients with some formatted results. However, the GUI cannot use these results immediately – they are either intermediate data that will be used by other queries, or they should be merged with other results (e.g. from database servers) and regrouped to present to the GUI. This paper will explain the Web Services part of the transformation and the result processing.

The last part of the Query Broker is the result processor. Different Web Services have different ways of data export structure. It is therefore important for me to know what

values are returned from the provider and how I should retrieve required data from the result. And even the format of the returned result would vary, but I will focus only on the XML in this paper for simplicity. A result schema should define the way the result stored so that the GUI or the Servlet developer knows what information is discovered. These design issues will also be found in the last sub-section of this section.

3.2 Choice of Tools

Many tools are used to develop this system. The coding platform is Java, Standard Edition (Sun JDK Update 7). The project is developed with Eclipse Ganymede (3.4) on a Microsoft Windows XP (SP2) system. I used Free Open Source Software (FOSS) a lot in the project. For example, I simulated the CBDB and MQWW databases in my local servers running MySQL 5.1, because those two remote databases are running with similar configurations and they are too slow (in speed) and limited (in functionality) for development use. I have also used Apache Tomcat 6.0 to host simulated CHGIS Web Services on my local server. I used IBM-supported Netbeans IDE 6.5 to create sample RESTful Web Services. I focus on three major tools that affected the design and implementation of the system. The discussion will be carried out in terms of the reason I employ them, the functionalities I used and the influence they have on the system.

There are three FOSS I have used – the Jena Semantic Framework, the D2RQ Platform, and JDOM. Jena mainly serves as a SPARQL parser that translates SPARQL queries into a query data structure. Jena relies heavily on ARQ, an SPARQL processor for Jena. D2RQ partially serves as the parser for the application ontology (I will talk about why it is partial). JDOM is used in the result processing.

D2R serves as the Query Broker, although as I will discuss in the next chapter, it required significant modification for my goals.

3.2.1 Jena Semantic Framework

Jena is an open source product by the HP Labs Semantic Web Programme. It provides "a programmatic environment for RDF, RDFS and OWL, SPARQL and includes a rule-based inference engine." (Hewlett-Packard Labs 2010) It is a popular tool to build Semantic Web applications (Hebeler and Fisher 2009).

The rationales for utilizing Jena are twofold. Jena is one of the most powerful tools in Semantic Web – it provides all the necessary technical functions I need, which are RDF APIs, reading and writing RDF in N3 (Beckett and Berners-Lee 2008) and a SPARQL query engine. It holds a strong development community and numerous open source and enterprise applications are closely related to Jena. That means that if there is a problem with Jena, A future developer of the system (or I) could rely on the user community to assist. The downside is the frequency with which updates are made to the framework, updates that could make it incompatible with other FOSS components. The Application Programming Interface (API) that Jena provides is also developer-friendly, an example of which is writing a Jena RDF model. The functions and the data structure are very intuitive to understand and resemble writing XML very much, which saves developers a great deal of time to study. This makes Jena fundamental for another FOSS I use – D2RQ. This contributes to the second motive I am in favor of Jena – better integrative ability than other solutions.

The power of Jena is embodied in its abilities to manage OWL, to provide in-memory and persistent storage and an SPARQL query engine that parses inference rules. For example, my ontologies are written in OWL. The SPARQL generator needs OWL APIs to find all the properties related to a concept. For the Query Broker, I want to limit Jena to its capability of parsing SPARQL queries. This is actually supported by an important part of Jena, ARQ, developed by the W3C RDF Data Access Working Group.

Choosing a framework, as noted above, comes with benefits. But it also constricts me. The choice of Jena has created some impact on my system. For example, I have to accept the data structure Jena creates for me. This limits the number of options I could choose for the library data structure, which is a data structure to hold application ontology. Since in the end I want to compare the query data structure and the library one, it is essential to design a good and flexible comparing framework but now Jena, to some extent, shapes the way I design. Nevertheless, the efficiency and accountability Jena gives me let me focus more on the data structure generator. And because Jena is initiated from an enterprise project, coding conventions and documentation are quite satisfactory for other developers. In addition, Jena's open source community is now worldwide which means I could get quick and various responses from other members instead of exploring "in the dark" by ourselves. This accelerates the development progress of my system.

3.2.2 D2RQ Platform

According to the official definition, D2RQ is "a descriptive language to describe mappings between relational database schemata and OWL/RDFS ontologies." (Bizer

2003) The platform on which D2RQ is running provides applications the ability to access non-RDF relational databases in a RDF view. It supports access via Jena APIs and SPARQL queries.

The reason that I use D2RQ results from the selection from a wide-range of FOSS. Initially, the ICHD project needs to provide a single query interface for three semantically different remote databases. When I decided to leverage the ontology as the medium to integrate knowledge from these data sources, I was faced with the problem to provide connection from the SPARQL queries to the Web Services and databases. There are tools to deal with relational databases - D2RQ platform and the OpenLink Virtuoso are among the best. I decided on D2RQ platform rather than the OpenLink Virtuoso because it is more intuitive to match my needs and D2RQ platform provides more mature solution. Besides, D2RQ platform is not only available for relational databases, but it could serve an important role for building the connection between SPARQL queries and Web Services as well. D2RQ regards databases in RDF-view and I too, regard Web Services in RDF-view. I was also guided by Zhao et al. (2008)'s research on geospatial databases; one of my databases is a geographic information system (GIS).

D2RQ platform provides some partial function in the transformation of application ontology to library data structure in my system. The library data structure connects the ontology and something else, e.g. relational databases.

The influence D2RQ platform exerts on my system cannot be overstated. It teaches me that by comparison between the query and the application ontology, I am able to generate SQL queries (the instructions to create which are embedded in the

application ontology). It affects the way I turn SPARQL queries into Web Services requests. It is enlightening in one way, but it also prohibits me from creating brand-new approaches, because I rely my ontology-to-relational-database connection on D2RQ platform. And I have to combine my trick about Web Services with D2RQ so as to create a stable and reliable system for both relational databases and Web Services.

3.2.3 JDOM

I need a tool to do XML parsing and to create XML files. My chose is JDOM. The JDOM is an open source document object model for XML that was designed specifically for the Java platform. It integrates the features of both Document Object Model (DOM) and Simple API for XML (SAX). JDOM "provide a complete, Java-based solution for accessing, manipulating, and outputting XML data from Java code" (JDOM 2010).

There are a myriad of open source XML manipulation tools, for example, Xerces, dom4j, JAXB, VTD-XML and so on. Each of these boasts its own features, like speed and memory saving, or support for numerous XML protocol. I need an XML tool to parse the outputs from the Web Services provider (Query Broker-Web Service in Figure 3.1) and to reorganize the data in XML so that GUI or the Servlet could utilize the result. My system is proof-of-concept and so there are not excessive data to process. All I need is stable and mature solution for XML manipulation and, that is what JDOM offers. It is lightweight and fast, providing very clear-cut APIs, and is optimized for Java applications (Java-Source 2010).

XML manipulation tool is an important part of the result processing because I need to parse the XML file containing the result (sent by the service provider) and creating an XML file for the user interface. Different Web Services providers will return structurally different results, at different times. The XML parser should know exactly what information is in the result so that they could retrieve the needed more quickly. After they are fetched from the returned results, the data is still in a mess without a recognizable format known to the user. So a returned result schema should be used to coordinate the result reporting. It is when the XML manipulation tool is put into use again to form a structured XML file for the GUI.

And fortunately, the use of JDOM has no considerable impact on the design of my system like the previous two FOSS. And I hope the introduction to the three FOSS I have used in the system help understand my work.

3.3 Query Broker

As I mentioned before, the major challenge is that the ICHD project is taking D2RQ platform to talk to databases, but D2RQ doesn't have a corresponding model to deal with Web Services. I'm proposing this Query Broker that could fix the problem.

In this section, I am discussing the design of the Query Broker. I will give out the overview of the architecture of the Query Broker in the first place, by which you'll be familiar with some basic ideas that turned an SPARQL query into Web Services. The overview will also briefly describe some important components of the architecture.

There are another two sub-sections. The data structure sections will cover discussions about the query data structure, which turns an SPARQL query into an internal data

structure; the library data structure, on the other hand, turns application ontology (I will introduce application ontology in details) into internal data structure. I also include a small section about the design of the schema file to regulate the design of application ontology. The second sub-section is about the transformation of internal data structures into Web Services requests – it explains the transformation at design level.

3.3.1 Architecture of the Query Broker

Figure 3.3 shows the basic components of the Query Broker. The top left two text components are the inputs to the Query Broker. The one on the top is a simplified SPARQL query that tries to retrieve the latitude information from the CHGIS Web Services by query a geological name (e.g., "Chengdu" is the provincial capital city of Sichuan Province, on the southwest of China). I will cover the generation of SPARQL queries in Chapter 4. The one below the SPARQL query is the application ontology written for including the CHGIS Web Services. The application ontology itself is defined by a schema file, which serves as the metadata to indicate what information should be contained in the application ontology. For example, because the application ontology assumes the duty to tell the Query Broker how to transform a query to Web Services request, it should include details of a web service, equivalent components that can be used to compare with a query's components (e.g. predicates), and the mapping information between the former two. I will introduce the design of the application ontology and the schema later in this section.

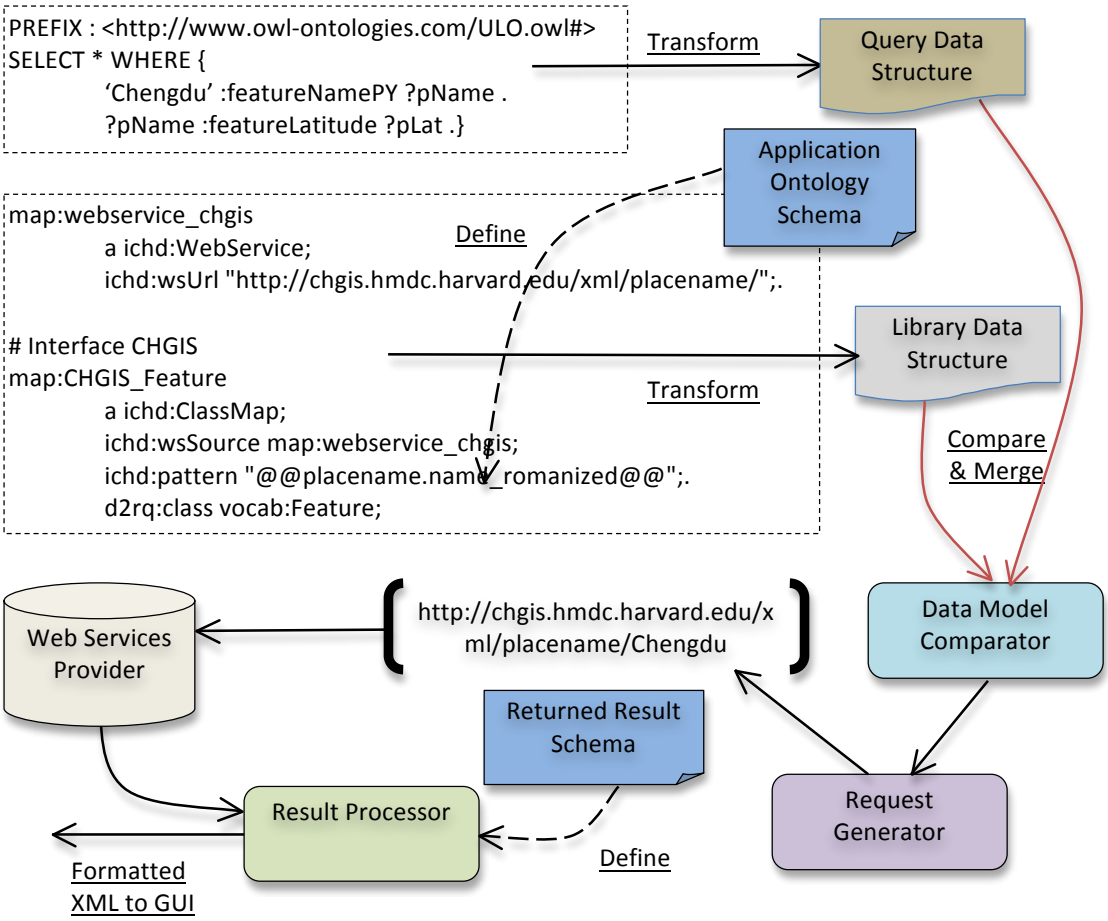


Figure 3.3. Architecture of the Query Broker.

As we can see in Figure 3.3, two inputs to the Query Broker should be internally modeled. The SPARQL query will be transformed into a Jena-conformed structure, the query data structure. The application ontology will be imported and parsed to form the library data structure. The power of the transformation (from SPARQL queries to Web Services requests) then comes from the integration of these two data structures. I will introduce these two data structures and the transformation later in this section. The implementation details are further described in Chapter 4. Note that the figure is that http://www.owl-ontologies.com/ULO.owl is a fake URL only serving a text identifier for the Query Broker.

The core of the Query Broker is the data structure comparator and the request generator. The data structure comparator compares data from the query data structure

and the library data structure, merges and optimizes them. The request generator then transforms the merged data structure to RESTful Web Services requests. The request will be sent via HTTP Get to the Web Services provider to process. In Figure 3.3 a sample request that calls the "placename" method of the CHGIS Web Services. The method receives geological name in the Chinese history in Chinese characters or Romanized region names. The major problem for the data structure comparator and request generator is to understand which queries belong to a specific Web Services method, what input parameters of the method are and how I could find them to feed the method.

3.3.2 Data Structures

3.3.2.1 Query Data Structure

When the SPARQL query is introduced to the system, it is still in the text form. Without further development to internal data structure, the system has no way of comparing it with the application ontology. There are many approaches to do this. However I rely on the Jena Semantic Framework to model the SPARQL query.

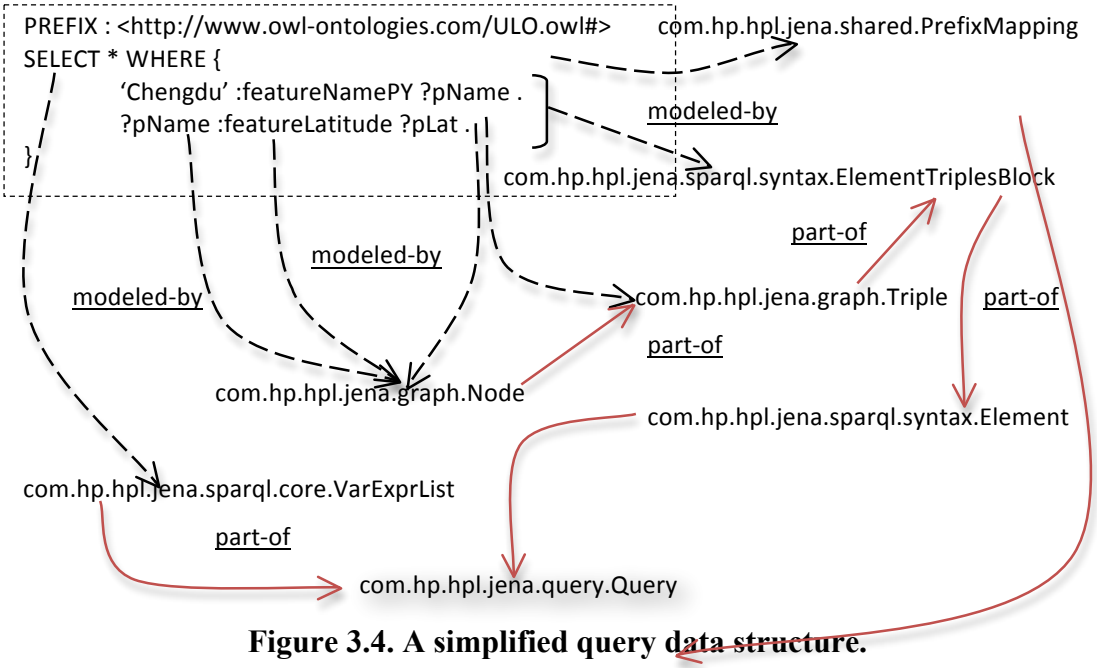


Figure 3.4. A simplified query data structure.

The feature of this query data structure (Figure 3.4) is it is hierarchical. The SPARQL query itself is created as an instance of the Query class (of the Jena Semantic Framework), while the prefixes, the return variables, the conditions in the WHERE clause are all modeled respectively and are pointing to the Query instance. Each condition (triple) of the WHERE clause is a Triple instance, the subject, predicate and object of which are Node instances. Triples instances of a graph (a graph is a block of conditions in the WHERE clause) constitute an ElementTriplesBlock.

3.3.2.2 Library Data Structure

Quite similar to the process of transforming an SPARQL query into an internal data structure, this section aims to converting schematically self-designed application ontology into an internal data structure. As I have discussed before, the benefits of doing so are to facilitate the Query Broker to compare the SPARQL query with some library so as to generate Web Services requests. The difference of this section from the last one is the introduction to the application ontology schema file. The design of the schema file highlights my solution to “ontologicalize” Web Services, which means it depicts my understanding of Web Services in RDF perspective.

In this section, I will rely heavily on the Jena and D2RQ libraries. Jena provides support in the N3 Turtle parsing but again it depends on me to tell the parser what to do given the application ontology schema file. The D2RQ appears even more important in this section because it set up the example for building the library data structure. In many cases I could follow the way it manages how to structure information from the application ontology and to assemble them into triple-based data structure. But the difference is it handles relational databases but I am doing Web Services. Not only the way to design the schema is different, but the data structure to

save mapping information is also varied. As the result, the processing logic to extract data from the application ontology is distinct.

There are two principles regarding the design of a schema file. First, the schema file should be able to describe RESTful Web Services. The use of a schema file is to regulate the way application ontology (a.k.a., mapping file in the D2RQ) is written. The application ontology creates a RDF representation of the Web Services with some vocabulary and the style of writing. The schema file dictates the metadata and the style of writing such ontology, so that other developers using this system could write RDF Web Services themselves. The other principle is that, the schema file should leave room for mapping information. Remember that the application ontology is to provide the mapping information between RDF terms (ontology terms) and the methods (or, the "web services"). This means that, if the triples (RDF terms) from an SPARQL query are fed to application ontology, and if they are found to be in the ontology, they should be mapped to certain web methods previously defined. This is the so-called "mapping". The design of the schema file should also represent such ideas. In short, the schema file should record the gist of the Web Services in the RDF form and the relations between web services and the RDF terms. The application ontology is therefore considered an instance of the execution of the schema, to some extent.

The schema is recorded in XML format. It provides descriptive metadata designed with help of other RDF terms (to name a few, RDF, OWL, DC (DCMI 2010) and etc.). The benefit of doing this is other RDF parsers are thus able to understand the information it wants to express. The recording format for the application ontology, as I recommend, is N3 Turtle, but it could be in other forms if only it follows the

metadata in the schema. The choice of N3 Turtle is its expression is intuitive. The basic unit of N3 Turtle is a triple, which is not only intuitive for the RDF developers but also very natural to process with SPARQL queries because they are also triple-based. There are some rules to write N3 Turtle and the XML, but I am not going to cover them in the text. The literature could be easily found online.

Considering all the demands I have to meet, I conclude the schema file in fact needs to provide information for two ends – the ontology and the Web Services. I should have pointers inside the application ontology that connects to some vocabulary of ontology used in the SPARQL query, and I should model all the necessary information to describe methods in Web Services and the connection information. In general, I propose four types of information that should be addressed in the schema. It should need connection details of the Web Services. At least, this contains the URL address of the service, user name and password. There are indeed some security parameters needed in the connection in reality, but I want to keep the model simple right now so, I only focus on the indispensible information. Then I want to describe the web methods, namely, the method's name, input parameters, and the types of the parameters.

One important point concerning the description of the web services is how to represent the return values. Web Services could return a simple sequence of text (including strings, numbers, Booleans and so on) as simple as the general Java method. But some Web Services provide a structured file, like an XML file, back to the service consumer. The XML file would enclose a collection of data. For example, if the user asks a name, the data provider gives a list of related information, e.g. gender, marriage, birth, employment, and so forth. This is a very efficient way to

provide information. And one of the project's data providers, the CHGIS, does so. It is imperative for me therefore, to create a schema suited for such a need. I want to model the request and the returned result in the following perspective. Figure 3.5 gives more insightful portrayal of such a perspective.

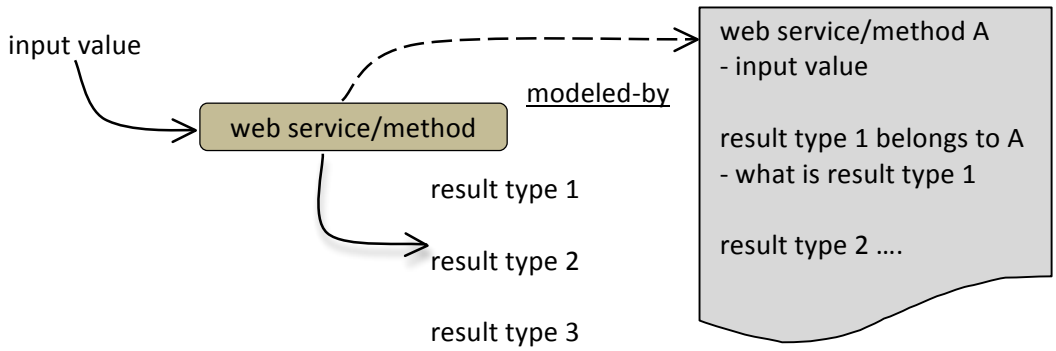


Figure 3.5. A way to model the Web Services

The proposal is to regard a web service (i.e., a method) an independent entity in the schema, with the input parameter as its dependent entity. All the returning result types, for example, gender and employment to a name request, are considered independent entities as well, but they should be connected to the web service. The example in Figure 3.5 renders the idea. The "what is result type 1" denotes the metadata to describe the result type 1, for example, to describe the data type of the result type 1 or, the ontology term corresponding to the result type 1. I will see a more concrete example from the CHGIS Web Services in the end of this section.

The third type of information that should be formed in the schema is the descriptive parts of web methods. Input value is one of them. The input value should give a pattern indicating what kind of information is accepted by the method. Besides, a pointer to the connection information of the Web Services should be included in the descriptive parts because there might be several distinct Web Services available. One last piece is mapping information to ontology. For example, the gender entity

probably has an equivalent predicate (a.k.a., property or verb) in the ontology vocabulary, called <http://www.owl-ontologies.com/ULO.owl#>hasGender (the URL indicates the location of the ontology it is defined). I want to tell the parser that when it encounters the hasGender predicate in the triple, generated from the query, it could map it to the gender entity related to the web method A. This also applies to the web method A. It may have its own equivalent class (a.k.a., concept) in the ontology. The mapping information is the fourth type of information I need to define in the schema.

I want to give a minimum set of the most often used metadata defined in the schema (Figures 3.6, 3.7, 3.8 and 3.9) and a simple example application ontology generated from the CHGIS RESTful Web Services. In essence, the schema file is a application of the D2RQ protocol in the field of Web Services. The rest of the schema file could be found at http://www.cs.mcgill.ca/~yjin11/ICHD.xml.

Items related to setting up the connection to RESTful Web Services

```
<rdfs:Class rdf:about="http://www.cs.mcgill.ca/~yjin11/ichd.xml#WebService">
  <rdfs:label>Web Service</rdfs:label>
  <rdfs:comment>Represents a Web Serivce node</rdfs:comment>
</rdfs:Class>

<rdf:Property rdf:about="http://www.cs.mcgill.ca/~yjin11/ichd.xml#wsUrl">
  <rdfs:label>web service url</rdfs:label>
  <rdfs:domain rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#WebService"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.cs.mcgill.ca/~yjin11/ichd.xml#wsUsername">
  <rdfs:label>web service user name</rdfs:label>
  <rdfs:domain rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#WebService"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.cs.mcgill.ca/~yjin11/ichd.xml#wsPassword">
  <rdfs:label>web service password</rdfs:label>
  <rdfs:domain rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#WebService"/>
</rdf:Property>
```

Figure 3.6. A minimum set related to the connection to RESTful Web Services.

Here is the explanation to Figure 3.6. The right part of rdf:about indicates the metadata that should be written to construct application ontology. The rdfs:label denotes the official name of the item, while the rdfs:comment means the documentation related to this item. The rdfs:domain is read a constraint for the RDF entity. For example, the rdfs:domain for wsUrl is rdf:resource WebService, which means it'd only be used with Web Service. This is reasonable since the URL for a web service could be comprehended as a property for it.

Items related to the major entities

```
<rdfs:Class rdf:about="http://www.cs.mcgill.ca/~yjin11/ichd.xml#ClassMap">
  <rdfs:subClassOf
rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#ResourceMap"/>
  <rdfs:label>Class map</rdfs:label>
  <rdfs:comment>Maps an RDFS or OWL class to its database
representation.</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.cs.mcgill.ca/~yjin11/ichd.xml#PropertyBridge">
  <rdfs:subClassOf
rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#ResourceMap"/>
  <rdfs:label>Property bridge</rdfs:label>
  <rdfs:comment>Maps an RDF property to one or more database
columns.</rdfs:comment>
</rdfs:Class>
```

Figure 3.7. Two major entities used to describe web methods.

The ClassMap is used to model the web service/method A in Figure 3.5, while PropertyBridge is to model result type 1. In the real life example, ClassMap corresponds to the web method that would send a name request to the service provider, and gender is the described by the PropertyBridge. The illustrative tags for ClassMap and PropertyBridge are easy to understand except the rdf:subClassOf. I assume an Object-Oriented (OO) structure in the schema file, and therefore both entities are sub-class of the ResourceMap. Another thing to mention is the ClassMap,

PropertyBridge and ResourceMap are inherited from the D2RQ literature because I need to combine my part with D2RQ engine in dealing with the relational databases.

Items to describe a ClassMap or PropertyBridge

```
<rdf:Property
rdf:about="http://www.cs.mcgill.ca/~yjin11/ichd.xml#belongsToClassMap">
  <rdfs:label>belongs to class map</rdfs:label>
  <rdfs:domain
rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#PropertyBridge"/>
  <rdfs:range rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#ClassMap"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.cs.mcgill.ca/~yjin11/ichd.xml#wsSource">
  <rdfs:label>web service source</rdfs:label>
  <rdfs:domain rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#ClassMap"/>
  <rdfs:range rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#WebService"/>
</rdf:Property>
```

Figure 3.8. Reduced set of descriptive items for ClassMap or PropertyBridge.

The belongsToClassMap used by PropertyBridge connects itself to a ClassMap. The rdfs:range determines the type of value of this item, i.e. it would only be a ClassMap. Figure 3.6 describes the Web Services entity and the wsSource is to match a ClassMap with it. The pattern item decides the form of the ClassMap or PropertyBridge's value (e.g. gender is in a string form like "male" or "female").

Items regarding ontology

```
<rdf:Property rdf:about="http://www.cs.mcgill.ca/~yjin11/ichd.xml#class">
  <rdfs:label>class</rdfs:label>
  <rdfs:domain rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#ClassMap"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <owl:inverseOf rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#classMap"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.cs.mcgill.ca/~yjin11/ichd.xml#property">
  <rdfs:label>property</rdfs:label>
  <rdfs:domain
rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#PropertyBridge"/>
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
  <owl:inverseOf
rdf:resource="http://www.cs.mcgill.ca/~yjin11/ichd.xml#propertyBridge"/>
</rdf:Property>
```

Figure 3.9. Items to connect ontology vocabulary.

Both the class and property are used to map to equivalent Class and Property from ontology. So when users write a query generating SPARQL triples that contains the property from an ontology, also identified in the application ontology (here, a mapping file will be less misleading in meaning), the associated web method will be found and a request can be issued.

Figure 3.10 gives simple application ontology. The wsUrl of the CHGIS Web Services gives the base URL of its web method. The CHGIS_Feature method provides name request by adding Romanized Chinese characters of a region's name as suffix to the wsUrl. So the ichd:pattern of the CHGIS_Feature ClassMap is actually the input parameter. The returned result would include the regionNamePY – the region's Romanized name and, regionLatitude – the latitude of the region. This is only a simplified version of the CHGIS Web Services. The advanced web method could even find the region's latitude at different times.

```
map:webservice_chgis a ichd:WebService;
        ichd:wsUrl "http://chgis.hmdc.harvard.edu/xml/placename/";
        .
# Interface CHGIS
map:CHGIS_Feature a ichd:ClassMap;
        ichd:wsSource map:webservice_chgis;
        ichd:pattern "@@placename.name_romanized@@";
        ichd:class vocab:Feature;
        .
map:region_name_romanized a ichd:PropertyBridge;
        ichd:belongsToClassMap map:CHGIS_Feature;
        ichd:property vocab:regionNamePY;
        ichd:pattern "@@placename.name_romanized@@";
        .
map:region_latitude a ichd:PropertyBridge;
        ichd:belongsToClassMap map:CHGIS_Feature;
        ichd:property vocab:regionLatitude;
        ichd:pattern "@@placename.feature_latitude@@";
```

Figure 3.10. A simplified version of the application ontology.

The Library Data Structure

The library data structure is created from the application ontology (for example, the one in Figure 3.10). The details of how the library model is produced from the application ontology will be discussed in Chapter 4. In this section, I just present and describe the library data structure.

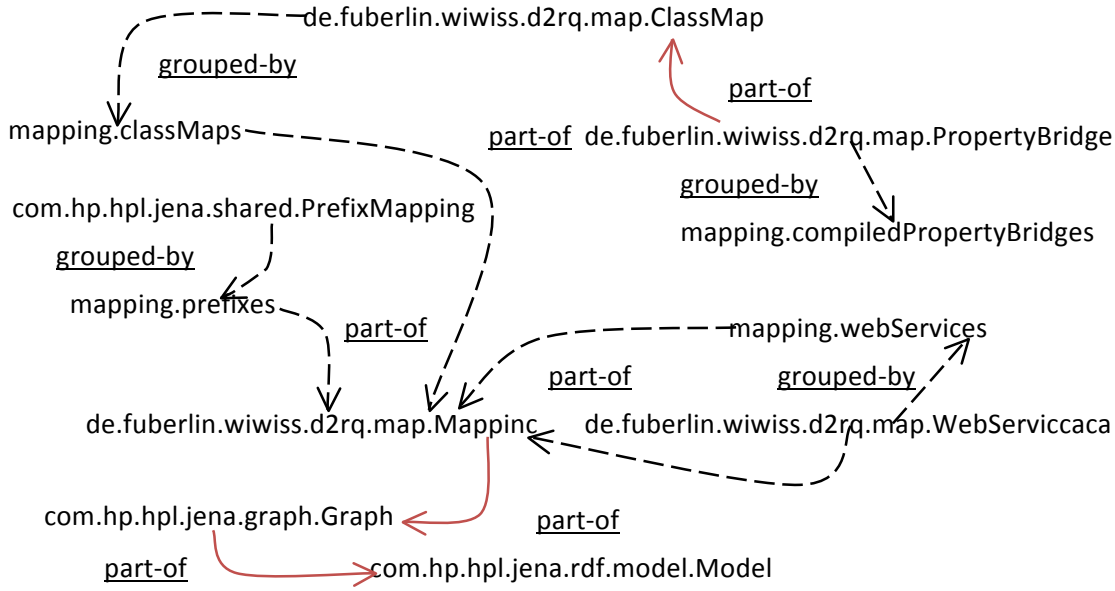


Figure 3.11. The library data structure after transformation

The most important part of the library data structure is the Mapping. All the ClassMaps, PropertyBridges and prefixes are extracted from the application ontology and are part of the Mapping. The information regarding Web Services in the application ontology is also identified to create an instance of Web Service class. The Web Service instance becomes part of the Mapping as well. In conclusion, a Mapping instance contains all the information written in the application ontology. The Mapping class, which is a D2RQ class, is then associated with the Graph class and Model class, which are from Jena.

3.3.3 The Transformation

The problem for the current query data structure is it is not sequence-based, which means the execution of a query follows an order but the model shows no sense of

ordering right now. However, the sequence of the execution is quite important. Figure 23 gives an example. On the top left of the figure is a simplified SPARQL query against the CHGIS Web Services. It gives the meaning - "find out the Romanized and Chinese name of the region 'Chengdu', and try your best to find out the alias for the region 'Kunming' ". The second triple "?regionNamePY :regionNameHZ ?regionNameHZ." actually depends on the execution of the previous triple. If the second triple is placed before the first one, it should, according to the language specification of SPARQL, give the entire mappings between Romanized name and the Chinese name of all the regions. The semantics differ considerably in the number of execution and the expectation of the query creator. So, the sequence of the query execution is quite significant.

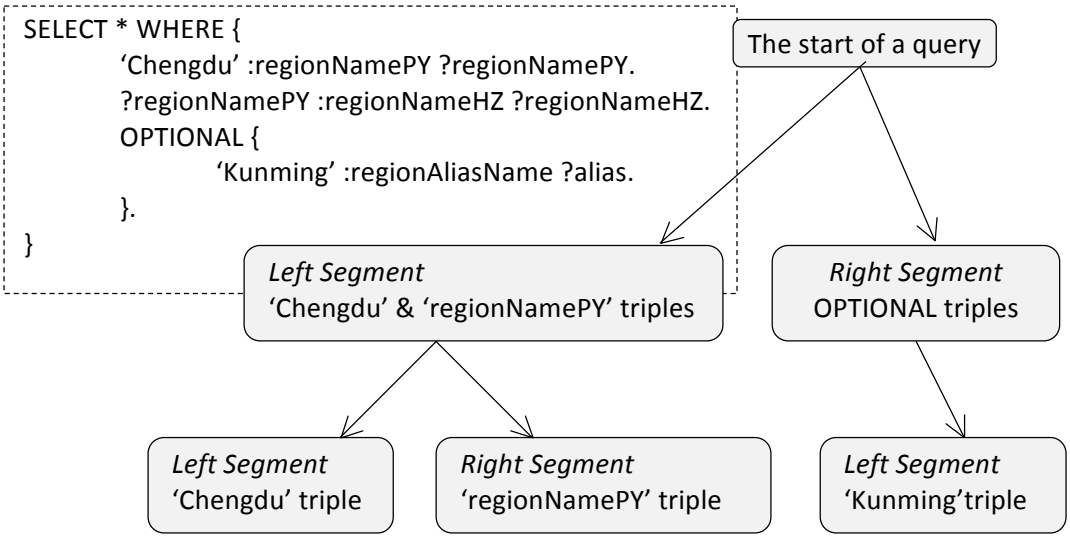


Figure 3.12 Sample SPARQL query and an expected sequence-based query execution flow

What I expect is a binary tree structure and a depth first query execution. The execution will always run the segment on the left first and after all the levels deeper than this level are executed, the right segment will then be run. This simulates the execution in real life. One thing to notice is that each segment on the left part of

Figure 3.12 could have several pointers to Web Services or databases. For example, the leftmost box that executes the 'Chengdu' triple. 'Chengdu' could be found in a Geographical Information System as a town name in the 650 B.C. but it could also be found in a biographical database, in a column representing a poet's birthplace. Thus, at least two pointers to both Web Services and the database should be created and executed. Chapter 4 will detail the implementation of the modification to the query data structure.

Now that I have two data structures, each of which has its own feature that would contribute to the success of the Query Broker. The query data structure has a structure that models the execution sequence of the SPARQL query. In each segment of the sequence-based query data structure there is the triple that models a condition of the WHERE clause. The library data structure has a collection of all the triples found in the application ontology. More importantly, these triples have information about Web Services. Combining these two data structures will give me a new sequence-based query data structure that knows how to connect to Web Services.

3.4 OTHER COMPONENTS IN THE SYSTEM

3.4.1 GUI and GUI Schema

The user interface for the ICHD project is built by Jimmy Li with Flex. It right now provides the basic functionality to query a person (the "Person" concept of ontology) and a region (the "Place" concept of ontology). One feature that distinguishes this user interface from others is it is ontology-driven, which means all the entities (including concepts and relations) users find and query are passed from the ontology file I defined. This is all due to the user interface schema file.

There are two jobs the user interface schema file assumes – absorb the content and structure of the ontology file so as to assist the generation of the user interface, and determine the creation of the output to the Servlet. When the user inputs some values to a property of a certain concept, the user interface knows the semantics of the values (for example, 'Chengdu' of the "hasRegionNamePY" means the value refers to a region's name, whereas 'Chengdu' of the "hasBrotherNamePY" probably means somebody's brother's name). So it would be easy for it to group the values with appropriate concepts and relations of the ontology file (into a semantic bundle) and to send the bundle to the Servlet.

3.4.2 Ontologies

RDF has been repeatedly used in my paper; it is short for the Resource Description Framework. It provides the metadata to describe resources and the relationships between resources. RDF is already a W3C standard XML framework. ("Resource Description Framework" 2010) Ontology is an application (or instance) of RDF.

OWL (Web Ontology Language) is a series of W3C-endored knowledge representation languages for writing ontologies. ("Web Ontology Language." 2010) The languages are composed by formal semantics and RDF/XML-based serialization for the Semantic Web. (*Ibid.*) My ontology is also coded in OWL.

My ontology (officially called the Upper Level Ontology, ULO) was composed by Chris Wellen using FOSS Protégé. Protégé is an ontology editor and a knowledge acquisition system ("Knowledge Acquisition" 2010); it provides flexible framework and APIs for other projects. ("Protégé" 2010)

The ULO models knowledge from the historical, geographical, biographical and literary respects related to China. Its concepts and relations are extracted from CBDB, MQWW and CHGIS. Its main focus is on the People and the Places. The ULO is a proof-of-concept to do interdisciplinary studies for the humanities and encourages creating new opportunities for research in Chinese Women Writings.

3.4.3 Databases

There are three databases that this project is dependent on. China Biographical Database (CBDB) is the largest online data source for Chinese Biographical information. The database server is running in Harvard University. The CBDB currently possesses more than 30,000 officials and their kin, mostly from the 9^{th} -14^{th} centuries. (Fong 2007) CBDB is a relational database and is depicted as "structuring the characteristics of those names on the basis of multiple variables (e.g., place, time, occupation, kinship, non-kinship affiliations, writings, and office-holding)". (*Ibid.*) For the project, I have acquired the permission to make a copy of it on my own MySQL server.

Ming Qing Women's Writings database (MQWW) is the only online database that provides information about women writers from the 15^{th} to the 20^{th} centuries. It is a relational database running in MySQL at McGill University. MQWW stores biographical, geographical and literary data around women writers. It is said to feature "more than 5,000 women poets and other writers, more than 10,000 poems, and roughly 20,000 images of original texts". (*Ibid*.) I only have limited permission to read data from the database server.

China Historical Geographical Information System (CHGIS) is a database of inhabited places and administrative units for China between 222 B.C and 1911 A.D. The database contains mainly historical and geographical information. The web portal for CHGIS is quite powerful – they can either download historical map data or submit their own databases (*Ibid.*). However, the administrator of CHGIS only provides RESTful Web Services for me.

3.4.4 Web Services

The CHGIS RESTful Web Services is different from many other web services. First it only provides a limited collection of services (or web methods/functions, interchangeably). Users can only query the system with a region's name, a CHGIS ID of a geographical feature and a region's name with specific time. Second, whichever service a user chooses, the Web Services server will only return a schema-fixed XML file. This means the structure of the returning result is not dependent on user's choice of services, e.g. the result always contains a region's Romanized name, Chinese name, current name, latitude and so on. Third, unlike databases, (for example, people could query the primary key for any other values or query any other attribute for the primary key), CHGIS Web Services can be queried only with primary keys (i.e. a region's name, an ID or a region's name with time constraint). This impacts the way I deal with SPARQL queries.

A triple (or condition) of the WHERE clause in an SPARQL query is composed of a subject, a predicate and an object. With D2RQ that supports query database with SPARQL, people could assign a value to either subject or object. For example, a database table "person" which has "person ID" and "person name" attributes. The

relation (or predicates/properties, interchangeably) I define for the “person” concept in ontology would probably be “hasPersonName” with “person ID” as subject. People can either query the ID by writing “?personID :hasPersonName ‘John’ .” or, query the name by “’p1’ :hasPersonName ?personName .”. With CHGIS RESTful Web Services, people can only do the latter.

Chapter 4 Implementation

This chapter is the implementation part of the previous chapter. It describes my effort to bring the Semantic Web and Web Services with data structures and algorithms. I want to start with the SPARQL generator first, followed by the implementation details of the components described in the abstract model. These include the Query Broker and the result processing.

4.1 The SPARQL Generator

4.1.1 Overview of the SPARQL Generator

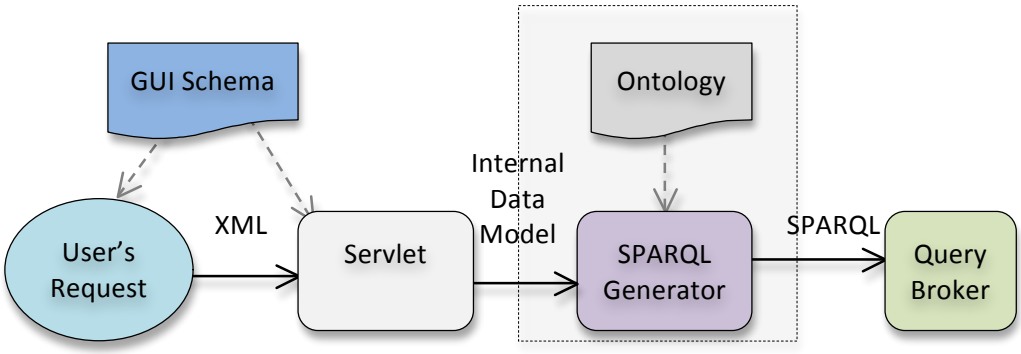


Figure 4.1. Architecture of the SPARQL Generator

The rectangle P in the Figure 4.1 shows the scope of the SPARQL Generator (the Servlet was coded by Jin Xing (2010)). When the encoded data are sent from the user interface in XML, the Servlet hosted on an application server (e.g. Apache Tomcat) catches the message. The parser inside the Servlet then retrieves the data from the XML and regroups the data into an internal data structure. The internal data structure is called so because the SPARQL generator actually resides in the Servlet and so the data structure the Servlet creates to save the XML data is called the internal. The SPARQL generator then takes the data structure as the input, works with the help of

the ontology file, and generates SPARQL queries. In this section, I am about to illustrate how this works.

4.1.2 The XML Input

There are at least two ways to enable ontological intelligence in the SPARQL generation – one is to ask the user interface developer to create SPARQL queries based on user's input; the other way is to separate the user interface from any other jobs except serving the users, which means the Servlet takes the duty to handle the query generation.

The benefits of the former are multifold. First is that the user interface knows everything about the topic and content of the query. The user interface itself should be built upon the knowledge of the ontology file – for example, concepts constitute the main search items while relationships serve the abilities users could do with the concepts. When the user clicks the "Person" class and "name" relationship (or property), the user interface knows the semantic meaning of the entered value. It is thus easier to build an SPARQL query for the user interface than delaying that to the Servlet. Apart from this, the Servlet is assured to serve only one purpose – as the controller in the MVS architecture. Future developers of the project are therefore easily limited to their own knowledge. For example, Servlet developers are not compelled to spend time on the knowledge of ontology except the user interface developer and the Query Broker developer.

However, the drawback is evident – the user interface developers should be confused about their roles. Except creating HTML or Flash code, they have to know how to

transform user's request into a strange query language – SPARQL. Considering this, people compromise to the second solution, that is, the user interface needs only to export an encoded form of its collected data and let the Servlet generate SPARQL queries for the Query Broker.

The design of the schema of the exported data could vary. But something should always be embedded in the design. For simple queries, the predicate and the value, as well as the location of the value (i.e. as subject or object). This is important because the request format for Web Services differ. For some Web Services the query distinguishes by identifying the ID of the table. On the other hand, some Web Services are not indexed by numbers, but characters. For complex queries, since the sequence matters for the execution of multiple queries, how to encode the sequence of different queries is essential. Especially for SPARQL queries, because the variable of the first query could be the input data of the second one, schema designers probably should even include connecting variables in the XML.

4.1.3 The Result Parser and the Internal Data Structure

As I said earlier in this section, the result parser sits on the Servlet in an application server. It is always listening to the incoming HTTP requests. Once it catches the XML file from the user interface, the result parser needs to deserialize data in the XML file into an internal data structure.

Since the XML files are just instances of the user interface schema file, the result parser could take full advantage of the semantics and surely the schematic information from the user interface schema file. The XML file from the user interface

will be validated firstly against that schema. The parser could then facilitate itself by deserializing necessary parts in the XML. This is a good design from the architect's point of view - low coupling between the user interface and the parser of the output of the user interface. From time to time, designers are changing the information inserted in the exported XML file. So even if the structure of the XML file is changed or replaced by a totally new schema, nothing in the code of the result parser has to change.

Another important charge of the result parser is, to instantiate the internal data structure with data from the XML file exported by the user interface. The overall objective of the SPARQL Generator is to generate SPARQL queries from user's requests. So before the working of the generator, it needs the data in memory. Again, the design of the data structure could vary but it is critical to retain features of the information in the XML in the data structure, e.g. the sequence of different queries.

4.1.4 The SPARQL Generator

When the parsed data is ready in memory, the SPARQL Generator is prepared to create new SPARQL queries. As I mentioned earlier, there are two things necessary to finish its work. Except the structured data from the user interface, the ontology file is also required.

The sole task of this generator is to produce ontology-aligned SPARQL queries. However, there are simple and complex situations. I regard a request from the user that only correlates to a single property or two in the ontology file as simple. For this simple situation, the SPARQL Generator only needs to validate the demand found in

the user's request (now as an internal data structure) against the ontology file. This is to ensure the legal status of the user's request as well as the possible existence of data that could be retrieved from Web Services. When the validation is of no problem, the SPARQL Generator places appropriate prefix and suffix to the candidate query, before generate variables found vacant in the triple. The SPARQL query is based on triples. And then the data user inputted will be filled in a suitable place in the triple. Figure 4.2 describes such process.

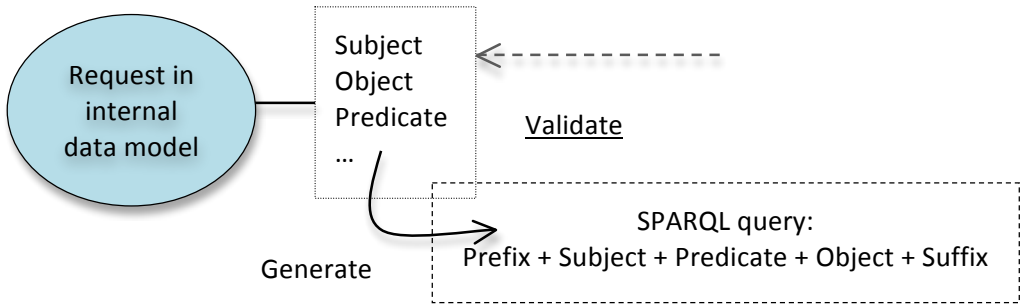


Figure 4.2. SPARQL generation in a simple situation

However, there are some complicated situations. For example, there are some user queries that demand database-procedure-like process. When the user types a person name, he/she expects the system to find out all the related information to this person, and sometimes, a list of people who share the same name. I have previously discussed this a little bit, saying that the ontology would define what information are considered pertinent to a person, for example. Figure 4.3 gives me a concrete example of this. As you can easily perceive that, a "Person" is a concept (or class). So there are lots of relationships (interchangeably, properties, verbs) has some kind of relation to a concept. You would find properties about a person's last name, the gender, nationality and so on. In fact, properties of a concept should iterate all the relationships found associated to this concept. One would never find them in any other places, except the sub-classes. Therefore, it is not difficult to comprehend if I can dump out all the properties of a class in the ontology, I could find all the information to a person.

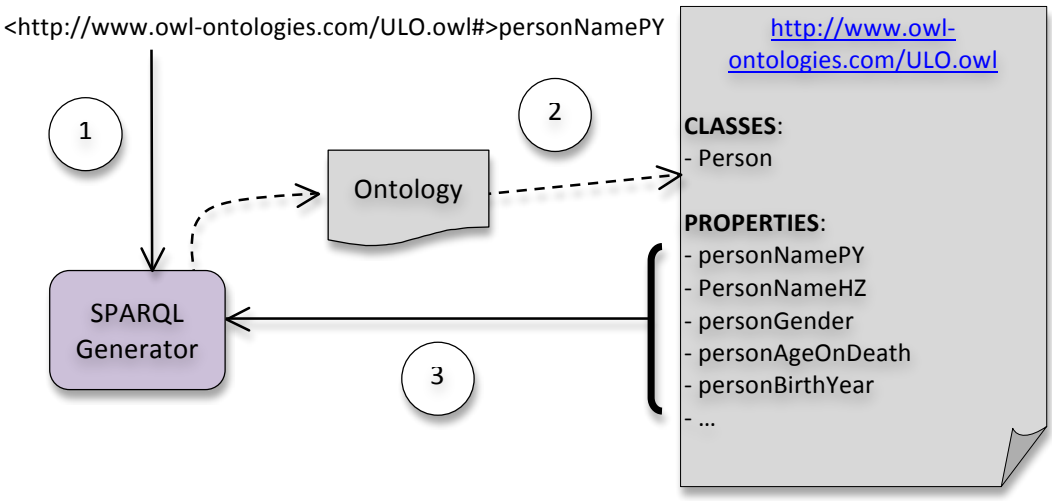


Figure 4.3. A resolution process to find all the properties related to a "Person" class by a "personNamePY" property.

The very property indicates a Chinese person's name in the alphabetical form. The URL of the property indicates the place of its definition. The property, class and ontology are example from the ICHD project.

From the user's input, I have successfully generated equivalent SPARQL queries. Now I am at the edge of feeding the query to the Query Broker. The Query Broker should parse the query according to some rules and match the features of the query to the library, which stores the mapping information to turn these features into ontologically aware Web Services. I am about to discuss the design and implementation details of these in the following section.

4.2 The Query Broker

In this section, I discuss the implementation of the Query Broker. According to Figure 3.3 ("Architecture of the Query Broker") in Chapter 3, there are three sub-sections I want to clarify. The data structures section mainly introduces the implementation of two data structures – how an SPARQL query is transformed into the query data

structure and how the application ontology is translated to the library data structure. The transformation section describes two components, the data structure comparator and the query generator. The implementation of the data structure comparator is then surrounded by the explanation of two processes, the process of the query data structure before the data structure comparison and the optimization of the data structure after the data structure comparison.

4.2.1 Data Structures

4.2.1.1 Query Data Structure

4.2.1.1.1 Introduction

I need a query parser that could specifically handle the parsing of SPARQL. This is provided by the *com.hp.hpl.jena.sparql.lang.sparql* package. There are two functioning classes in this package that is responsible – ParserSPARQL and SPARQLParser classes. The ParserSPARQL class is a direct child of the Parser class and works as a transfer station for the calling class that asks it to parse, and the actual parser, the SPARQLParser class. The problem for the parser is the input to it is a text. How does it get the structural information and the data attached to the text? I will determine the mechanism inside the SPARQLParser later in this section.

Once the parser finishes parsing, it should save the structured data in a recognizable data structure, which is available in the *com.hp.hpl.jena.query* package. The Query class is the place where data are represented in the Jena/ARQ form and the Query Broker will need its object and the intelligence in the application ontology to generate a Web Services request. The Query class itself is again a data structure of various information that are used to represent all possible situations and data in the SPARQL

query. For example, SPARQL is allowed to be composed of different graph patterns, which is embodied in the Query class as different Element or ElementGroup (*com.hp.hpl.jena.sparql.syntax*) objects. It also has to deal with query modifiers like OPTIONAL (which means one graph pattern could be neglected if it returns nothing in the result set), or the SQL-like ORDER (to dictate the order of the result display). In general the Query class has a hierarchical structure to model the SPARQL query – the Query object has data to represent the prefix (which states the ontology vocabulary that could be used in the query), the type of the query execution (an SPARQL query could be any one of SELECT, CONSTRUCT, ASK and DESCRIBE – For simplicity, I will cover SELECT only in this paper), the return variables (like SQL, it needs to know what information is needed; it could also be a wildcard), the WHERE clause (which is like the WHERE in SQL – it lists all the conditions to match some requirements) and, the modifier clause (e.g. FILTER, GROUP, ORDER, LIMIT and so on). The layer below this level in the Query class contains more detailed classes that support these structures.

There is one more data structure that reveals the structure of SPARQL and its difference from SQL queries. The Triple class *com.hp.hpl.jena.graph* models the triples inside the conditional WHERE clause of SPARQL queries. The type of the subject, predicate (a.k.a. property or verb) and object is first determined as a com.hp.hpl.jena.graph.Node, for example, subject and object could be a variable, a string, a numeric, a blank Node and etc. The predicate is always from a vocabulary so the ontology is extracted firstly and turned into a com.hp.hpl.jena.sparql.lang.sparlq.Verb object. After that they are inserted into a Triple object to denote a triple of the SPARQL query.

4.2.1.1.2 Structure

Becuase it is both exhaustive to extend this paper pointlessly and distract me from my focus on how an SPARQL query is transformed into Web Services request. I want to concentrate on some simple SPARQL examples. Figure 4.4 gives a simplified data structure of the query in memory.

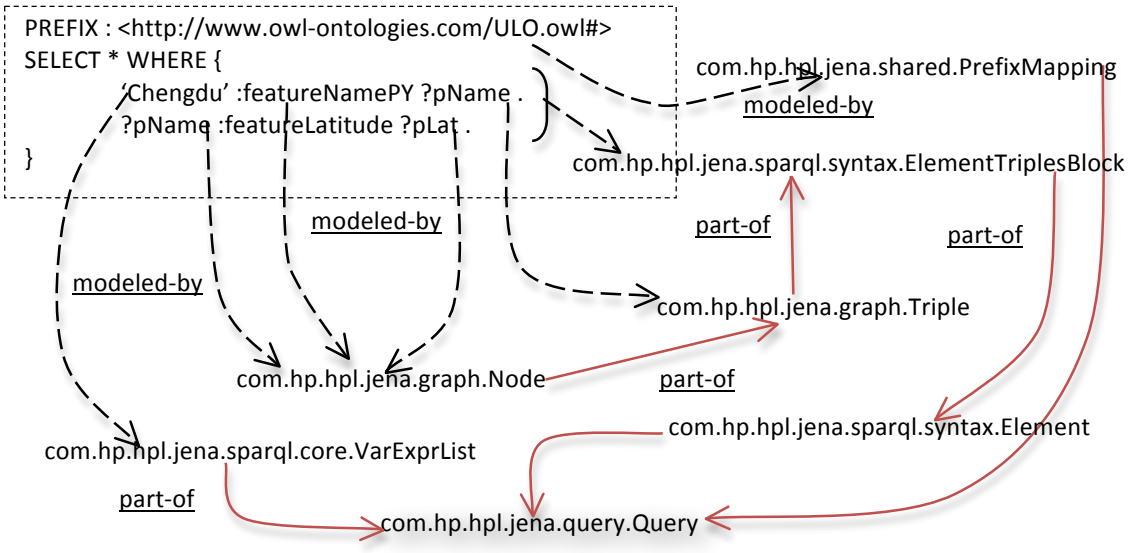


Figure 4.4. A simplified query data structure for the sample SPARQL query using the Jena Semantic Framework.

As Figure 4.4 suggests, the Query class contains pointer to mainly three kinds of different objects, namely the PrefixMapping, the VarExprList and the Element. PrefixMapping manages the vocabulary that could be used to construct a query; the VarExprList hosts all the return variables that users need – this could also be wildcard instead of specific variable names; the Element class denotes the conditions in the WHERE clause. The ElementTriplesBlock is a collection of Triples. If I considering several graph patterns could co-exist in the WHERE clause, the number of ElementTriplesBlock could be more than one. As I mentioned earlier a Triple object comprises of Node objects denoting the subject, the predicate and the object.

4.2.1.1.3 Method

The logic in the generation of the Element of a Query is more relevant to understanding the query data structure. Because the code reveals how each triple is organized and iteratively constructed from the scratch. Figure 4.5 gives a pseudo code segment for the creation of an Element object (the code is abstracted from D2RQ). Triples with the same subjects are put into the same list, because it would be easier for the Web Services request generator to find the triples that could share the same request. Note that the Node type could vary in a wide range, from variable to constant-based values (e.g. string, Boolean). Each of these types would map to a corresponding derivative Node class.

```
procedure createElement(query, tokens)
        Element element
        query.addElement(element)
        # repeat until all triples in the WHERE clause are found
        while not tokens.done() do
                ElementTriplesBlock triples
                element.add(triples)
                # only begins from the subject
                if tokens.next() is start of a triple then
                        Triple triple
                        Node sub, pred, obj
                        sub ← findNodeType(tokens.get())
                        pred ← Verb(tokens.next())
                        obj ← findNodeType(tokens.next())
                        triple ← Triple(sub, pred, obj)
                        # aggregate triples with the same subject
                        if sub is listed in a triple list then
                                list.add(triple)
                        end if
                        else # the list is based on the subject
                                List list
                                list.add(triple)
                        end else
                        triples.add(triple)
                end if
                # move to the next line of triple
                tokens.next()
        end while
end procedure
```

Figure 4.5. Generation of an Element object for the Query class

4.2.1.2 Library Data Structure

4.2.1.2.1 Introduction

Quite similar to the process of transforming an SPARQL query into an internal data structure, this section aims to turning schematically self-designed application ontology into an internal data structure. As I discussed before, the benefits of doing so are to facilitate the Query Broker to compare the SPARQL query with some library (or mapping file) and generate Web Services requests. Many a procedure is actually the same – loading the application ontology into the memory and parsing thereafter. However there is something different. Firstly, instead of loading some widely known protocol, the structure of the application ontology is revised to suit for RESTful Web Services, from the original D2RQ design. The advantage is I could devise any format I prefer, but the problem is I need to write my own parser to do this. And secondly the application ontology is based on a RDF-conformed file, N3 Turtle for example, so I need a N3 parser instead of a simple tokenizer for the query.

There are three connected parts I need to know in this section. The schema design focuses on repeating my roadmap to schematically bring Web Services to the Semantic Web world. More importantly it should embed the instructions in the file about how to map some part of an SPARQL query to a specific Web Services request. The application ontology loading assumes the duty to load and parse the application ontology with a RDF file parser. The RDF file parser should use the application ontology schema to save the parsed data in a meaningful way. This is to help the mapping parser in the application ontology parsing phase to reformulate these data in memory to a triple-based data structure. The triple-based data structure is very much similar to the data structure generated from the SPARQL query except that they carry

some mapping information. The mapping information will in the end reform the query data structure to HTTP request in a semantic meaningful way.

In this section, I will rely heavily on the Jena and D2RQ libraries. Jena provides support in the N3 Turtle parsing but again it depends on me to tell the parser what to do considering the application ontology schema file. The D2RQ appears even more important in this section because it set up the example for building the library data structure. In many cases I could follow the way it manages how to structure information from the mapping library (the application ontology) and to assemble them into triple-based data structure. But the difference is it handles relational databases but I am doing Web Services. Not only is the way to design the schema is different, but the data structure to save mapping information is also varied. So is the processing logic to extract data from the application ontology.

4.2.1.2.2 Method

There are two phases involved in the transformation, both of which need parsing. The application ontology loading focuses on parsing the structure data from the application ontology to non-RDF Jena-based Web Services data structure. There is no big problem until the Query Broker wants to compare the Jena-based query data structure to this one. Because the Jena model has no hierarchical and mapping information I defined in the schema file. For example, the triples in the library could be found the collected randomly. Triples don't know if there is any relation among them, e.g. belongsToClassMap, not to mention if they know some mapping information is associated with others. The Application Ontology serves the purpose of converting such a semantic-unaware data structure to a more reasonable one. It relies heavily on how the schema file is written to regroup the relations in the data structure.

- Application Ontology Loading

To the Query Broker, the library (or the mapping file) is just a sequence of texts. The sequence gives the location where the library could be found – on a local machine or in a remote server. The most important thing for the Query Broker in this application ontology loading phase is to find the file, import the file into memory, parse the file according to itself schema (e.g. N3 Turtle) and save the data structure for further processing.

For the parser involved in this phase, the triples in the application ontology has no special semantic meanings – they are only RDF triples. For example, in Figure 4.5, what the map:region_latitude a ichd:PropertyBridge seems to the parser is subject, predicate and object. The only problem is the "a"; it needs to find out where the "a" is provided (probably a well-known RDF data structure defined in the Prefix area of the application ontology). The job to understand ichd:PropertyBridge and, construct some relation with the map:CHGIS_Feature because a ichd:belongsToClassMap is found, for instance, is left for the parser in the next phase.

The application ontology is firstly imported into memory, evoking the system to find out the syntax of the file. This is to help the system to get an appropriate derived class instance of the com.hp.hpl.jena.rdf.model.RDFReader. The input parameters of the RDFReader are com.hp.hpl.jena.rdf.model.Model, the java.io.InputStream that keeps the application ontology and, the path to the application ontology. The model is used because it is the data structure to hold the parsed triples. There are mainly two things the RDFReader (a.k.a. parser) needs to know – the triples and the prefix mappings. The prefix mappings disclose information about the source of the subject, predicates

and objects of triples. And it is therefore essential for the parser in the next phase to imply relations among these triples. The prefix mappings are saved in the com.hp.hpl.jena.share.PrefixMapping while a Triple (com.hp.hpl.jena.graph.Triple) instance is collected by comp.hp.hpl.jena.graph.impl.TripleStore. Both of these two data structures are again parts of the com.hp.hpl.jena.graph.Graph, which is cached by the com.hp.hpl.jena.rdf.model.Model. Figure 4.6 gives a reduced description of the algorithm above in pseudo code (the code is abstracted from D2RQ).

```
procedure load(ontology)
        Model model
        Graph graph
        # set the graph to a model
        model.addGraph(graph)
        InputStream input ← ontology
        # determine the type of the ontology
        syntax ← ontology.findSyntax()
        Reader reader ← graph.findReader(syntax)
        # find the prefix
        PrefixMapping prefix ← reader.findPrefix(input)
        graph.setPrefix(prefix)
        TripleStore triples
        # find all the triples
        while not input.done() do
                Triple triple ← input.next().generateTriple()
                triples.add(triple)
        end while
        graph.setTriples(triples)
end procedure
```

Figure 4.6. Algorithm to collect all triples and prefix mappings.

- Application Ontology Parsing

For the application ontology parsing, how to find the potential relations among triples is important. Triples generated from the previous phase are considered semantic-free. This is why the schema file defined earlier is critical, because it embeds hierarchical information to support regrouping the triples. For example, the ichd:belongsToClassMap predicate should coach its subject be associated with the

object. And the semantics of this predicate mean the subject should be a PropertyBridge while the object a ClassMap. The ichd:class and ichd:property are even more important predicates, because they include mapping information about ontologies. When a triple is parsed in this phase and a ichd:class is found, there are two steps to go – one is this triple should be associated with a ClassMap and the subject should be a pointer to that ClassMap; and the object of this triple will give information on the specific concept (a.k.a. class) of a ontology, which should provide the vocabulary for an SPARQL query inquiring the Query Broker for some Web Services data defined in the application ontology. The ichd:property serves a very similar purpose except it will match a property (i.e. predicate or verb) in the ontology.

The triples (com.hp.hpl.jena.graph.Triple) saved in a model (com.hp.hpl.jena.rdf.model.Model)'s graph (com.hp.hpl.jena.graph.Graph) are retrieved first. They are compared with the schema file (represented as a Java class) to remove unknown terms. If it finds a triple with a misspelled predicate "ichd:claass", it should remind the system. So all the RDF terms are conforming to the schema. A de.fuberlin.wiwiss.d2rq.map.Mapping instance will be constructed then. The very instance provides the data structure to hold all the regrouped triples and prefix mappings. This means, every piece of the details of the application ontology will be found in the Mapping instance. Lastly, all the prefix mappings saved in the Jena Model will be transferred to the Mapping instance.

The regrouping of the triples starts from recognizing Web Services connection information, i.e. the Web Services entity declared in the application ontology. Predicates from the schema will be extracted to assist the finding. All of the connection parameters for the Web Services will be put into a

de.fuberlin.wiwiss.d2rq.map.WebService instance. The WebService class is created inside a D2RQ package to process both relation databases and Web Services. ClassMaps are also recognized and used to instantiate a de.fuberlin.wiwiss.d2rq.map.ClassMap instance. These ClassMap instances are again collected by a Java HashMap in the Mapping instance. In addition, triples that have PropertyBridges predicates are used to instantiate de.fuberlin.wiwiss.d2rq.map.PropertyBridge instances. They are however not processed like the ClassMap instances that are kept in a HashMap of the Mapping instance. They are processed as attached corresponding ClassMap instances.

Then the parser will validate the semantic correctness of the data structure against the schema. And the Mapping instance is cached by another graph and a model.

```
procedure parse(model)
        graph ← model.getGraph()
        triples ← graph.getTripleStore()
        Mapping mapping
        # begins parsing
        while not triples.done() do
                triple ← triples.next()
                # check if the terms in a triple conform to the schema
                if not triples.checkTerms(triple) then
                        triples.remove(triple)
                end if
                # copy all the prefix mappings
                mapping.copyPrefixMappings(graph.getPrefixMappings())
                # parse the Web Services
                if triple is about Web Services then
                        mapping.addWebServices(triple)
                end if
                # similar process for ClassMaps and PropertyBridges
        end while
        # attach PropertyBridges to ClassMaps
        mapping.regroup(propertyBridges)
        # attach ClassMaps to Web Services
        mapping.regroup(classMaps)
        graph.addMappings(mapping)
        model.addGraph(graph)
end procedure
```

Figure 4.7. An algorithm for the application ontology parsing

Structure

Figure 4.8 gives a simplified overview of the data structure after the transformation is done. I only show important classes and data structures that will be used in the next section. The structure is quite intuitive and most of the components are explained in this section, so I don't give any more remarks for this.

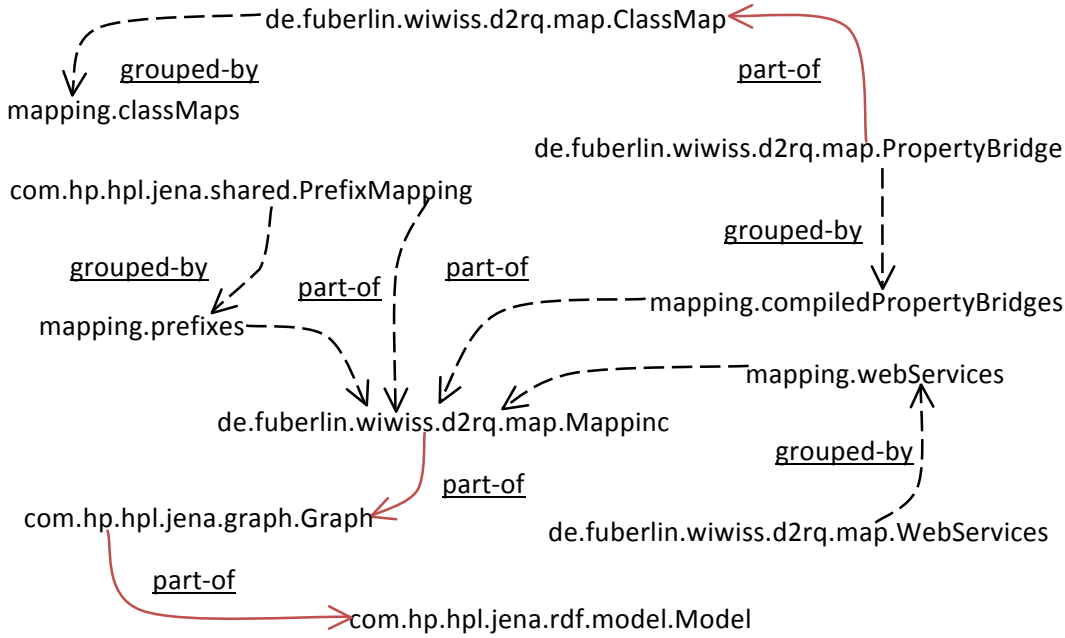


Figure 4.8. The library data structure after transformation

4.2.2 The Transformation

The major challenges in this section are how to combine the query data structure and the library data structure, and how to generate Web Services requests out of the result of the previous challenge. There are logically two components in this section – the data structure comparator and the request generator. The names are intuitive but there are still some details in these two components.

There are actually 5 main phases involved in this section, classified into two logical components. The data structure comparator includes the transformation of the query data structure, integrating the query and library data structures and, optimization of

the selected relations. The selected relations are the results of the integration work. Creating the Iterators for the selected relations and, the request generation are parts of the request generator.

4.2.2.1 Data Structure Comparator

When a request generation engine wants to create HTTP request based on the library data structure, it needs a few things to begin with, namely, for example classified relations. Relation is a data structure that assumingly combines the shared parts of the query and library data structures, and attaches useful information of the Web Services. “Classified” is not strictly a term like in Machine Learning, but it also should mean relations should be grouped according to some rules. For example, triples in a query refer to different Web Services should naturally considered in different groups. Relations are a structure logically descending from the query data structure and they should follow the classification rules.

Similarly to the concept in classified relations, the order of the query execution is also important for the request generator. Query execution follows a rule – executing in a linear order stated in the query. This rule seems to be simple because the query embeds the order in itself. However it is not, the query data structure does not have a clear structure able to be developed into a sequence-based query execution. Instead, the query data structure is more like a collection of all the triples in a query. Neither does the library data structure have such a sequence-based structure. Therefore, it is very necessary for me to find out the query execution order and regroup the query data structure into a new data structure.

I discuss these issues with other details in this section.

I mentioned earlier in this section that the query data structure does not have a clear sequence-based structure. Figure 4.9 gives an example. On the top left of the figure is a simplified SPARQL query against the CHGIS Web Services. It gives the meaning - "find out the Romanized and Chinese name of the region 'Chengdu', and try your best to find out the alias for the region 'Kunming' ". The second triple "?regionNamePY :regionNameHZ ?regionNameHZ." actually depends on the execution of the previous triple. If the second triple is placed before the first one, it should, according to the language specification of SPARQL, give the entire mappings between Romanized name and the Chinese name of all the regions. The semantics differ considerably in the number of execution and the expectation of the query creator. So, the sequence of the query execution is quite significant.

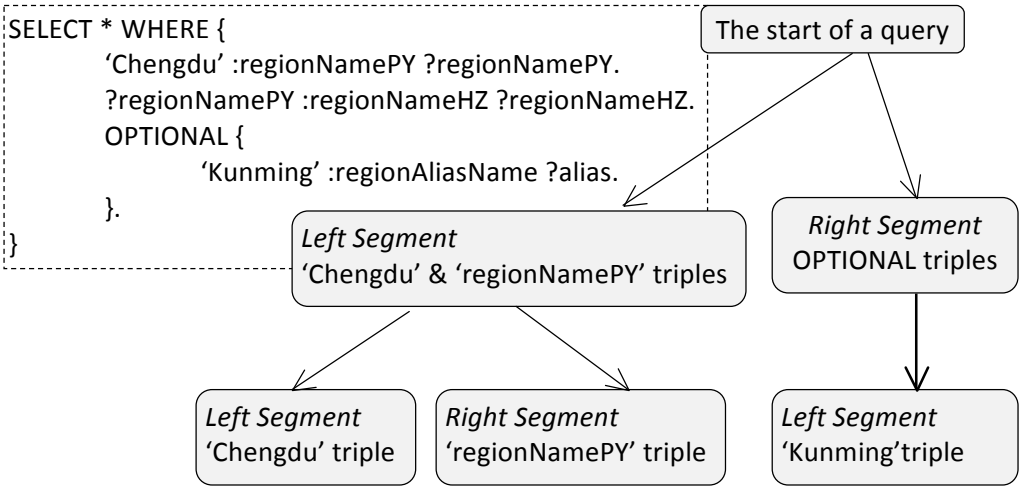


Figure 4.9. Sample SPARQL query and an expected sequence-based query execution flow

What I expect is a binary tree structure and a depth first query execution. The execution will always run the segment on the left first and after all the levels deeper than this level are executed, the right segment will then be run. This simulates the execution in real life. One thing to notice is that each segment on the left part of Figure 4.9 could have several pointers to Web Services or databases. For example, the

leftmost box that executes the ‘Chengdu’ triple. ‘Chengdu’ could be found in a Geographical Information System as a town name in the 650 B.C. but it could also be found in a biographical database, in a column representing a poet’s birthplace. Thus, at least two pointers to both Web Services and the database should be created and executed.

However, words above are only descriptions for what I expect the data structure for the SPARQL is. They are far away from what are constructed in the build of query data structure. Figure 4.10 shows what I have right now.

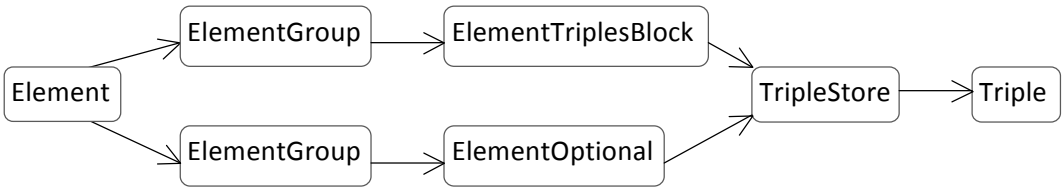


Figure 4.10. Query data structure

So the problem is to fill the query data structure into a sequence-based data structure. Jena/ARQ’s package com.hp.hpl.jena.sparql.algebra.Op comes into replacing the original sequence-free data structure. One distinguishing feature of the Op family is it is designed to carry sequence information. For example, the OpLeftJoin class has a left component and right one (while each component could be another instance of an Op’s child class, e.g. OpLeftJoin again). During the query execution, the request generator will run the component labeled left and then the right. A real-life example is the SPARQL query in Figure 4.9 – the block including ‘Chengdu’ and “?regionNamePY” triples is the left component and will be executed in the first place; the block containing ‘Kunming’ triple is the right. The most basic unit of execution is com.hp.hpl.jena.algebra.op.OpBGP, which represents an ElementTriplesBlock in the query data structure.

I will see the pseudo code transforming the query data structure into this new structure in Figure 4.11, followed by a printout of the result of the transformation in Figure 4.12 (the code is abstracted from D2RQ).

```
procedure transform(queryDataModel)
        # find the query model
        Element ← queryDataModel.getElement()
        # match the type of Element
        Op op
        while not Element.done() do
                if ElementTriplesBlock & ElementOptional then
                        Op ← OpLeftJoin
                else if size(ElementTriplesBlock) > 1 then
                        Op ← OpJoin
                # other situations might apply
                end if
        end while
        # deal with the content of op
        VariableList vs
        if Element.next() instanceof ElementTriplesBlock then
                OpBGP opBGP ← OpBGP(Element.get())
                op.addComponent(opBGP)
                VariableList vars <- Element.get().generateVariables()
                opBGP.addVarList(vars)
                vs.addVarList(vs)
                op.addLeft(opBGP)
        end if
        # similar measure applies to the right component
end procedure
```

Figure 4.11. Pseudo code for transforming query data structure to sequence-based data structure.

The algorithm above actually adds variables found in the triple to the new structure. Doing this is to help the result-processing phase find out what is needed in the triple and the users. I only demonstrate one case among many other cases. Things could get quite complicated when the modifiers (LIMIT, ORDER or GROUP) apply to the SPARQL query. However I'm not discussing that right now, instead I only focus the most basic situation to prove the feasibility of my proposal. Figure 4.12 is the result of the transformation of the query data structure for the SPARQL query in Figure 4.9.

```
(leftjoin "[?regionNameHZ, ?regionAliasName, ?regionNamePY]"
  (bgp "[?regionNameHZ, ?regionNamePY]"
        (triple "Chengdu" <http://www.cs.mcgill.ca/~yjin11/ULO.owl#regionNamePY>
        ?regionamePY)
        (triple ?regionNamePY <http://www.cs.mcgill.ca/~yjin11/ULO.owl#regionNameHZ>
        ?regionameHZ))
  (bgp "[?regionAliasName]"
        (triple "Kunming" http://www.cs.mcgill.ca/~yjin11/ULO.owl#regionAliasName
        ?regionAliasName)))
```

Figure 4.12. Result of the transformation of the query data structure of the query in Figure 4.9.

4.2.2.1.1 Integrating the Query and the Library Data Structures

Now there are two data structures, each of which serves different purposes. The new query data structure instructs the return variables as well as the order of query execution, while the library data structure provides mapping information to transform query data structure into Web Services requests. I have seen how the two purposes of the query data structure are realized and now it is important to know how the triples and mappings in the library data structure fulfill its duty.

Let me start from learning what information is in these two data structures. The query data structure in Figure 4.12 demonstrates that the triple (subject, predicate and object), the returning variables and the order of execution are evident. Figure 4.13 describes the basic components of the library data structure – a NodeRelation instance of the PropertyBridges. The library data structure is full of NodeRelation instances – data structure to describe the relations and properties of a ClassMap or PropertyBridges. The example in Figure 4.13 models a PropertyBridge of the regionNameHZ property of the CHGIS Web Services. I can find variables of the PropertyBridge, web service request URL, the value for the request (currently nothing is in it) as well as the subject, predicate and object of the triple. In fact, a

PropertyBridge should also have a pointer to its ClassMap but the pointer is not printed out here.

```
NodeRelation(
  variables:      [@@placename.name_romanized@@,
@@placename.name_vernacular@@]
  web services (w.s):     http://chgis.hmdc.harvard.edu/xml/placename/
  value for w.s: ()
  subject => Literal(Pattern(@@placename.name_romanized@@))
  predicate => Fixed(<http://www.cs.mcgill.ca/~yjin11/ULO.owl#regionNameHZ>)
  object => Literal(Pattern(@@placename.name_vernacular@@))
)
```

Figure 4.13. A printout of the triple in the library data structure

Clearly there is something in common between the query (Figure 4.12) and the library (Figure 4.13) data structures. For example, they both have a structure to represent the triple, and they extract the variables out of the structure. But even in these similarities there are differences. The triple in the query data structure has changeable values for the subject and object. They could be constant or variables. Even variables have random names. The library data structure, in comparison, has a fixed pattern – for instance, the subject's type is Literal and the representation is "@@placename.name_romanized@@". The representation looks weird because it serves the regular expression matcher to extract string from it. For triples, only the predicate is the same and could be used to compare models. The differences are the library data structure has Web Services information to construct a request but it has no information about how to proceed in a query execution. Instead, that information is in the query data structure. And even if the library data structure knows the pattern to create a request but it has no knowledge of the parameters of the request, which should be found in the query data structure.

Considering the similarities and differences between the query and library data structures - the query data structure tells me the execution order of a query while the

library data structure knows how to construct a Web Services request, I am thinking whether it is possible to combine the features of these two models.

There are two issues I want to remember – first is I want to keep the sequence information of the query data structure. And second is there is probably more than one match that can be found for each triple in the query data structure, so I need to go through all the triples in the library data structure. These two issues help me design an algorithm to compare the two data structures.

There should be two iterations of this algorithm. The first iteration is to get all the triples in the *structured* query data structure exposed to the matching. I highlight the structured here meaning that, the triples should be iterated in structured blocks. For example, there are currently two blocks in the query data structure in the example of Figure 4.9, the ElementTriplesBlock that hosts triples for 'Chengdu' and 'regionNamePY' and, the ElementOptional block that hosts the 'Kunming' triple. These two blocks are not considered with the same privileges – they are ordered in the query execution. So they should not be iterated at the same time. The ElementTriplesBlock will be iterated firstly. When all the matched NodeRelations are combined with the triples in the ElementTriplesBlock, the triples in the ElementOptional block will be iterated. In this sense, the integration of the query and library data structures could be considered as a structural update (from the library data structure) to the query data structure.

The second iteration of this matching algorithm should go through all the triples in the library data structure. And the comparator works in this iteration. The triple from the query data structure is used to compare with the NodeRelation of the library data

structure. The predicate is the major source of comparisons. If the prefix of the predicates and the predicates themselves are found to be the same, they are considered referring to the same thing. Mappings between the subject and object of the matched triples (from both query and library data structures) are created. For example, the "?regionNamePY :regionNameHZ ?regionNameHZ." will return mappings like "regionNamePY => Literal(Pattern(@@placename.name_romanized@@))" and "regionNameHZ => Literal(Pattern(@@placename.name_vernacular@@))" – the left part representing the query data structure, the right the library data structure. This is to help the result processer understand which parts of the result set are needed by the query.

Information about Web Services from the library data structure should also be attached to the new query data structure, for example, the connection URL for the web method of the CHIGS Web Services. This information will be combined with the parameters found in the query data structure to form HTTP requests. For instance, the connection URL for CHGIS's placename web method is http://chgis.hmdc.harvard.edu/xml/placename/ but no parameter for the place name is in the library data structure, instead, 'Chengdu' as the subject for both "'Chengdu' :regionNamePY ?regionNamePY." and "?regionNamePY :regionNameHZ ?regionNameHZ." triples will be identified and appended to the web method as the parameter.

```
procedure match(queryDataModel, libraryDataModel)
        List queryList ← findAllBlocks(queryDataModel)
        List libList ← findAllNodeRelations(libraryDataModel)
        List selectedNodes
        while not queryList.done() do
                Lists nodes
                tripleList ← queryList.next()
                while not tripleList.done() do
                        triple ← tripleList.next()
                        while not libList.done() do
                                if triple.getPredicate() == libList.next().getPredicate() then
                                        NodeRelation node
                                        node.addMapping(triple, libList.get())
                                        node.addWebServices(libList.get())
                                        node.addParameter(triple)
                                        nodes.add(node)
                                end if
                        end while
                end while
                selectedNodes.addAll(nodes)
        end while
end procedure
```

Figure 4.14. Matching algorithm to combine query and library data structures.

Now I'd like to show the part of the result of running the algorithm as in Figure 4.16.

The result is based on the SPARQL query in Figure 4.9.

```
[NodeRelation(
   variables:      [@@placename.name_romanized@@]
   web services (w.s):     http://chgis.hmdc.harvard.edu/xml/placename/
   value for w.s: ("Chengdu")
   regionNamePY => Literal(Pattern(@@placename.name_romanized@@))
), NodeRelation(
   variables:      [@@placename.name_romanized@@,
@@placename.name_vernacular@@]
   web services (w.s):     http://chgis.hmdc.harvard.edu/xml/placename/
   value for w.s: ("Chengdu")
   regionNameHZ => Literal(Pattern(@@placename.name_vernacular@@))
   regionNamePY => Literal(Pattern(@@placename.name_romanized@@))
)]

[NodeRelation(
   variables:      [@@placename.name_vernacular@@]
   web services (w.s):     http://chgis.hmdc.harvard.edu/xml/placename/
   value for w.s: ("Kunming")
   regionNameHZ => Literal(Pattern(@@placename.name_vernacular@@))
)]
```

Figure 4.15. NodeRelation lists of query and library data structures.

One point I should notice about the NodeRelation lists is there are actually two lists, each representing a block of the query data structure. In fact after the lists are created, they are added to the corresponding block of the query data structure. The new query data structure is then composed of both triples and NodeRelations.

4.2.2.1.2 Optimization for the NodeRelations

I have so far achieved an important intermediate result. But before feeding the request generator with the NodeRelation-based query data structure, I need to do some optimizations on the NodeRelations.

The problem for the current NodeRelations is redundancy. During the generation of the new NodeRelations, the integration algorithm does not care about if it will produce two NodeRelations with the same Web Services and parameters, but differ in the mappings. For example, the first two NodeRelations are all referring to the placename web method and 'Chengdu' as the parameter. The only difference between these two NodeRelations is the second one has one more mapping between the query and the library data structures than the first one. In fact, these two NodeRelations should be joined, because apparently all the variables in the second NodeRelations can be found in the result returned from the first request.

So the optimizing challenge is to locate NodeRelations with the same Web Services and input parameters and, to combine these NodeRelations into new ones. Figure 4.16 gives the pseudo code of the algorithm in the following.

Assume I have NodeRelations A and B for the NodeRelations in Figure 4.15. NodeRelation A is the one with only one mapping ("name_romanized"), and B two

mappings (“name_romanized” and “name_vernacular”). I start with NodeRelation A. A new list for NodeRelations integration is created. A virtual NodeRelation that can be combined with any others is inserted to the list to instantiate it. I then compare NodeRelation A’s Web Services part and the input parameter with everything in the list. Since the first and only one NodeRelation in it could be combined with any others, NodeRelation A actually replaces the virtual NodeRelations and becomes the first entry. And then NodeRelation B is compared with everything in the list – right now only NodeRelation A. I find that not only A and B share the same Web Services (or more exactly web method here), but also they have the same input parameter – ‘Chengdu’. Mappings from NodeRelation B will be extracted to combine with the mapping in NodeRelation A. Duplicated mappings will not be accepted in the new NodeRelation A. NodeRelation B will then be removed from the NodeRelation list kept by the very block of the query data structure.

While I should combine NodeRelations with the same Web Services and parameters, I should be cautious with other similar but different situations, for example, queries with the same Web Services but a different input parameter. It is quite common for CHGIS Web Services to get two queries with the same method (e.g. regionNameHZ – get the Chinese name of the region), but different values (e.g. ‘Chengdu’ and ‘Chongqing’). These NodeRelations should be kept independent to create their own HTTP requests.

Another issue is two NodeRelations might share the same subject and the object but not the predicate, partially. It means two Web Services probably share the same predicate (e.g. “regionNameHZ”) but they are from two service providers. Again, they should be treated as independent NodeRelations to create their own requests.

One more situation is, NodeRelations with everything the same, but differ in blocks, e.g. one in the ElementTriplesBlock block and another in the ElementOptional block. The example is the 'Chengdu' NodeRelation and the 'Kunming' NodeRelation. Even if they match with each other, they are considered independent to each other, because a different block has a different context. A NodeRelation in another group might serve as the basic for other NodeRelations in the same block. And the execution of other NodeRelations might be impeded or ignored because nothing returns. I don't want one NodeRelation in one block has any influence on another.

```
procedure optimize (Block block)
        # initiate the list with a NodeRelation that can be combined with any other
        # NodeRelations
        List joiner ← NodeRelation(Any)
        # the block may contain several NodeRelations
        while not block.done() do
                while not joiner.done() do
                        # joiner.next(), the previously inserted NodeRelation
                        # block.next(), the next NodeRelation in the block
                        if joiner.next().webService() != block.next().webService() then
                                joiner.add(block.get())
                        end if
                        else
                                if joiner.next().parameter() != block.next().parameter() then
                                        joiner.add(block.get())
                                end if
                        end else
                        if joiner.next().webService() == block.next().webService () then
                                if joiner.get().parameter() == block.get().parameter() then
                                        # join() is to combine two NodeRelations
                                        joiner.join(joiner.get(), block.get())
                                end if
                        end if
                end while
        end while
end procedure
```

Figure 4.16. A segment of pseudo code for the algorithm above.

I don't show the result of the optimization because one just needs to remove the first NodeRelation in Figure 4.15 to see the result.

4.2.2.2 Request Generator

When the data flow arrives at the request generator, information in the query data structure is sufficient enough to be used to create Web Services requests. And it is not that difficult to generate a request. But there could be several requests that need to be generated and sent out. The principal challenge to the request generator therefore, is how to sequentially generate and send out the requests.

I want to start this section by picturing how several Web Services requests of a query data structure are sequentially generated and sent out and then I will zoom in to a specific request generation. Figure 4.17 demonstrates the generation algorithm.

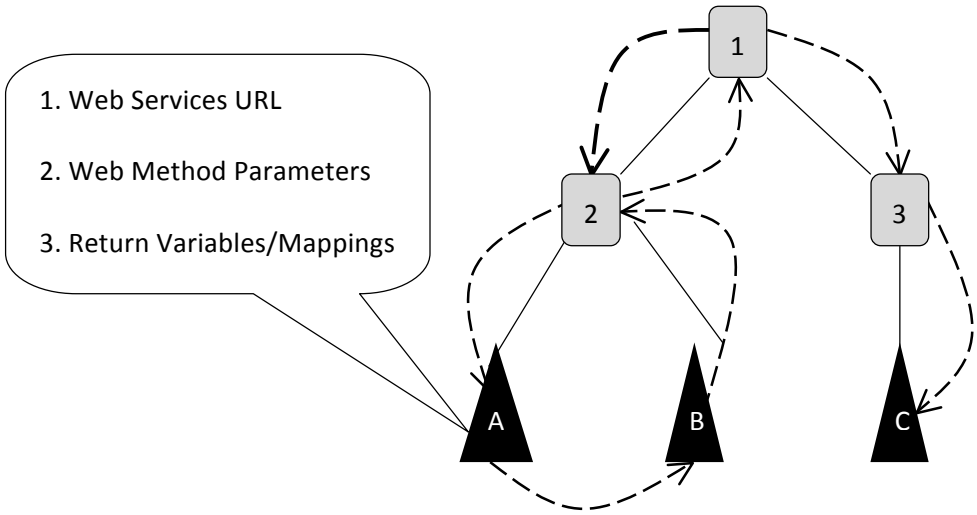


Figure 4.17. A simple algorithm to generate several Web Services requests

The graph on the right simulates the query data structure generated from the SPARQL query in Figure 4.9. But one thing is different – the subject of the second triple in Figure 4.9 is replaced by a constant ‘Chongqing’, which makes this triple an independent NodeRelation. The rectangles represent NodeRelations. The first rectangle represents NodeRelation ‘Chengdu’. Instead of being combined with the first NodeRelation A (‘Chengdu’), NodeRelation B will also generate a new request.

Rectangle C denotes the NodeRelation (‘Kunming’) in the OPTIONAL graph. The rounded rectangle boxes represent OpLeftJoint/Element (1), OpBGP/ElementTriplesBlock (2) and OpBGP/ElementOptional (3). The rectangular callout on the left of Figure 4.17 is a zoomed-in abstract view of the NodeRelation. I have listed the major information of the NodeRelation that will be used for the request generation.

The algorithm begins at box 1, the OpLeftJoin/Element of the query data structure. It knows it should go to the left component first, which is the OpBGP/ElementTriplesBlock. The ElementTriplesBlock will identify two distinct NodeRelations – the one with ‘Chengdu’ querying for the Romanized name of ‘Chengdu’ (actually ‘Chengdu’ is the official Romanized name, but other forms like ‘Chengtu’, ‘Chengdu Shi’ or ‘Chengdu Fu’ are possible), and the one with ‘Chongqing’ querying for the Chinese name. The NodeRelation represents all the triples in this block querying for the placename web method with an input parameter ‘Chengdu’. So the return variables list (or called the mappings) possibly contain several distinct mappings between the query and the library data structures. It is probable NodeRelation A (‘Chengdu’) is inserted before NodeRelation B, so information in the NodeRelation A will be transformed to generate a Web Service HTTP Get request. The request generation of NodeRelation B will not proceed until the data NodeRelation A asks for arrives. When the Query Broker receives data NodeRelation B asks for, ElementTriplesBlock (2) will take control of the Query Broker again before it returns the control to the Element (1). Now the Element will process component on the right side. The ElementOptional (3) will move the control

to NodeRelation C (the 'Kunming' triple). When the request of NodeRelation C is issued and is sent out, this phase is over.

From Figure 4.15 in the last section, I learned that there are three types of information that could be found in a NodeRelation – data structure of Web Services, the input parameters for a web method, as well as a set of mappings between the variables (that need to be returned to the user) and the representation found in the library data structure. The data are more than sufficient to form the request – I just need to append the input parameters for a web method to the URL string found in the Web Services data structure. Figure 4.18 describes such a process

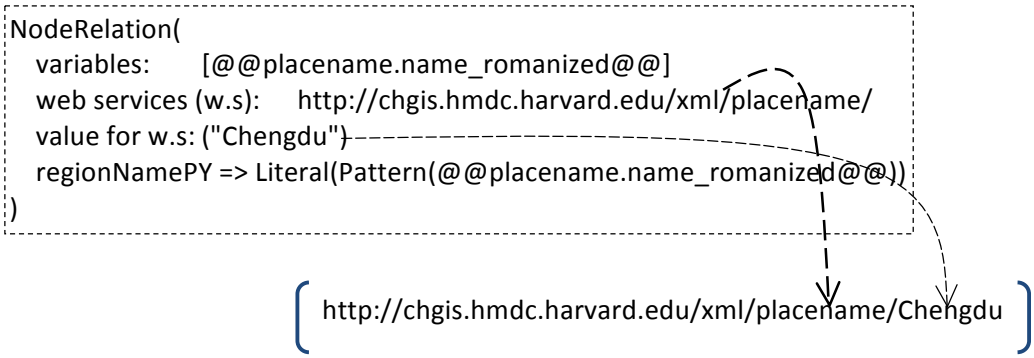


Figure 4.18. Request generation

When the request is generated, it will be sent out in HTTP GET to the Web Services provider. The other information in the NodeRelation – mappings (or variables), is important for the result processing.

4.2.3 Result Processing

After the HTTP requests are sent out to the Web Services provider and, before the Servlet or GUI receives the result, there are two important steps the result processor needs to do – returned result parsing and regrouping the result.

Web Services usually respond to user's request by sending a structured file containing the result, e.g. XML. This is to help the result processor of the service demander efficiently extract data from the result. CHGIS Web Services do the same way (Berman, 2008). There are two sections in the returned XML from the CHGIS server – HEADER and RESULTS. The HEADER mainly contains some statistics of the result, for example, the number of items found in the CHGIS database, the number of items transmitted to the request demander, and the time of execution. The RESULTS section is composed of <item> elements. An <item> element represents a record found in the CHGIS database. The sub-elements of an <item> are <placename>, <feature_type>, <temporal>, <spatial>, <part_of>, <preceded_by>, <evidenced_by> and <links>. Here, I don't to introduce all the details. For the sake of demonstration, I only need to know things in the <placename>. The <placename> element contains three sub-elements: <name_romanized>, <name_vernacular> and <name_alternate>, which have their corresponding definitions in the application ontology.

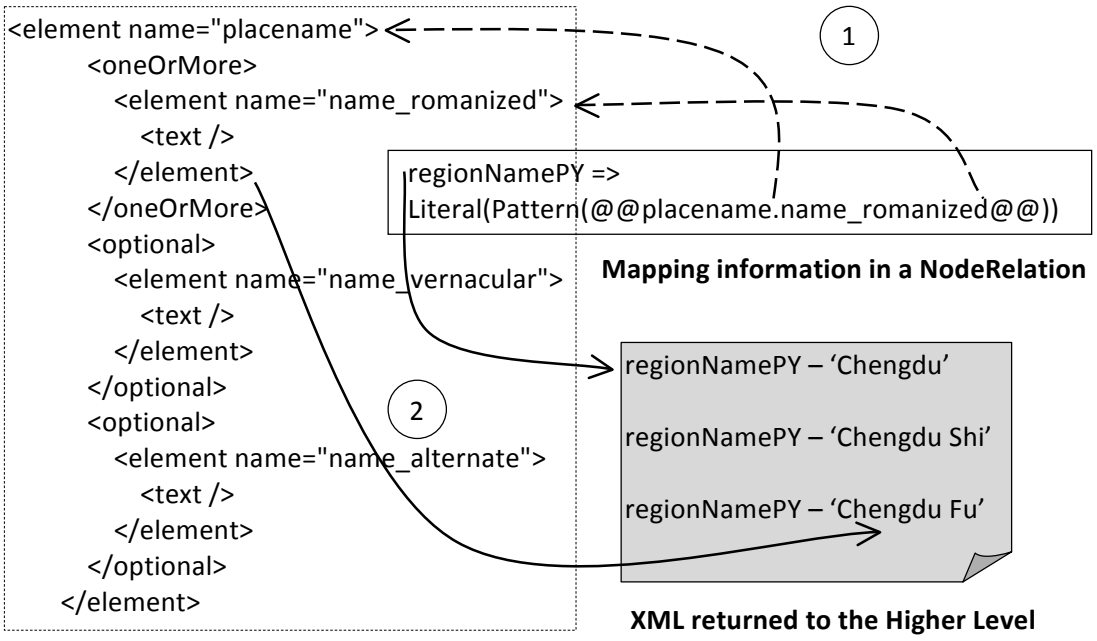


Figure 4.19. Result processing and the mappings in a NodeRelation

The mapping information between the query and the library data structures are essential in the result processing. During the parsing of the returned result from the Web Services provider, the library part of the mapping, "Literal(Pattern(@@placename.name_romanized@@))" is used to match the element in the returned result. A regular expression matcher will break the "@@placename.name_romanized@@" into "placename" and "name_romanized". When the sub-element "name_romanized" of "placename" element is found in the result, the value of the "name_romanized" (<text/>) will be extracted. It is highly possible that more than one value will be found in the returned result. So it is important to design a data structure to keep these values.

The other half of the mapping in a NodeRelation ("regionNamePY") is from the query data structure. It is used as the key to be coupled with the values found in the result. The key-value combination will form the content returned to the Servlet or the GUI.

Chapter 5 Results Returned from the System in Operation

My research proposal is to bridge the ontological gap between the Semantic Web world and Web Services, specifically between SPARQL and REST. I expect to receive a user's query about a person or a place (only these two concepts are currently been fully supported by the ontology). The query will be exported as an SPARQL query to the Query Broker. The Query Broker sends the translated request to the RESTful services, which is then sent to the remote databases. The results come back to the Query Broker. The Query Broker should return a well-defined XML file containing all the results back to the GUI via the Servlet.

5.1 Proof of Concept

The Query Broker right now supports a single request to Web Services, which means each time a single request is sent to the services provider, and multiple requests to Web Services, as well as cross-data sources query. This cross-data sources query is a goal of the ICHD project, which allows users to query one data source based on the result of a query to another data source. For example, people might be interested in all the women poets related to one region (e.g. 'Chengdu'). In this case, the Query Broker should query all my data sources, CBDB, MQWW and CHGIS that possibly have spatial information. One scenario would be that the region's name is found from the CHGIS Web Services, and the names (possibly more than one name) are then queried in MQWW database to locate any poets related to this region.

I am still far from all the requirements of the humanities researchers who will need this system to facilitate their work. I am only a proof-of-concept project that demonstrates the possibility to query databases and Web Services with a simple ontology file. So in this results chapter I want to present some sample queries, mainly around the querying RESTful Web Services, and then a demonstration of querying both the database and Web Services.

All my queries related to Web Services are based on the RESTful Web Services offered by CHGIS (CHGIS 2010). Currently CHGIS accepts RESTful URIs ("Representation State Transfer" 2010) containing query values and it returns result in XML format. There are only three web methods offered by CHGIS and they are read only. These web methods are placename search, which queries the geographical database by a region's name; unique identifier search, supporting database querying by IDs (e.g., a region ID and a geographic feature ID); and combined placename and year search – this allows users to narrow down their placename search by feeding a temporal parameter. All of these web methods will return the XML file. For demonstration, I only use the placename search method. An example is below. http://chgis.hmdc.harvard.edu/xml/placename/QUERY-STRING.

I assume I have the result of the SPARQL generator, which is a SPARQL query. I don't demonstrate the work of SPARQL generator in this section, because how it really works depends on the design of the GUI schema file. I find it unnecessarily related to my research question by showing how an SPARQL query is generated based on my own design of the schema file. The vocabulary of the ontology used to create SPARQL queries in the project can be found in http://www.cs.mcgill.ca/~yjin11/ULO.owl. The application ontology, which serves as

the library for query transformation, can be found in

http://www.cs.mcgill.ca/~yjin11/AO.n3.

5.2 Single request to Web Services

The first example asks about both the Romanized and Chinese names of a region called 'Chengdu'. The two triples ":regionNamePY" and ":regionNameHZ", which are defined by the project ontology (ULO.owl) should be found in the application ontology as well. This will help the Query Broker identify the CHGIS placename web method as the main body of the request. The subject of the first triple of the query, 'Chengdu', should be extracted by the Query Broker and be combined with the placename URI to form a request. Figure 5.1 gives the SPARQL query.

```
PREFIX :<http://www.cs.mcgill.ca/~yjin11/ULO.owl#>
SELECT * WHERE {
        'Chengdu' :regionNamePY ?regionNamePY .
        ?regionNamePY :regionNameHZ ?regionNameHZ . }
```

Figure 5.1. Sample single request query.

Figure 5. 2 gives the generated Web Services request and the result. The object of the first triple in the query is the same as the subject of the second triple. So in SPARQL it means the result of the first query (values of the variable "?regionNamePY") will be the "input" or subject of the second triple. If neither the subject nor the object of a triple is a value (e.g. 'Chengdu'), then all the relations between the subject and the object should be found.

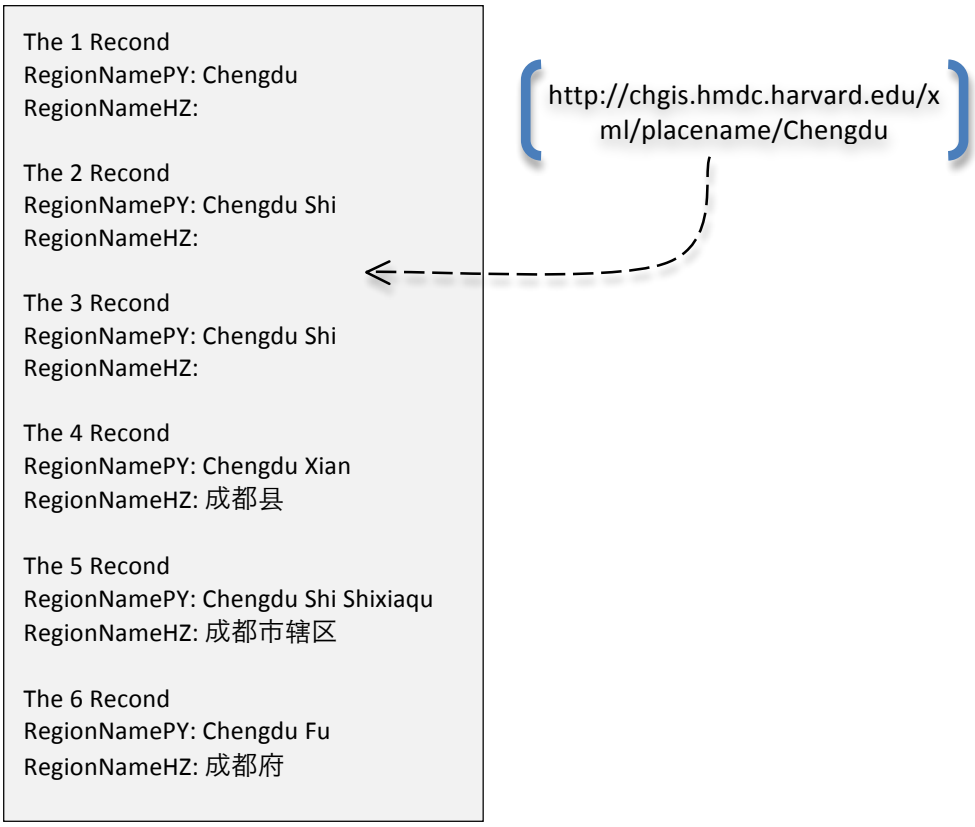


Figure 5.2. Web Services request and its result.

Most of times, more than one record will be found. Some records will only have part of the request information. So it is important for the result processor in the last section to map all the results to their corresponding variables appropriately.

5.2 Multiple requests to Web Services

My system supports sending multiple requests to a Web Services server. Figure 5. 3 gives an example query that produces two requests. When the Query Broker meets after the first triple, it tries to combine the second with the first one. But then the Query Broker soon realizes that even these two triples share the same Web Services/web method (i.e. CHGIS placename search method), they don't have the same input parameter. So the second triple will be used to generate the second request. After the first request ('Chengdu') is executed and the result is processed, the second request ('Kunming') will be launched.

This example proves that thanks to a structured way to store those requests (this is like a binary tree in which the request about ‘Chengdu’ is left child while ‘Kunming’ is the right one.), I can process multiple requests in a query.

```
PREFIX :<http://www.cs.mcgill.ca/~yjin11/ULO.owl#>
SELECT * WHERE {
        'Chengdu' :regionNamePY ?regionNamePY .
        ‘Kunming’ :regionNameHZ ?regionNameHZ . }
```

Figure 5.3. Sample multiple requests query.

Figure 5.4 gives the requests and the result. Notice that the results of two independent requests are merged into one result set. This is because there might be several graphs (i.e. other blocks, like an OPTIONAL block). One result set of a graph/block is better to be processed for the next graph. Also, please be advised that this result has been modified to remove 10 records that have nothing for the Chinese name of ‘Kunming’, to save the space.

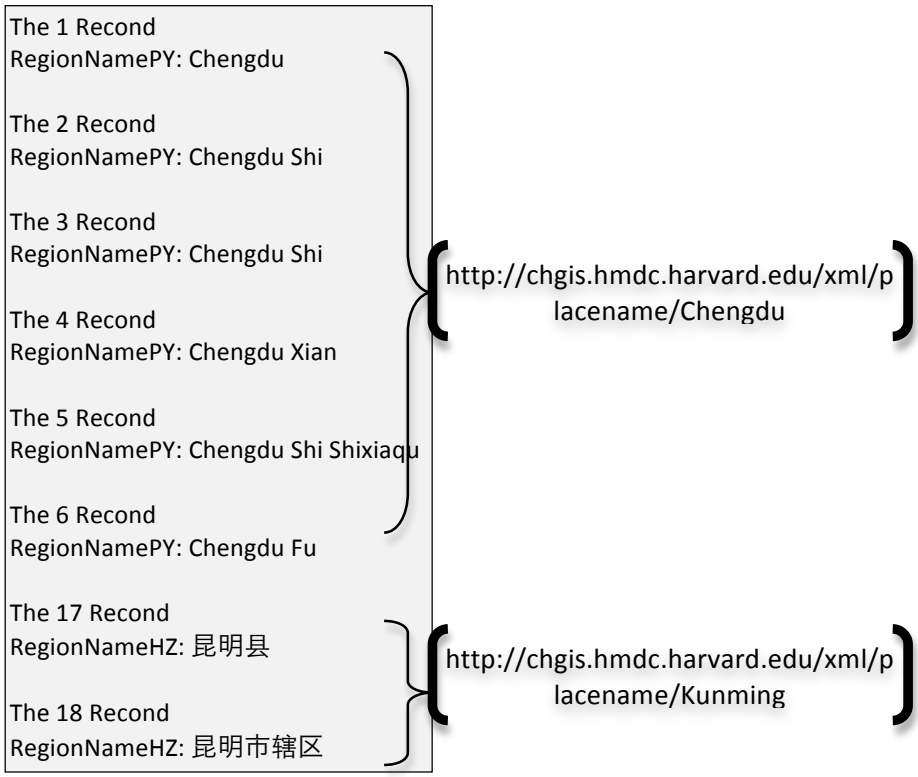


Figure 5.4. Multiple requests and the results.

5.3 Multiple requests for multiple graphs

Multiple graphs means a query has multiple blocks (it might have OPTIONAL or GROUP blocks). Different graphs should have independent requests. Even a triple in one graph that shares everything the same as a triple in another graph, they should generate different requests. This is because a graph is considered quite independent in SPARQL – whether triples in an OPTIONAL block return anything is not related to the result return outside of the OPTIONAL block. The query would be very similar to Figure 5.3 except that the 'Kunming' triple in put in an OPTIONAL block. And the result would be the same as Figure 5.4. To save the space, I don't demonstrate the query and the result any more.

5.4 Multiple requests for multiple data sources.

One of the goals of the ICHD project is to enable users query multiple data sources with one independent ontology (i.e. http://www.cs.mcgill.ca/~yjin11/ULO.owl). Before designing and implementing this proposal, I can only query multiple databases, but not Web Services.

I give an example (Figure 5.5) to do cross-data sources query to both databases and Web Services. To simplify my example, the database will only be MQWW, the largest online database for Chinese Women Writers in the Ming and Qing dynasties. And the Web Services will be the CHGIS RESTful Web Services.

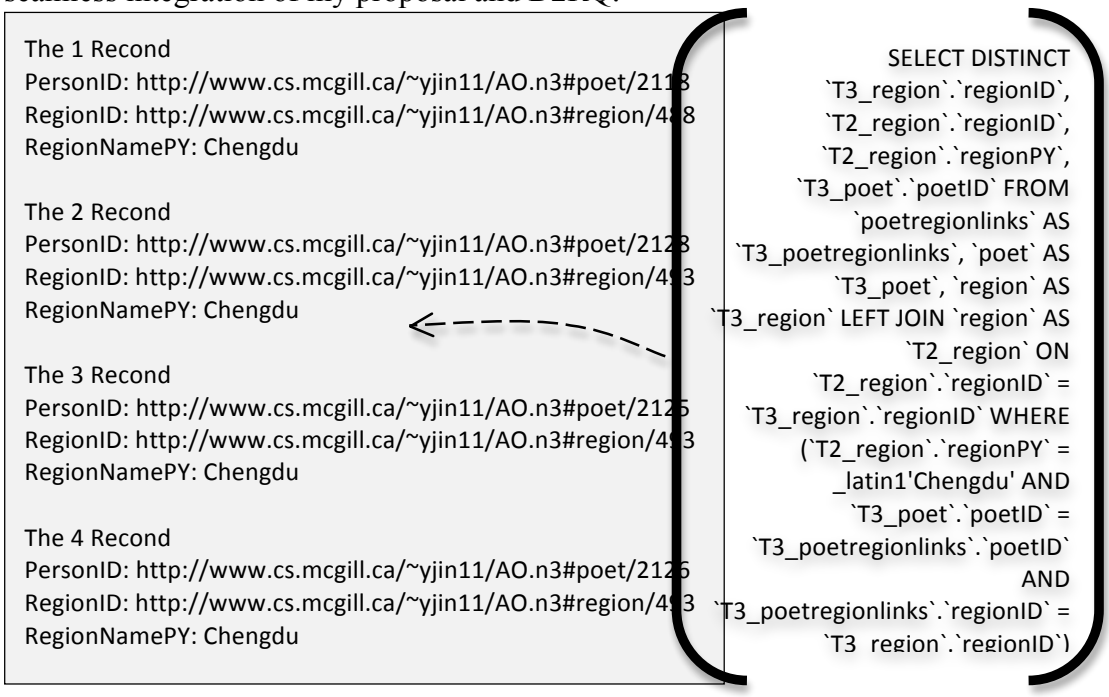
The query first asks CHGIS about all the geographic information related to region 'Chengdu'. One of them is the Romanized name of 'Chengdu'. The result is shown in Figure 5.2. The triple in the first OPTIONAL block redirects the query to MQWW

database. Now the Romanized names of 'Chengdu' (considered as strings) are the values for the object of this triple. The query is to find the entire region IDs in MQWW with a name from the Romanized name list. If any region ID is returned, the query moves on to the next OPTIONAL block. The triple in this block still queries MQWW database. The ":personRelatedToRegion" predicate, which maps to a PropertyBridge in the application ontology (http://www.cs.mcgill.ca/~yjin11/AO.n3), connects the person concept (or the poet table in MQWW) with the region concept (or region table in MQWW). So this predicate allows people to find person IDs with region IDs. In conclusion, this query is to find all the poets related to 'Chengdu' in all times.

```
PREFIX :<http://www.cs.mcgill.ca/~yjin11/ULO.owl#>
SELECT * WHERE {
        'Chengdu' :regionNamePY ?regionNamePY .
        OPTIONAL {
                ?regionID :regionPY ?regionNamePY .
                OPTIONAL {
                        ?personID :personRelatedToRegion ?regionID .}.}.}
```

Figure 5.5. Sample cross-data sources query.

The Web Services request generation is provided by my proposal's implementation, while the SQL query generation is supported by D2RQ. This demonstrates the seamless integration of my proposal and D2RQ.



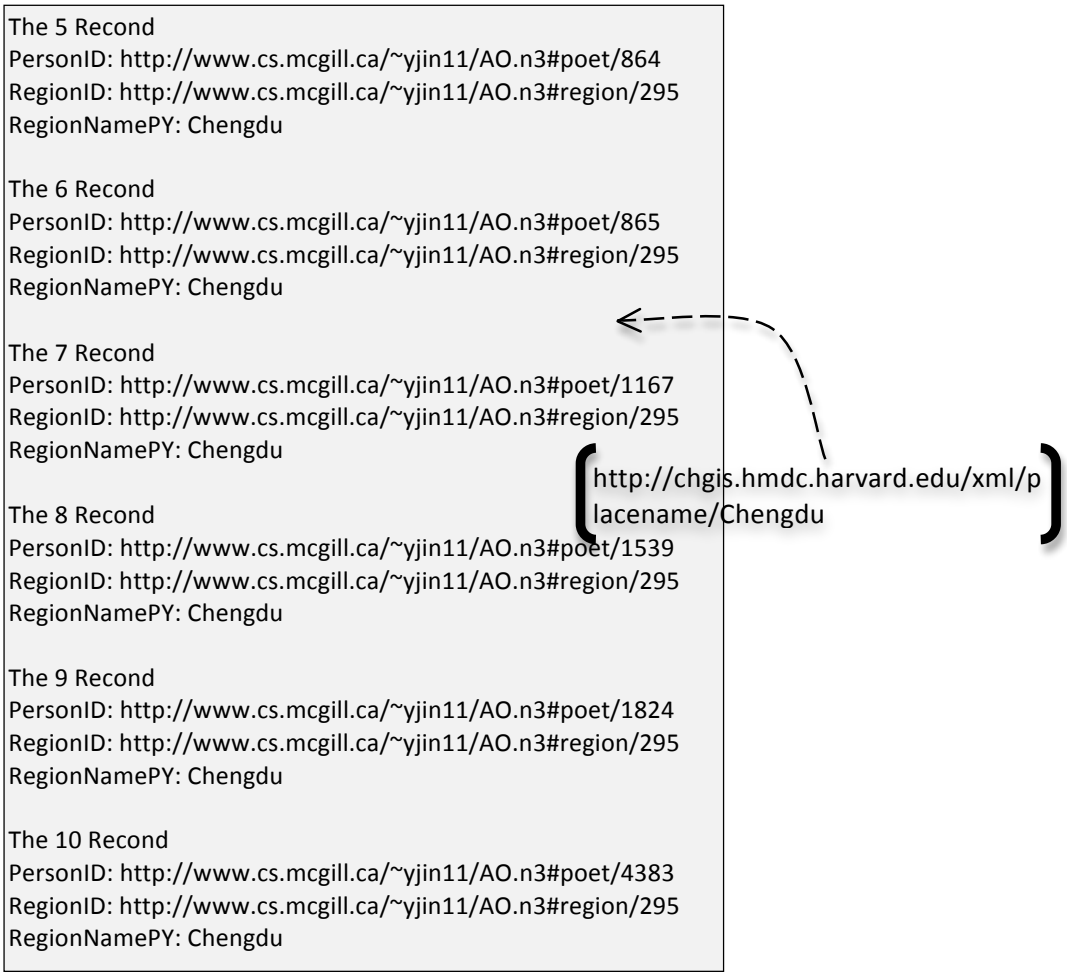


Figure 5.6. Web Services request, and their results.

5.5 Conclusion

I have showed a limited portion of the final result, to be exact, only one sixth of all the results. This shows the consequence of "Chengdu", from the result of the Web Services request, in MQWW. "Chengdu Fu", "Chengdu Shi" and etc. queried in MQWW database actually return nearly nothing, so I don't put them in Figure 5.6.

Chapter 6 Conclusion

This thesis proposed a new approach to integrate geospatial web services with legacy databases – connecting ontologies directly from the content of geospatial and relational databases and AO for the exposed web services and databases. My proposal established a common prototype for many geographic information systems-related projects that experienced the difficulty in combining RESTful Web Services.

There are still some problems in my design. When the ontology designed has created the ULO, manual mapping should be used to connect it with the AO, or else the AO will only be an isolated database/web service schema. This won't be easy for some larger-scale projects because if the concepts in ULO or AO are too many, it will take incredible time to map the two. If I had more time for the project, I will try to work out an automatic mapping builder to do that. Another problem I missed is the potential semantics that could be found in the names of RESTful Web Services. Although most of the names of these interfaces are not born with special meanings, it is possible to create conventions for REST developers to abide by. In that case, the difficulty in involving RESTful Web Services could be reduced a lot.

6.1 Future Directions

For the SPARQL Query Generator, a few technical problems remain. For example, how could I dump out all the properties of a class in the ontology file? And a following technical question would be, what I do with these innocent properties and form them into an appropriate SPARQL query? The answer to the problem is not very complicated. I will import a well-known library that specifically deals with ontology

files – the Jena Semantic Framework. Generally speaking, my approach is to understand which concept this request belongs to. And dump all the relationships of which the domain is the concept. This ensures that all the information regarding a concept will be discovered. After a collection of the relationships is discovered, the SPARQL Generator needs to chain all the items in the collection to form a huge SPARQL query. Caution should be devoted to eliminate duplicate relationships and new variable generation and storage. I need to store the variables because after I receive the result from Web Services, the Query Broker needs these variables as keys to get the values in the result set, and thus form the result to the user interface.

In a complex situation, I am not limited to such a problem only. Another situation is the chain queries – multiple queries asked at the same time in an advanced mode of query. For example, when the user asks about a person and then the geographic information regarding this person. The challenge to the SPARQL Generator is that, how to validate the correctness of the combination of multiple queries and how to create such an SPARQL query?

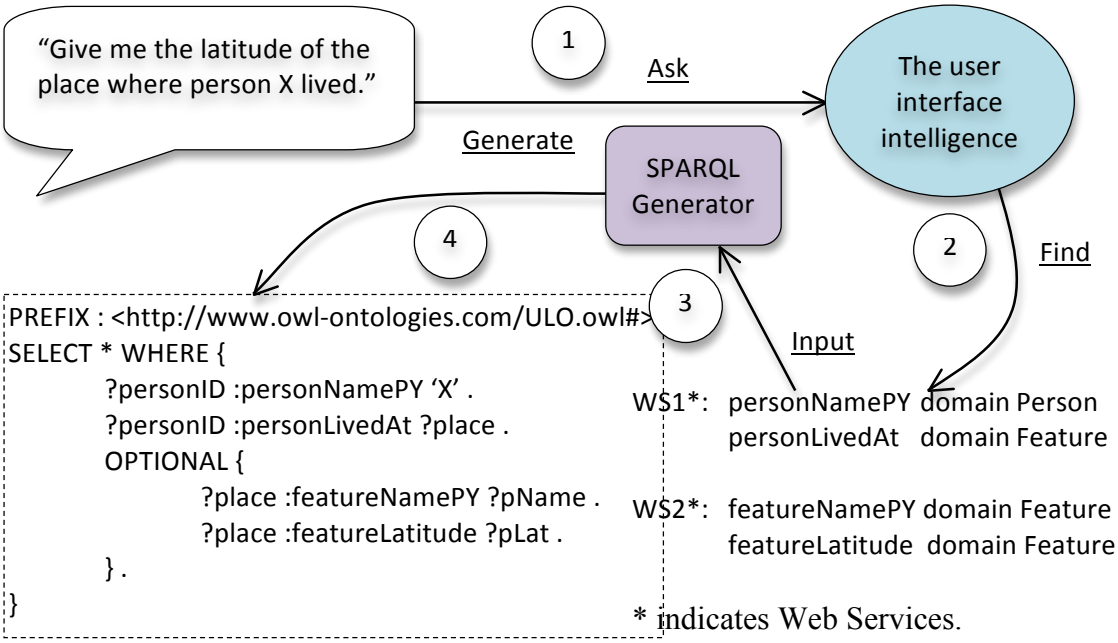


Figure 6.1. Generation of an SPARQL query for the advanced search.

The user interface of the project is still working on the final form of the advanced search. So this is not finalized. Notice that when the SPARQL query is generated, an "OPTIONAL" segment in the code is also added. This is because two different data sources are found. The "OPTIONAL" will enable the SPARQL parser to return result of the segment outside of the "OPTIONAL" even which does not find any thing.

Figure 6.1 represents one possible way to do advanced search. Different queries that share the same range and domain could be considered as associable. When the user finishes the first simple search, he/she will be prompted a list of relationships within the same concept and across concepts. When the SPARQL Generator needs to validate the correctness of the combination, it could query the ontology file to match the domain of a relationship and that of another. Surely this is error prone since in many cases, the two fields involved don't even match in type, e.g. a place ID and a place String. In this case, the generator should check for these trivial details for the correctness.

Now I'd like to consider this query data structure from a programmatic perspective, i.e. the building of the query data structure. Receiving a string representing the SPARQL query from the SPARQL generator, the first problem for the Query Broker is how to recognize the terms in the string. This is not an easy problem since the query is based on pure text without any structure information.

The solution to this problem is to tokenize the text into flexible length n-grams, or terms. There are two approaches – by a tokenizer from an open-source text-processing tool, e.g. lucene is good enough to do so. And the other way is to write a specialized text-processing code segment to generate terms. Comparing these two methods, the

former is more fashionable but it needs special knowledge about writing a robust tokenizer to create flexible length n-grams and, it is subject to further API changes in lucene. So the latter is preferred. Figure 6.2 illustrates a sample implementation of the tokenizer (the sample will not cover all the aspects of the tokenizer).

```
procedure tokenize (query)
        keywords
        list ← new list()
        regexp ← "<.*?>"
        list ← query.match(regexp)
        for each term in list do
                if term.checkIgnoreCase(keywords) then
                        term.upperCase()
                end if
        end for
end procedure
```

Figure 6.2. A sample tokenizer.

Inside the tokenize procedure, keywords is a list of SPARQL keyword string that are previously defined, e.g. SELECT, PREFIX, WHERE, {, }, the dot and etc. The procedure will find out all the terms using the regular expression. The terms are inserted into a list first and are checked if there are any keywords in it. The keywords found in the list will be promoted to uppercase for the sake of convenient processing in the next stage.

More importantly for the query data structure building is to create the Query object with these tokens. Now the token list includes all the necessary data and structure for the parser to begin its work. The problem is to find all the necessary pieces from the token list, to create the corresponding class instances and to build the Query object. I want to first give a pseudo code segment in Figure 6.3 that outlines the order of the major operations (the code is abstracted from D2RQ).

```
procedure buildQuery(tokens)
        Query query
        while not tokens.done() do
                while tokens.get() != "." do
                        if tokens.get().is(Prefix) then
                                query.setPrefix(tokens.next())
                        end if
                        # similar to the above
                        query.setResultVarList()
                        query.setElement()
                end while
        end while
end procedure
```

Figure 6.3. Major operations in the building of a Query object.

The code segment above describes a naive algorithm of creating a Query object given the fact that the creation of the most of the constituents of a Query is actually neglected (for example, modifier to the query – FILTER and etc.). Another thing to notice is these procedures happen in the SPARQLParser class. Last thing to note is the code above is just pseudo code.

References

Andrews, Christopher J. “Emerging Technology: Geospatial Web Services and REST.” *Directions Magazine*, August 2, 2007.

Ankolekar, A.; Burstein, M.; Hobbs, J. R.; Lassila, O.; Martin, D. L.; McDermott, D.; McIlraith, S. A.; Narayanan, S.; Paolucci, M.; Payne, T. R.; and Sycara, K. “DAML-S: Web Service Description for the Semantic Web.” Presented at International Semantic Web Conference (ISWC), Sardinia, Italy, June 9th - 12th, 2000.

ARQ. “A SPARQL Parser for Jena.” Accessed August 26, 2010. http://jena.sourceforge.net/ARQ/.

Barnard, David T., and Ide, Nancy M. “The text encoding initiative: Flexible and extensible document encoding.” Journal of the American Society for Information Science, 1997.

Battle, Robert, and Benson, Edward. “Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST)”. Journal of Web Semantics: 61–69, 2008.

Beaujardiere, Jeff de La, ed. “Web Map Service.” OGC document 04-024, Version 1.3, 2004.

Beckett, David and Berners-Lee, Tim. “Turtle – Terse RDF Triple Language.” W3C Team Submission, January 14, 2008.

Berman, Lex. “CHGIS Web Services Schema,” last modified 2008. CHGIS. Accessed August 26, 2010. http://chgis.hmdc.harvard.edu/xml/chgis.rng.

Bizer, Christian. “D2Rmap - A Database to RDF Mapping Language.” Poster presented at the 12th World Wide Web Conference, Budapest, Hungary, 2003.

Bizer, Christian, and Seaborne, Andy. “D2RQ - treating Non-RDF databases as virtual RDF graphs.” In Proceedings of the 3rd International Semantic Web Conference, 2004.

Bizer, Christian, Cyganiak, Richard. “D2RQ – Lessons Learned.” Position paper for the W3C Workshop on RDF Access to Relational Databases. September 8, 2007.

Borgman, Christine L. “Scholarship in the Digital Age: Information, Infrastructure, and the Internet.” The MIT Press, 2007.

Bray, Tim; Paoli, Jean; Sperberg-McQueen, C. M.; Maler, Eve; and Yergeau, François, eds. “Extensible Markup Language (XML) 1.0, Fifth Edition.” W3C Recommendation 26 November 2008.

Brickley, Dan, and Miller, Libby. “FOAF Vocabulary Specification 0.98.” Namespace Document August 9, 2010.

Brickley, Dan; Guha, R.V.; McBride, Brian. “RDF Vocabulary Description Language 1.0: RDF Schema.” W3C Recommendation February 10, 2004.

Broekstra, J.; Kampman, A.; and Harmelen, F. van. “Sesame: A generic architecture for storing and querying RDF and RDF Schema”. Presented at ISWC, 2002.

Chebotko, A.; Lu, S.; Jamil, H. M.; and Fotouhi, F. “Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns.” Technical Report TR-DB-052006-CLJF. May 2006.

CHGIS. “CHGIS XML API.” Last modified 2008. http://chgis.hmdc.harvard.edu/xml/.

Christensen, E.; Curbera, F.; Meredith, G.; and Weerawarana, S. “Web Services Description Language (WSDL) 1.1.” W3C Note March 15, 2001.

Crawford, William; Farley, Jim; and Flanagan, David. “An Introduction to JDBC, Part 1,” In Java Enterprise in a Nutshell, 2nd Edition. O'Reilly Media, April 2002. Accessed August 26, 2010. http://onjava.com/pub/a/onjava/excerpt/javaentnut_2/index1.html.

Cuenca-Acuna, F. M., and Nguyen, T. D. “Text-based content search and retrieval in ad hoc p2p communities.” Presented at International Workshop on Peer-to-Peer Computing, 2002.

Cyganiak, Richard. “A relational algebra for SPARQL.” Technical Report HPL-2005-170. 2005.

DCMI. “The Dublin Core Metadata Initiative” Accessed December, 2010 http://dublincore.org/

El-Gayar, Omar F.; Sarnikar, Surendra; and Wills, Matthew J.. “A Cyberinfrastructure Framework for Comparative Effectiveness Research in Healthcare.” In Proceedings of the 43rd Hawaii International Conference on System Sciences, 2010.

Fielding, Roy Thomas. “Architectural Styles and the Design of Network-based Software Architectures.” PhD diss., University of California, Irvine, 2000.

Fisher, Matthew; Dean, Mike; Joiner, Greg. “Use of OWL and SWRL for Semantic Relational Database Translation.” OWLED 2008.

Foltz, Peter W. “Latent semantic analysis for text-based research, Behavior Research Methods.” Instruments & Computers, 1996.

Fong, Grace, “Proposal for the ICHD project.” SSHRC-IOF, 11 September 2007.

Gietz, P.; Aschenbrenner, A.; Budenbender, S.; Jannidis, F.; Kuster, M.W.; Ludwig, C.; Pempe, W.; Vitt, T.; Wegstein, W.; Zielinski, A. “TextGrid and eHumanities.” e-Science and Grid Computing, 2006. e-Science '06. Second IEEE International Conference on 133 - 133, Dec. 2006.

Gudgin, Martin; Hadley, Marc; Mendelsohn, Noah; Moreau, Jean-Jacques; Nielsen, Henrik Frystyk; Karmarkar, Anish; and Lafon, Yve. “SOAP Version 1.2 Part 1: Messaging Framework, Second Edition.” W3C Recommendation April 27, 2007.

Gruber, Thomas R. "Toward Principles for the Design of Ontologies Used for Knowledge Sharing." Presented at *International Journal Human-Computer Studies*, 43(5-6):907-928, 1995.

Hadley, Marc J. “Web Application Description Language (WADL).” W3C Member Submission August 31, 2009.

Haas, Hugo, and Brown, Allen. “Web Services Glossary.” W3C Working Group Note February 11, 2004.

Hebeler, John, and Fisher, Matt, “LAB-4449: Semantic Web Programming”, Tutorial presented at JavaOne 2009.

Hewlett-Packard Labs. “Jena Semantic Framework.” Accessed August 26, 2010. http://jena.sourceforge.net/.

Hobona, Gobe; Faribairn, David; and James, Philip. “Workflow Enactment of GridEnabled Geospatial Web Services.” Paper presented at the 6th UK e-Science All Hands Meeting, Nottingham, U.K., 2007.

Java-Source. “Open Source XML Parsers in Java”. Accessed August 26, 2010. http://java-source.net/open-source/xml-parsers,

JDOM. “JDOM Overview.” Accessed August 26, 2010. http://www.jdom.org/.

Jones, Christopher B.; Abdelmoty, Alia I.; Finch, David; Fu, Gaihua; and Vaid, Subodh, “The SPIRIT Spatial Search Engine:Architecture, Ontologies and Spatial Indexing.” In Proceedings of the Third International Conference on Geographic Information Science GIScience, Maryland, USA, 2004.

Li, Wenwen, and Yang, Chaowei. “Semantic Search Engine for Spatial Web Portals.” *Geoscience and Remote Sensing Symposium, 2008*. IEEE International Volume 2 pp. 1278-1281, 2008.

Mack, R., Hehenberger, M. “Text-based knowledge discovery: search and mining of life-sciences documents.” Drug Discovery Today, 2002.

Martin, D.; Burstein, M.; Hobbs, J.; Lassila, O.; McDermott, D.; McIlraith, S.; Narayanan, S.; Paolucci, M.; Parsia, B.; Payne, T. R.; Sirin, E.; Srinivasan, N.; and Sycara, K. “OWL-S: Semantic Markup for Web Services.” W3C Member Submission November 22, 2004.

McCarty, Willard. “Humanities Computing” Encyclopedia of Library and Information Science, June 23, 2003.

McGuinness, Deborah L., and Harmelen, Frank van. “OWL Web Ontology Language Overview.” W3C Recommendation February 10, 2004.

OpenLink Software. “Universal Server Platform for the Real-Time Enterprise.” Accessed August 26, 2010. http://www.openlinksw.com/virtuoso/index.html.

Pan, Zhengxiang, and Heflin, Jeff. “DLDB: Extending relational databases to support Semantic Web queries.” Technical Report LU-CSE-04-006, Dept. of Computer Science and Engineering, Lehigh University, 2004.

Paolucci, M., Sycara, K. "Autonomous Semantic Web services." *Internet Computing* (IEEE) 7: 34–41, 2003.

Paul, Manoj, and Ghosh, S.K. “An approach for service oriented discovery and retrieval of spatial data.” In Proceedings of the 2006 international workshop on Service-oriented software engineering, pp. 88–94, 2006.

Prud'hommeaux, Eric. “Optimal RDF Access to Relational Databases.” Last modified Aprial 30, 2004. http://www.w3.org/2004/04/30-RDF-RDB-access/.

Prud'hommeaux, Eric, and Seaborne, Andy, eds. “SPARQL Query Language for RDF.” W3C Recommendation 15 January 2008. Accessed August 26, 2010. http://www.w3.org/TR/rdf-sparql-query/.

Sieber, Renee E.; Wellen, Christopher C.; and Jin, Yuan. “Spatial CIs, ontologies and the humanities.” Forthcoming at Proceedings of the National Academy of Sciences Special Feature on Cyber Infrastructure.

Schmidt, Christopher, message to “[Geowanking] Critiques of WFS, WMS”, January 24, 2010. http://www.mail-archive.com/geowanking@geowanking.org/msg01222.html.

Short, Harold. “The Role of Humanities Computing: Experiences and Challenges.” Literary and Linguistic Computing Vol. 21, No. 1, 2006.

Turner, Andrew, email message to Renee Sieber’s call for “Critiques for WFS/WMS”, January 24, 2010.

Unsworth, John. “My Cultural Commonwealth.” Report of the American Council of Learned Societies’ Commission on Cyberinfrastructure for the Humanities and Social Sciences. New York: American Council of Learned Societies, 2006.

Vretanos, Panagiotis A., ed. “Web Feature Service Implementation Specification.” OGC document 04-094, Version 1.1.0, 2005.

Wikipedia Contributors. “Knowledge Acquisition (K.A.).” Wikipedia, The Free Encyclopedia. Accessed August 26, 2010. http://en.wikipedia.org/wiki/Knowledge_management.

Wikipedia Contributors. “Web Ontology Language.” Wikipedia, The Free Encyclopedia. Accessed August 26, 2010. http://en.wikipedia.org/wiki/Web_Ontology_Language,

Wikipedia Contributors. “Protégé (Software).” Wikipedia, The Free Encyclopedia. Accessed August 26, 2010. http://en.wikipedia.org/wiki/Protégé_(software).

Wikipedia Contributors. “Representation State Transfer.” Wikipedia, The Free Encyclopedia. Accessed August 26, 2010. http://en.wikipedia.org/wiki/Representational_State_Transfer.

Wikipedia Contributors. “Resource Description Framework.” Wikipedia, The Free Encyclopedia. Accessed August 26, 2010. http://en.wikipedia.org/wiki/Resource_Description_Framework.

Zhao, Rong, and Grosky, William I. “Narrowing the semantic gap-improved text-based web document retrieval using visual features.” IEEE Transactions on Multimedia, 2002.

Zhao, Tian; Zhang, Chuanrong; Wei, Mingzheng; and Peng, Zhong-Ren. “Ontology-based geospatial data query and integration.” Lecture notes in Computer Science for the Fifth International Conference on Geographic Information Science 2008, Vol. 5266, 370-392.