Practical Semantic Web and Linked Data Applications

Java, JRuby, Scala, and Clojure Edition

Mark Watson

March 12, 2011

Contents

List of Figures

List of Tables

Preface

This book is intended to be a practical guide for using RDF data in information processing, linked data, and semantic web applications using both the AllegroGraph product and the Sesame open source project. RDF data represents a graph. You probably are familiar to at least some extent with graph theory from computer science. Graphs are a natural way to represent things and the relationships between them. RDF data stores are optimized to efficiently recognize graph sub-patterns[1] and there is a standard query language SPARQL that we will use to query RDF graph data stores. You will learn how to use SPARQL first with simple examples and later by using SPARQL in applications.

This book will show you how to effectively use AllegroGraph, a commercial product written and supported by Franz and the open source Sesame platform. While AllegroGraph itself is written in Common Lisp, this book is primarily written for programmers using either Java or other JVM languages like Scala, Clojure, and JRuby. A separate edition of this book covers using AllegroGraph in Lisp applications.

I take an unusual approach in both Java and Lisp editions of this book. Instead of digging too deeply into proprietary APIs for available data stores (for example, AllegroGraph, Jena, Sesame, 4Store, etc.) we will concentrate on a more standards-based approach: we will deal with RDF data stored in easy to read N-Triple and N3 formats and perform all queries using the standard SPARQL query language. I am more interested in showing you how to model data with RDF and write practical applications than in covering specific tools that already have sufficient documentation.

While I cover most of the Java AllegroGraph client APIs provided by Franz, my approach is to introduce these APIs and then write a Java wrapper that covers most of the underlying functionality but is, I think, easier to use. I also provide my wrapper in Scala, Clojure, and JRuby versions. Once you understand the functionality of AllegroGraph and work through the examples in this book, you should be able to use any combination of Java, Scala, Closure, and JRuby to develop information processing applications.

I have another motivation for writing my own wrapper: I use both AllegroGraph and the open source Sesame system for my own projects. I did some extra work so my

[1] Other types of graph data stores like Neo4j are optimized to traverse graphs. Given a starting node you can efficiently traverse the graph in the region around that node. In this book we will concentrate on applications that use sub-graph matching.

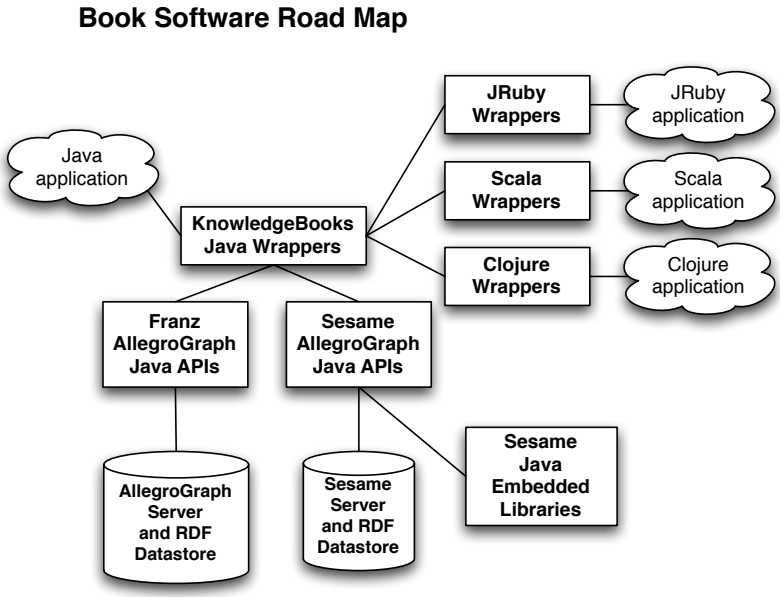

Figure 1.: Software developed and used in this book

wrapper also supports Sesame (including my own support for geolocation). You can develop using my wrapper and Sesame and then deploy using either AllegroGraph or Sesame. I appreciate this flexibility and you probably will also.

Figure 1 shows the general architecture roadmap of the software developed and used in this book.

AllegroGraph is written in Common Lisp and comes in several ”flavors”:

1. As a standalone server that supports Lisp, Ruby, Java, Clojure, Scala, and Python clients. A free version (limited to 50 million RDF triples - a large limit) that can be used for any purpose, including commercial use. This book (the Java, Scala, Clojure, and JRuby edition) uses the server version of AllegroGraph.

2. The WebView interface for exploring, querying, and managing AllegroGraph triple stores. WebView is standalone because it contains an embedded AllegroGraph server. You can see examples of AGWebView in Section 16.4.

3. The Gruff for exploring, querying, and managing AllegroGraph triple stores using table and graph views. Gruff is standalone because it contains an embedded AllegroGraph server. I use Gruff throughout this book to generate screenshots of RDF graphs.

4. AllegrGraph is compatible with several other commercial products: TopBraid Composer, IO Informatics Sentient, and RacerSystems RacerPorter.

5. A library that is used embedded in Franz Common Lisp applications. A free version is available (with some limitations) for non-commercial use. I covered this library in the Common Lisp edition of this book.

Sesame is an open source (BSD style license) project that provides an efficient RDF data store, support for the standard SPARQL query language, and deployment as either an embedded Java library or as a web service. Unlike AllegroGraph, Sesame does not natively support geolocation and free text indexing, but my KnowledgeBooks Java Wrapper adds this support so for the purposes of this book, you can run the examples using either AllegroGraph or Sesame "back ends."

Most of the programming examples will use the Java client APIs so this book will be of most interest to Java, JRuby, Clojure, and Scala developers. I assume that most readers will have both the free server version of AllegroGraph and Sesame installed. However, the material in this book is also relevant to writing applications using the very large data store capabilities of the commercial version of AllegroGraph.

Regardless of which programming languages that you use, the basic techniques of using AllegroGraph are very similar.

The example code snippets and example applications and libraries in this book are licensed using the AGPL. As an individual developer, if you purchase the either the print edition of this book or purchase the for-fee PDF book, then I give you a commercial use waiver to the AGPL deploying your applications: you can use my examples in commercial applications without the requirement of releasing the source code for your application under the AGPL. If you work for a company that would like use my examples with a commercial use waiver, then have your company purchase two print copies of this book for use by your development team. Both the AGPL and my own commercial use licenses are included with the source code for this book.

Acknowledgements

I would like to thank my wife Carol Watson for editing this book. I would like to thank Alex Ott for text corrections and improvements in the Clojure code examples. I would also like to thank the developers of the software that I use in this book: AllegroGraph, Sesame, Lucene, JavaDB, and D2R.

Part I.

Introduction to AllegroGraph and Sesame

1. Introduction

Franz has good online documentation for all of their AllegroGraph products and the Sesame open source project also has good online documentation. While I do not duplicate the available documentation, I do aim to make this book self contained, providing you with an introduction to AllegroGraph and Sesame. The broader purpose of this book is to provide application programming examples using RDF and RDFS data models and data stores. I also covers some of my own open source projects that you may find useful for Semantic Web and general information processing applications.

AllegroGraph is an RDF data repository that can use RDFS and RDFS+ inferencing. AllegroGraph also provides three non-standard extensions:

1. Test indexing and search
2. Geo Location support
3. Network traversal and search for social network applications

I provide you with a wrapper for Sesame that adds text indexing and search, and geo location support.

1.1. Why use RDF?

We may use many different types of data storage in our work and research, including:

1. Relational Databases
2. NoSQL document-based systems (for example, MongoDB and CouchDB)
3. NoSQL key/value systems (for example, Redis, MemcacheDB, SimpleDB, Voldemort, Dynamo[1], Big Table, and Linda style tuple stores)
4. RDF data stores

I would guess that you are most familiar with the use of relational database systems but NoSQL and RDF type data stores are becoming more commonly used. Although I

[1] SimpleDB, Voldemort and Dynamo are ”eventually consistent” so readers do not always see the most current writes but they are easier to scale.

have used NoSQL data stores like MongoDB, CouchDB, and SimpleDB on projects I am not going to cover them here except to say that they share some of the benefits of RDF data stores: no pre-defined schema required[2] and decentralized data store without having to resort to sharding. AllegroGraph and Sesame can also be used for general purpose graph-based applications[3].

The biggest advantages of using RDF are:

1. RDF and RDFS (the RDF Schema language) are standards, as is the more descriptive Web Ontology Language (OWL) that is built on RDF and RDFS and offers richer class and property modeling and inferencing.[4] The SPARQL query language is a standard and is roughly similar to SQL except that it matches patterns in graphs rather than in related database tables.

2. More flexibility: defining properties used with classes is similar to defining the columns in a relational database table. However, you do not need to define properties for every instance of a class. This is analogous to a database table that can be missing columns for rows that do not have values for these columns (a sparse data representation). Furthermore, you can make ad hoc RDF statements about any resource without the need to update global schemas. SPARQL queries can contain optional matching clauses that work well with sparse data representations.

3. Shared Ontologies facilitate merging data from different sources.

4. Being based on proven Internet protocols like HTTP naturally supports web-wide scaling.

5. RDF and RDFS inference creates new information automatically about such things as class membership. Inference is supported by several different logics. Inference supports merging data that is defined using different Ontologies or schemas by making statements about the equivalence of classes and properties.

6. There is a rich and growing corpus of RDF data on the web that can be used as-is or merged with proprietary data to increase the value of in-house data stores.

7. Graph theory is well understood and some types of problems are better solved using graph data structures (more on this topic in Section 1.7)

[2] I argue that this increases the agility of developing systems: you can quickly add attributes to documents and add RDF statements about existing things in an RDF data store

[3] Like Neo4j

[4] I am not covering OWL in this book. However, AllegroGraph supports RDFS++ which is a very useful subset of OWL. There are backend OWL reasoners for Sesame available but I will not use them in this book. I believe that the ”low hanging fruit” for using Semantic Web and Linked Data applications can be had using RDF and RDFS. RDF and RDFS have an easier learning curve than does OWL.

1.2. Who is this Book Written for?

I wrote this book to give you a quick start for learning how to write applications that take advantage of Semantic Web and Linked Data technologies. I also hope that you have fun with the examples in this book and get ideas for your own projects. You can use either the open source Sesame project or the commercially supported AllegroGraph product as you work through this book. I recommend that you try using them both, even though almost all of the examples in this book will work using either one.

AllegroGraph is a powerful tool for handling large amounts of data. This book focuses mostly on Java clients and I also provide wrappers so that you can also easily use JRuby, Clojure, and Scala. Franz documentation covers writing clients in Python and C-Ruby and I will not be covering these languages.

Since AllegroGraph is implemented is Common Lisp, Franz also provides support for embedding AllegroGraph in Lisp applications. The Common Lisp edition of this book covers embedded Lisp use. If you are a Lisp developer then you should probably be reading the Lisp edition of this book.

If you own a AllegroGraph development license, then you are set to go, as far as using this book. If not, you need to download and install a free edition copy at:

http://www.franz.com/downloads/

You might also want download and install the free versions of the standalone server, Gruff (Section 2.3.2), and WebView (Section 2.3.1).

You can download Sesame from http://openrdf.org and also access the online documentation.

1.3. Why is a PDF Copy of this Book Available Free on My Web Site?

As an author I want to both earn a living writing and have many people read and enjoy my books. By offering for sale the print version of this book I can earn some money for my efforts and also allow readers who can not afford to buy many books or may only be interested in a few chapters of this book to read it from the free PDF on my web site.

Please note that I do not give permission to post the PDF version of this book on other people's web sites. I consider this to be at least indirectly commercial exploitation in violation the Creative Commons License that I have chosen for this book.

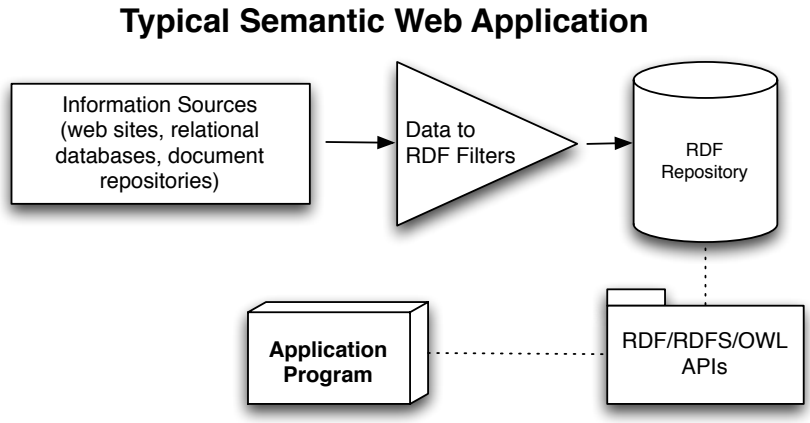

Figure 1.1.: Example Semantic Web Application

As I mentioned in the Preface, if you purchase a print copy of this book then I grant you a "AGPL waiver" so that you can use the book example code in your own projects without the requirement of licensing your code using the AGPL. (See the commercial use software license on my web site or read the copy included with the example code for this book.)

1.4. Book Software

You can get both the KnowledgeBooks Sesame/AllegroGraph wrapper library and the book example applications from the following git repository:

```
git clone \\
http://github.com/mark-watson/java_practical_semantic_web.git
```

This git repository also contains the version of my NLP library seen in Chapter 12 and all of the other utilities developed in this book.

1.5. Important Notes on Using the Book Examples

All of the examples can be run and experimented with using either the AllegroGraph back end or My Sesame back end. If you are using the free version of AllegroGraph and you need to set some environment variables to define a connection with the server:

```
ALLEGROGRAPH_SERVER=localhost # or an IP address of
                              # a remote server
ALLEGROGRAPH_PORT=10035
ALLEGROGRAPH_USERNAME=root
ALLEGROGRAPH_PASSWD=z8dj3jk7dqa
```

You should set the username and password to match what you used when installing and setting up AllegroGraph following Franz's directions.

You can set these environment variables in your .profile file for OS X, in your .bashrc or .profile file for Linux, or using "Edit System Environment Variables" on Windows 7.

If you don't set these values then you will get a runtime error followed by a message telling you which environment variables were not set. Some Java IDEs like IntelliJ do not "pick up" system environment variables so you will have to set them per project in the IDE.

If you want to use Sesame and my wrappers for Java, Scala, JRuby, and Clojure, then you are already set up if you fetched the git repository for this book because I have the required JAR files in the repository.

1.6. Organization of this Book

The book examples are organized in subdirectories organized by topic:

- Part I contains an overview of AllegroGraph and Sesame including code samples for calling the native AllegroGraph and Sesame APIs.
- Part II implements high level wrappers for AllegroGraph and Sesame including code examples in Java, Scala, Clojure, and JRuby.
- Part III provides you with an overview of Semantic Web Technologies: RDF, RDFS, SPARQL query language, and linked data.

- Part IV contains utilities for information processing and ends with a large application example. I cover web spidering, Open Calais, my library for Natural Language Processing (NLP), Freebase, SPARQL client for DBpedia, and the GeoNames web services.

1.7. Why Graph Data Representations are Better than the Relational Database Model for Dealing with Rapidly Changing Data Requirements

When people are first introduced to Semantic Web technologies their first reaction is often something like, "I can just do that with a database." The relational database model is an efficient way to express and work with slowly changing data models. There are some clever tools for dealing with data change requirements in the database world (ActiveRecord and migrations being a good example) but it is awkward to have end users and even developers tagging on new data attributes to relational database tables.

A major theme in this book is convincing you that modeling data with RDF and RDFS facilitates freely extending data models and also allows fairly easy integration of data from different sources using different schemas without explicitly converting data from one schema to another for reuse. You will learn how to use the SPARQL query language to use information in different RDF repositories. It is also possible to publish relational data with a SPARQL interface.[5]

1.8. Wrap Up

Before proceeding to the next two chapters I recommend that you take the time to set up your development system so that you can follow along with the examples. Chapter 2 will give you an overview of AllegroGraph while Chapter 3 will introduce you to the Sesame platform.

The first part of this book is very hands on: I'll give you a quick introduction to AllegroGraph and Sesame via short example programs and later the implementation of my wrapper that allows you to use AllegroGraph and Sesame using the same APIs. In Chapter 6 I will cover Semantic Web technologies from a more theoretical and

[5] The open source D2R project (see Appendix B for information on setting up D2R) provides a wrapper for relational databases that provides a SPARQL query interface. If you have existing relational databases that you want to use with RDF data stores then I recommend using D2R.

reference point of view. The book will end with information gathering and processing tools for public lined data sources and larger example applications.

2. An Overview of AllegroGraph

This chapter will show you how to start the AllegroGraph server on a Linux laptop or server and use the AllegroGraph Java APIs with some small example programs. In Chapters 4 and 5, I will wrap these APIs and the Sesame RDF data store APIs in a common wrapper so that the remaining example programs in this book will work with either the AllegroGraph or Sesame back ends and you will be able to use my Scala, Clojure, or JRuby wrappers if you prefer a more concise (or alternative) language to Java.

2.1. Starting AllegroGraph

When you downloaded a copy of the AllegroGraph server from Franz's web site, there were installation instructions provided for 64-bit editions of Linux, Windows, and OS X. Note that AllegroGraph version 4 specifically requires a 64-bit operating system.[1][2]

When you run the installation script assign a non-obvious password for your AllegroGraph root account. This is especially important if you are installing the server on a public server. I use the following commands to start and stop the AllegroGraph service on my Linux server:

```
cd /home/mark/agraph-4.0/
agraph-control --config /home/mark/AG/agraph.cfg start
agraph-control --config /home/mark/AG/agraph.cfg stop
```

[1] While writing this book, I kept AllegroGraph running on a low cost 64-bit Linux VPS (I use RimuHosting, but most Linux hosting companies also support 64-bit kernels). Because I work using laptops (usually Ubuntu Linux and OS X, sometimes Windows 7) I find it convenient keeping server processes like AllegroGraph, MongoDB, PostgreSQL, etc. running on separate servers so these services are always available during development and deployment small systems. Commercial VPS hosting and Amazon EC2 instances are inexpensive enough that I have given up running my own servers in my home office.

[2] Initially, only the Linux 64 bit edition will be available, followed later with the Windows and OS X editions.

2.1.1. Security

For my purposes developing this book I was initially satisfied with the security from using a long and non-obvious password on a small dedicated server. If you are going to be running AllegroGraph on a public server that contains sensitive information you might want to install it for local access only when running the installation script and then use a SSH tunnel to remotely access it; for example:

```
ssh -i ~/.ssh/id_rsa-gsg-keypair  \\
    -L 10035:localhost:10035 \\
    mark@agtest123.com
```

Here I assume that you have SSH installed on both your laptop and your remote server and that you have copied your public key to the server. I often use SSH tunnels for secure access of remote CouchDB, MongoDB, etc. services.

2.2. Working with RDF Data Stores

Chapter 6 will provide an introduction to RDF data modeling.[3] For now, it is enough to know that RDF triples have three parts: a subject, predicate, and object. Subjects and predicates are almost always web URIs while an object can be a typed literal value or a URI.

RDF data stores provide the services for storing RDF triple data and provide some means of making queries to identify some subset of the triples in the store. I think that it is important to keep in mind that the mechanism for maintaining triple stores varies in different implementations. Triples can be stored in memory, in disk-based btree stores like BerkeleyDB, in relational databases, and in custom stores like AllegroGraph. While much of this book is specific to Sesame and AllegroGraph the concepts that you will learn and experiment with can be useful if you also use other languages and platforms like Java (Sesame, Jena, OwlAPIs, etc.), Ruby (Redland RDF), etc. For Java developers Franz offers a Java version of AllegroGraph (implemented in Lisp with a network interface that also supports Python and Ruby clients) that I will be using in this book and that you now have installed so that you can follow along with my examples.

The following sections will give you a brief overview of Franz's Java APIs and we will take a closer look in Chapter 4. After developing a wrapper in Chapter 4, we will use the wrapper in the rest of this book.

[3] I considered covering the more formal aspects of RDF and RDFS early in this book but decided that most people would like to see example code early on. You might want to read through to Chapters 6 and 7 now if you have never worked with any Semantic Web technologies before and do not know what RDF and RDFS are.

2.2.1. Connecting to a Server and Creating Repositories

The code in this section uses the Franz Java APIs. While it is important for you to be familiar with the Franz APIs, I will be writing an easier to use wrapper class in Chapter 4 that we will be using in the remainder of this book.

The Java class AGServer acts as a proxy to communicate with a remote server:

```
String host = "example.com";
int port = 10035;
String username = "root";
String password = "kjfdsji7rfs";
AGServer server =
  new AGServer("http://" + host + ":" + port,
               userName, password);
```

Once a connection is made, then we can make a factory root catalog object that we can use, for example, to create a new repository and RDF triples. I am using the SPARQL query language to retrieve triples from the datastore. We will look at SPARQL in some depth in Chapter 8.

```
AGCatalog rootCatalog = server.getRootCatalog();
AGRepository currentRepository =
  rootCatalog.createRepository("new-repo-1");
AGRepositoryConnection conn =
  currentRepository.getConnection();
AGValueFactory valueFactory =
  conn.getRepository().getValueFactory();

// register a predicate for full text
// indexing and search:
conn.registerFreetextPredicate(valueFactory.
       createURI("http://example.org/ontology/name"));

// create a RDF triple:
URI subject = valueFactory.
       createURI("http://example.org/people/mark");
URI predicate = valueFactory.
       createURI(http://example.org/ontology/name");
String object = "Mark Watson;
conn.add(subject, predicate, object);

// perform a SPARQL query:
```

```
String query =
  "SELECT ?s ?p ?o WHERE {?s ?p ?o .}";
TupleQuery tupleQuery = conn.
      prepareTupleQuery(QueryLanguage.SPARQL, sparql);
TupleQueryResult result = tupleQuery.evaluate();
try {
  List<String> bindingNames =
               result.getBindingNames();
  while (result.hasNext()) {
    BindingSet bindingSet = result.next();
    int size2 = bindingSet.size();
    ArrayList<String> vals =
               new ArrayList<String>(size2);
    for (int i=0; i<size2; i++)
      String variable_name = bindingNames.get(i));
      String variable_value = bindingSet.
          getValue(variable_name).stringValue();
      System.out.println(" var: " + variable_name +
          ", val: " + variable_value);
    }
} finally {
  result.close();
}
```

2.2.2. Support for Free Text Indexing and Search

The AllegroGraph support for free text indexing is very useful and we will use it often in this book. The example code snippets use the same setup code used in the last example - only the SPARQL query string is different:

```
// using free text search; substitute the SPARQL
// query string, and re-run the last exaple:
String query =
  "SELECT ?s ?p ?o
   WHERE { ?s ?p ?o . ?s fti:match 'Mark*' . }";
```

The SPARQL language allows you to add external functions that can be used in matching conditions. Here Franz has defined a function fti:match that interfaces with their custom text index and search functionality. I will be wrapping text search both to make it slightly easier to use and also for compatibility with my text indexing and search wrapper for Sesame. We will not be using the fti:match function in the remainder of this book.

2.2.3. Support for Geo Location

Geo Location support in AllegroGraph is more general than 2D map coordinates or other 2D coordinate systems. I will be wrapping Geo Location search and using my wrapper for later examples in this book. Here I will briefly introduce you to the Geo Location APIs and then refer you to Franz's online documentation.

```
// geolocation example: start with a one-time
// initialization for this repository:
URI location = valueFactory.
   createURI("http://knowledgebooks.com/rdf/location");
// specify a resolution of 5 miles, and units in degrees:
URI sphericalSystemDegree =
  conn.registerSphericalType(5f, "degree");

// create a geolocation RDF triple:
URI subject = valueFactory.
     createURI("http://example.org/people/mark");
URI predicate = location; // reuse the URI location
float latitude = 37.81385;
float longitude = -122.3230;
String object = valueFactory.
      createLiteral(latitude + longitude,
                    sphericalSystemDegree);
conn.add(subject, predicate, object);

// perform a geolocation query:
URI location = valueFactory.
    createURI("http://knowledgebooks.com/rdf/location");
float latitude = 37.7;
float longitude = -122.4;
float radius_in_km = 800f;
RepositoryResult<Statement> result =
   conn.getGeoHaversine(sphericalSystemDegree, location,
                        latitude, longitude, radius_in_km,
                        "km", 0, false);
try {
  while (result.hasNext()) {
    Statement statement = result.next();
    Value s = statement.getSubject();
    Value p = statement.getPredicate();
    Value o = statement.getObject();
    System.out.println("subject: " + s +
                       ", predicate: " + p +
```

```
                          ", object: " + o);)
} finally {
  result.close();
}
```

We will be using Geo Location later in this book.

2.3. Other AllegroGraph-based Products

Franz has auxiliary products that extend AllegroGraph adding a web service interface (WebView) and an interactive RDF graph browser (Gruff).

2.3.1. AllegroGraph AGWebView

AGWebView is packaged with the AllegroGraph server. After installing AllegroGraph 4.0 server, you can open a browser at http://localhost:10035 to use AGWebView.

I will be using AGWebView in Chapter 16 to show generated RDF data. You might want to use it instead of or in addition to AllegroGraph if you would like a web-based RDF browser and administration tool for managing RDF repositories. AGWebView is available for Linux, Windows, and OS X[4].

2.3.2. Gruff

Gruff is an interactive RDF viewer and editor. I use Gruff to create several screen shot figures later in this book; for example Figure 11.1. When you generate or otherwise collect RDF triple data then Gruff is a good tool to visually explore it. Gruff is only available for Linux and requires AllegroGraph 4.[5]

2.4. Comparing AllegroGraph With Other Semantic Web Frameworks

Although this book is about developing Semantic Web applications using just AllegroGraph and/or Sesame, it is also worthwhile looking at alternative technologies that you

[4] Initially available for Linux, followed by Windows and OS X.

[5] As an alternative to using Gruff, you can use the open source GrapViz program to generate technical figures showing RDF graphs. I covered this in my book "Scripting Intelligence, Web 3.0 Information Gathering and Processing" [Watson 2009, Apress/Springer-Verlag, pages 145-149]

can use. The alternative technology that I have used for Semantic Web applications is Swi-Prolog with its Semantic Web libraries (open source, LGPL). Swi-Prolog is an excellent tool for experimenting and learning about the Semantic Web. The Java Jena toolkit is also widely used. These alternatives have the advantage of being free to use but lack advantages of scalability and utility that a commercial product like AllegroGraph has.

Although I do not cover OpenLink Virtuoso, you might want to check out either the open source or commercial version. OpenLink Virtuoso is used to host the public SPARQL endpoint for the DBPedia linked data web service that I will use later in two example programs.

2.5. AllegroGraph Overview Wrap Up

This short chapter gave you a brief introduction to running AllegroGraph as a service and showed some Java client code snippets to introduce you to the most commonly used Franz client APIs.

Before implementing a Java wrapper for the AllegroGraph in Chapter 4, we will first take a look at the Sesame toolkit in the next chapter. If you are do not plan on using Sesame, at least in the near term, then you can skip directly to Chapter 4 where I develop the wrapper for Franz's Java APIs.

AllegroGraph is a great platform for building Semantic Web Applications and I encourage you to more fully explore the online dcoumentation. There are interesting and useful aspects of AllegroGraph (e.g., federated AllegroGraph instances on multiple servers) that I will not be covering in this book.

3. An Overview of Sesame

There are several very good open source RDF data stores but Sesame is the one I use the most. I include the Sesame JAR file and all dependencies with the examples for this book. However, you will want to visit the Sesame web site at www.openrdf.org for newer versions of the software and online documentation.

Sesame has a liberal BSD style license so it can be used without cost in commercial applications. I find that Sesame and AllegroGraph are complementary: AllegroGraph provides more features and more scalability but when I use my compatibility wrapper library (see Chapters 4 and 5) I can enjoy using AllegroGraph with the assurance that I have flexibility of also using Sesame as needed.

Sesame is used in two modes: as an embedded component in a Java application and as a web service. We will look at both uses in the next two sections but my wrapper library assumes embedded use.

Sesame is an RDF data store with RDF Schema (RDFS) inferencing and query capability. AllegroGraph also supports RDFS inferencing and queries, but adds some features[1] of the Web Ontology Language (OWL) so query results may differ using Sesame or AllegroGraph on identical RDF data sets. Out of the box Sesame has a weaker reasoning capability than AllegroGraph but optional Sesame backends support full OWL reasoning if you need it.[2]

3.1. Using Sesame Embedded in Java Applications

You can refer to the source file SesameEmbeddedProxy.java for a complete example for embedding Sesame. In this section I will cover just the basics. The following code snippet shows how to create an RDF data store that is persisted to the local file system:

```
// index subject, predicate, and objects in triples
// for faster access (but slower inserts):
String indexes = "spoc,posc,cosp";
```

[1] AllegroGraph supports RDFS++ reasoning.
[2] We will not use OWL in this book.

```
// open a repository that is file based:
org.openrdf.repository.Repository myRepository =
  new org.openrdf.repository.sail.SailRepository(
    new org.openrdf.sail.inferencer.fc.
              ForwardChainingRDFSInferencer(
      new org.openrdf.sail.nativerdf.
            NativeStore("/tmp/rdf", indexes)));
myRepository.initialize();
Connection con = myRepository.getConnection();
// a value factory can be made to construct Literals:
ValueFactory valueFactory =
    con.getRepository().getValueFactory();
// add a triple in N-Triples format defined
// as a string value:
StringReader sr = new StringReader(
 "<http://example.org/people/mark> \\
    <http://example.org/ontology/name> "Mark" .");
conn.add(sr, "", RDFFormat.NTRIPLES);
// example SPARQL query:
String sparql_query =
  "SELECT ?s ?o WHERE \\
    { ?s <http://example.org/ontology/name> ?o .}";
org.openrdf.query.TupleQuery tupleQuery =
  con.prepareTupleQuery(
    org.openrdf.query.QueryLanguage.SPARQL,
    sparql_query);
TupleQueryResult result = tupleQuery.evaluate();
List<String> bindingNames = result.getBindingNames();
while (result.hasNext()) {
  BindingSet bindingSet = result.next();
  int size2 = bindingSet.size();
  ArrayList<String> vals = new ArrayList<String>(size2);
  for (int i=0; i<size2; i++) {
     String variable_name = bindingNames.get(i);
     String variable_value =
        bindingSet.getValue(
           bindingNames.get(i)).stringValue());
     System.out.println(
        variable_name + ": " + variable_value);
  }
}
```

There is some overhead in making SPARQL queries that can be avoided using the native Sesame APIs. This is similar to using JDBC prepared statements when querying

a relational database. For most of my work I prefer to use SPARQL queries and 'live with' the slight loss of runtime performance. After a small learning curve, SPARQL is fairly portable and easy to work with. We will look at SPARQL in some depth in Chapter 8.

3.2. Using Sesame Web Services

The Sesame web server supports REST style web service calls. AllegroGraph also supports this Sesame HTTP communications protocol. The Sesame online User Guide documents how to set up and run Sesame as a web service. I keep both a Sesame server instance and an AllegroGraph server instance running 24/7 on a server so I don't have to keep them running on my laptop while I am writing code that uses them. I recommend that you run at least one RDF data store service; if it is always available then you will be more inclined to use a non-relational data store in our applications when it makes sense to do so.

You saw an example of using the AllegroGraph web interface in Section 2.3.1. I am not going to cover the Sesame web interface in any detail, but it is simple to install:

- Download a binary Tomcat server distribution from tomcat.apache.org
- Install Tomcat
- Copy the sesame.war file from the full Sesame distribution to the TOMCAT/webapps directory
- Start Tomcat
- Access the Sesame admin console at http://localhost:8080/openrdf-sesame
- Access the Sesame work bench console at http://localhost:8080/openrdf-workbench

I cover the Sesame web service and other RDF data stores in my book [Watson, 2009][3]

3.3. Wrap Up

This short Chapter has provided you with enough background to understand the implementation of my Sesame wrapper in Chapter 5. Sesame is a great platform for building Semantic Web Applications and I encourage you to more fully explore the online Sesame documentation.

[3] "Scripting Intelligence, Web 3.0 Information Gathering and Processing" Apress/Springer-Verlag 2009

Part II.

Implementing High Level Wrappers for AllegroGraph and Sesame

4. An API Wrapper for AllegroGraph Clients

We have looked at Java client code that directly uses the Franz AllegroGraph APIs in Chapter 2. I will implement my own wrapper APIs for AllegroGraph in this chapter and in Chapter 5 I will write compatible wrapper APIs for Sesame. These two wrappers implement the same interface so it is easy to switch applications to use either AllegroGraph with my AllegroGraph client wrapper APIs or to use Sesame with my wrapper (with my own text index/search and geolocation implementation).

4.1. Public APIs for the AllegroGraph Wrapper

The following listing shows the public interface for both the AllegroGraph and Sesame wrappers implementations.

```
package com.knowledgebooks.rdf;

import org.openrdf.model.Literal;
import org.openrdf.model.URI;

import java.util.List;

public interface RdfServiceProxy {
  public void deleteRepository(String name)
         throws Exception;
  public void createRepository(String name)
         throws Exception;
  public void addTriple(String subject,
                        String predicate,
                        String object) throws Exception;
  public void addTriple(String subject,
                        URI predicate,
                        String object) throws Exception;
  public void addTriple(String subject,
                        String predicate,
```

```
                        Literal object) throws Exception;
  public void addTriple(String subject,
                        URI predicate,
                        Literal object) throws Exception;
  public List<List<String>> textSearch(String text)
         throws Exception;
  public List<String> textSearch_scala(String text)
         throws Exception;
  public List<List<String>> query(String sparql)
         throws Exception;
  public List<String> query_scala(String sparql)
         throws Exception;
  public void registerFreetextPredicate(String predicate)
         throws Exception;
  public void initializeGeoLocation(
         Double strip_width_in_miles) throws Exception;
  public List<List<String>> getLocations(
         Double latitude, Double longitude,
         Double radius_in_km) throws Exception;
  public List<String> getLocations_scala(
         Double latitude, Double longitude,
         Double radius_in_km) throws Exception;
  public Literal latLonToLiteral(double lat, double lon);
  public void close();
}
```

The AllegroGraph Java APIs use the Sesame classes in the package *org.openrdf.model*. The method *addTriple* is overloaded to accept several combinations of String, URI, and Literal arguments.

4.2. Implementing the Wrapper

You can find the implementation of the AllegroGraph wrapper class *AllegroGraphServerProxy* in the package *com.knowledgebooks.rdf*. Most of the implementation details will look familiar from the code examples in Chapter 2. This class implements the *RdfServiceProxy* interface that is listed in the last section. I am not going to list the entrie implementation here. I refer you to the source code if you want to read through the entire implementation[1].

We will look at a snippet of the code for performing a SPARQL query. You use the classes *TupleQuery* and *TupleQueryResult* to prepare and execute a query:

[1] You will find Franz's online documentation useful.

```
public List<List<String>> query(String sparql)
       throws Exception {
  List<List<String>> ret = new ArrayList<List<String>>();
  TupleQuery tupleQuery =
    conn.prepareTupleQuery(QueryLanguage.SPARQL,sparql);
  TupleQueryResult result = tupleQuery.evaluate();
```

Since a SPARQL query can use a variable number of variables, the first thing that you need to do is to get a list of variables defined for the result set. You can then iterate though the result set and build a return list of lists of strings containing the bound values bound to these variables:

```
  try {
    List<String> bindingNames =
       result.getBindingNames();
    while (result.hasNext()) {
      BindingSet bindingSet = result.next();

      int size2 = bindingSet.size();
      ArrayList<String> vals =
        new ArrayList<String>(size2);
      for (int i = 0; i < size2; i++)
        vals.add(bindingSet.
           getValue(bindingNames.get(i)).stringValue());
      ret.add(vals);
    }
  } finally {
    result.close();
  }
  return ret;
}
```

The first list of strings contains the variable names and the rest of the list of strings in the method return value contain the values.[2]

4.3. Example Java Application

For Java clients, use either of the two following statements to access either a remote AllegroGraph server or an embedded Sesame instance (with my search and geolocation enhancements):

[2]The geospatial APIs use different AllegroGraph class *RepositoryResult*; see the *getLocations* method for an example.

```
RdfServiceProxy proxy = new AllegroGraphServerProxy();
RdfServiceProxy proxy = new SesameEmbeddedProxy();
```

The following test program is configured to use a remote AllegroGraph server:

```
import com.knowledgebooks.rdf.AllegroGraphServerProxy;
import com.knowledgebooks.rdf.RdfServiceProxy;
import com.knowledgebooks.rdf.Triple;

import java.util.List;

public class TestRemoteAllegroGraphServer {
  public static void main(String[] args)
                throws Exception {
    RdfServiceProxy proxy =
        new AllegroGraphServerProxy();
    proxy.deleteRepository("testrepo1");
    proxy.createRepository("testrepo1");
```

I first deleted the repository ”testrepo1” and then created it in this example. In a real application, you would set up a repository one time and reuse it. I want to use both free text indexing and search and geolocation so I make the API calls to activate indexing for all triples containing the predicate *http://example.org/ontology/name* and initialize the repository for handling geolocation:

```
    // register this predicate before adding
    // triples using this predicate:
    proxy.registerFreetextPredicate(
            "http://example.org/ontology/name");
    // set geolocation resolution strip width to 10 KM:
    proxy.initializeGeoLocation(10d);
```

The rest of this example code snippet adds test triples to the repository and performs a few example queries:

```
    proxy.addTriple("http://example.org/people/alice",
                    Triple.RDF_TYPE,
                    "http://example.org/people/alice");
    proxy.addTriple("http://example.org/people/alice",
                     "http://example.org/ontology/name",
                     "Alice");
    proxy.addTriple("http://example.org/people/alice",
```

```
                    Triple.RDF_LOCATION,
            proxy.latLonToLiteral(+37.86385,-122.3430));
    proxy.addTriple("http://example.org/people/bob",
                    Triple.RDF_LOCATION,
            proxy.latLonToLiteral(+37.88385,-122.3130));

    // SPARQL query to get all triples in data store:
    List<List<String>> results =
        proxy.query("SELECT ?s ?p ?o WHERE {?s ?p ?o .}");
    for (List<String> result : results) {
      System.out.println(
        "All triples result: " + result);
    }

    // example test search:
    results = proxy.textSearch("Alice");
    for (List<String> result : results) {
      System.out.println(
        "Wild card text search result: " + result);
    }

    // example geolocatio search:
    results = proxy.getLocations(
                +37.88385d,-122.3130d, 500d);
    for (List<String> result : results) {
      System.out.println(
        "Geolocation result: " + result);
    }
  }
}
```

My wrapper API for performing text search takes a string argument containing one or more search terms and returns all matching triples. The geolocation search method *getLocations* returns a list of triples within a specified radius around a point defined by a latitude/longitude value.

The file *test/TestRemoteAllegroGraphServer.java* contains this code snippet.

4.4. Supporting Scala Client Applications

While it is fairly easy calling Java directly from Scala, I wanted a more ”Scala like” API so I wrote a thin wrapper for the Java wrapper. The following Scala wrapper also works fine with the Sesame library developed in the next chapter. The following listing

has been heavily edited to make long lines fit on the page; you may find the source file easier to read.

```
package rdf_scala

import com.knowledgebooks.rdf
import org.openrdf.model.URI
import rdf.{RdfServiceProxy, SesameEmbeddedProxy,
            Triple, AllegroGraphServerProxy}

class RdfWrapper {

  val proxy : RdfServiceProxy = new AllegroGraphServerProxy()
  //val proxy : RdfServiceProxy = new SesameEmbeddedProxy()

  def listToTriple(sl : List[Object]) : List[Triple] = {
    var  arr = List[Triple]()
    var (skip, rest) = sl.splitAt(4)
    while (rest.length > 2) {
      val (x, y) = rest.splitAt(3)
      arr += new Triple(x(0), x(1), x(2))
      rest = y
    }
    arr
  }
  def listToMulLists(sl : List[Object]) :
       List[List[Object]] = {
    var  arr = List[List[Object]]()
    var (num, rest) = sl.splitAt(1)
    val size = Integer.parseInt("" + num(0))
    var (variables, rest2) = rest.splitAt(size)
    while (rest2.length >= size) {
      val (x, y) = rest2.splitAt(size)
      arr += x
      rest2 = y
    }
    arr
  }
  def query(q : String) : List[List[Object]] = {
    listToMulLists(proxy.query_scala(q).toArray.toList)
  }
  def get_locations(lat : Double, lon : Double,
      radius_in_km : Double) : List[Triple] = {
     listToTriple(
```

```
        proxy.getLocations_scala(lat, lon, radius_in_km).
          toArray.toList.toArray.toList)
  }
  def delete_repository(name : String) =
   { proxy.deleteRepository(name) }
  def create_repository(name : String) =
   { proxy.createRepository(name) }
  def register_free_text_predicate(
    predicate_name : String) =
     { proxy.registerFreetextPredicate(predicate_name) }
  def initialize_geolocation(strip_width : Double) =
  { proxy.initializeGeoLocation(strip_width) }
  def add_triple(subject : String, predicate : String,
                 obj : String) =
    { proxy.addTriple(subject, predicate, obj) }
  def add_triple(subject : String, predicate : String,
                  obj : org.openrdf.model.Literal) =
    { proxy.addTriple(subject, predicate, obj) }
  def add_triple(subject : String, predicate : URI,
                  obj : org.openrdf.model.Literal) =
    { proxy.addTriple(subject, predicate, obj) }
  def add_triple(subject : String, predicate : URI,
                  obj : String) =
    { proxy.addTriple(subject, predicate, obj) }
  def lat_lon_to_literal(lat : Double, lon : Double) = {
    proxy.latLonToLiteral(lat, lon)
  }
  def text_search(query: String) = {
    listToTriple(
       proxy.textSearch_scala(query).toArray.toList)
  }
}
```

Here is an example Scala client application that uses the wrapper:

```
import rdf_scala.RdfWrapper

object TestScala {
  def main(args: Array[String]) {
    var ag = new RdfWrapper
    ag.delete_repository("scalatest2")
    ag.create_repository("scalatest2")
    ag.register_free_text_predicate(
      "http://example.org/ontology/name")
```

```
    ag.initialize_geolocation(3)
    ag.add_triple("http://example.org/people/alice",
         com.knowledgebooks.rdf.Triple.RDF_TYPE,
          "http://example.org/people/alice")
    ag.add_triple("http://example.org/people/alice",
         "http://example.org/ontology/name", "Alice")

    ag.add_triple("http://example.org/people/alice",
       com.knowledgebooks.rdf.Triple.RDF_LOCATION,
       ag.lat_lon_to_literal(+37.783333, -122.433334))

    var results =
      ag.query("SELECT ?s ?p ?o  WHERE {?s ?p ?o .}")
    for (result <- results)
      println("All tuple result using class: " + result)
    var results2 = ag.text_search("Alice");
    for (result <- results2)
      println("Partial text match: " + result)
    var results3 =
      ag.get_locations(+37.513333, -122.313334, 500)
    for (result <- results3)
      println("Geolocation search: " + result)
  }
}
```

This example is similar to the Java client example in Section 4.3. I find Scala to be more convenient than Java for writing client code because it is a more concise language. I offer support for another concise programming language, Clojure, in the next section.

4.5. Supporting Clojure Client Applications

While it is fairly easy calling Java directly from Clojure, I wanted a more "Clojure like" API so I wrote a thin wrapper for the Java wrapper. The following Clojure wrapper also works fine with the Sesame library developed in the next chapter.

The source file *src/rdf_clojure.clj* contains this wrapper:

```
(ns rdf_clojure)

(import '(com.knowledgebooks.rdf Triple)
        '(com.knowledgebooks.rdf AllegroGraphServerProxy)
```

```
        '(com.knowledgebooks.rdf SesameEmbeddedProxy))

(defn rdf-proxy [] (AllegroGraphServerProxy.))
;;(defn rdf-proxy [] (SesameEmbeddedProxy.))

(defn delete-repository [ag-proxy name]
  (.deleteRepository ag-proxy name))
(defn create-repository [ag-proxy name]
  (.createRepository ag-proxy name))
(defn register-freetext-predicate [ag-proxy predicate-name]
   (.registerFreetextPredicate ag-proxy predicate-name))
(defn initialize-geoLocation [ag-proxy radius]
   (.initializeGeoLocation ag-proxy (float radius)))
(defn add-triple [ag-proxy s p o]
  (.addTriple ag-proxy s p o))
(defn query [ag-proxy sparql]
  (for [triple (seq (.query ag-proxy sparql))]
    [(.get triple 0) (.get triple 1) (.get triple 2)]))
(defn text-search [ag-proxy query-string]
   (.textSearch ag-proxy query-string))
(defn get-locations [ag-proxy lat lon radius]
  (.getLocations ag-proxy lat lon radius))
```

Here is a short Clojure example program (*test/test-rdf-clojure.clj*):

```
(use 'rdf_clojure)
(import '(com.knowledgebooks.rdf Triple))

(def agp (rdf-proxy))
(println agp)
(delete-repository agp "testrepo1")
(create-repository agp "testrepo1")
(register-freetext-predicate agp
   "http://example.org/ontology/name")
(initialize-geoLocation agp 3)
(add-triple agp
    "http://example.org/people/alice"
    Triple/RDF_TYPE
    "http://example.org/people")
(add-triple agp
    "http://example.org/people/alice"
    "http://example.org/ontology/name"
    "Alice")
(add-triple agp
```

```
    "http://example.org/people/alice"
    Triple/RDF_LOCATION
    (.latLonToLiteral agp +37.783333 -122.433334))

(println "All triples:\n"
         (query agp "select ?s ?p ?o where {?s ?p ?o}"))

(println "\nText match results\n"
         (text-search agp "Ali*"))

(println "\nGeolocation results:\n"
        (get-locations agp +37.113333 -122.113334  500.0))
```

4.6. Supporting JRuby Client Applications

While it is fairly easy calling Java directly from JRuby, I use a thin wrapper for the Java wrapper. The following JRuby wrapper also works fine with the Sesame library developed in the next chapter.

The source file *src/rdf_ruby.rb* contains this wrapper. For development, I run the Java, Clojure, and Scala examples inside the IntelliJ IDE and I have the Java JAR files in the *lib* directory in both my build and execution CLASSPATH. I usually run JRuby code from the command line and the first thing that the JRuby wrapper must do is to load all of the JAR files in the *lib* directory. The JAR file *knowledgebooks.jar* is created by the *Makefile* included in the git project for this book. If you are not going to use JRuby then you do not need to build this JAR file.

```
require 'java'
(Dir.glob("lib/*.jar") +
 Dir.glob("lib/sesame-2.2.4/*.jar")).each do |fname|
  require fname
end
require "knowledgebooks.jar"

class RdfRuby
  def initialize
    puts "\nWARNING: call either RdfRuby.allegrograph \\
     or RdfRuby.sesame to create a new RdfRuby instance.\n"
  end
  def RdfRuby.allegrograph
    @proxy =
       com.knowledgebooks.rdf.AllegroGraphServerProxy.new
  end
```

```
  def RdfRuby.sesame
    @proxy =
       com.knowledgebooks.rdf.SesameEmbeddedProxy.new
  end
  def delete_repository name
    @proxy.deleteRepository(name)
  end
  def create_repository name
    @proxy.createRepository(name)
  end
  def register_freetext_predicate predicate_name
    @proxy.registerFreetextPredicate(predicate_name)
  end
  def initialize_geo_location resolution_in_miles
    @proxy.initializeGeoLocation(resolution_in_miles)
  end
  def add_triple subject, predicate, object
    @proxy.addTriple(subject, predicate, object)
  end
  def lat_lon_to_literal lat, lon
    @proxy.latLonToLiteral(lat, lon)
  end
  def query sparql
    @proxy.query(sparql)
  end
  def text_search text
    @proxy.textSearch(text)
  end
  def get_ocations lat, lon, radius
    @proxy.getLocations(lat, lon, radius)
  end
end
```

Here is a short JRuby example program (file *test/test_ruby_rdf.rb*):

```
require 'src/rdf_ruby'
require 'pp'

#rdf = RdfRuby.sesame
rdf = RdfRuby.allegrograph
rdf.delete_repository("rtest_repo")
rdf.create_repository("rtest_repo")
rdf.register_freetext_predicate(
  "http://example.org/ontology/name")
```

```
rdf.initialize_geo_location(5.0)
rdf.add_triple("<http://kbsportal.com/oak_creek_flooding>",
   "<http://knowledgebooks.com/ontology/#storyType>",
   "<http://knowledgebooks.com/ontology/#disaster>")
rdf.add_triple("http://example.org/people/alice",
   "http://example.org/ontology/name", "Alice")
rdf.add_triple("http://example.org/people/alice",
   com.knowledgebooks.rdf.Triple.RDF_LOCATION,
   rdf.latLonToLiteral(+37.783333,-122.433334))
results = rdf.query("SELECT ?subject ?object WHERE \\
   { ?subject  \\
      <http://knowledgebooks.com/ontology/#storyType> \\
      ?object . }")
pp results
results = rdf.text_search("alice")
pp results
results = rdf.get_locations(+37.113333,-122.113334, 500)
pp results
```

Like Scala and Clojure, JRuby is a very concise language.[3] Here is the output from this example, showing some debug output from the geolocation query:

```
[[http://kbsportal.com/oak_creek_flooding,
 http://knowledgebooks.com/ontology/#disaster]]
[[<http://example.org/people/alice>,
 <http://example.org/ontology/name>, "Alice"]]
getLocations: geohash for input lat/lon = 9q95jhrbc4dw
Distance: 77.802345
[[<http://example.org/people/alice>,
 <http://knowledgebooks.com/rdf/location>,
 "+37.783333-122.433334" \\
    @http://knowledgebooks.com/rdf/latlon]]
```

4.7. Wrapup

You can also use the Allegrograph client APIs to access remote SPARQL endpoints but I do not cover them here because I write a portable SPARQL client library in Section 14.4 that we will use to access remote SPARQL endpoint web services like DBpedia.

[3] I do about half of my development using Ruby and split the other half between Lisp, Java, Scala, and Clojure. Ruby is my preferred language when fast runtime performance is not a requirement.

My coverage of the AllegroGraph APIs in Chapter 2 and the implementation of my wrapper in this chapter is adequate for both my current use for the AllegroGraph server and the examples in this book. If after working through this book you end up using the commercial version AllegroGraph for very large RDF data stores you will probably be better off using Franz's APIs since they expose all of the functionality of AllegroGraph web services. That said, the functionality that I expose in my wrapper (for both AllegroGraph and Sesame) serves to support the examples in this book.

5. An API Wrapper for Sesame

I created a wrapper for the Franz AllegroGraph APIs in the last chapter in Section 4.1. I will now implement another wrapper in this chapter for Sesame with my own text index/search and geolocation implementation.

The code to implement geolocation and text index/search functionality is in the source file SesameEmbeddedProxy.java. We will look at a few code snippets for non-obvious implementation details and then I will leave it to you to browse the source file.

5.1. Using the Embedded Derby Database

I use the embedded Derby database library for keeping track of RDF predicates that we are tagging for indexing the objects in indexed triples. Here is the database initialization code[1] for this:

```
String db_url =
  "jdbc:derby:tempdata/" + name +
  ".sesame_aux_db;create=true";
try { database_connection =
         DriverManager.getConnection(db_url);
} catch (SQLException sqle) {
  sqle.printStackTrace();
}
// create table free_text_predicates
// if it does not already exist:
try {
  java.sql.Statement stmt =
     database_connection.createStatement();
  int status = stmt.executeUpdate(
    "create table free_text_predicates \\
                (predicate varchar(120))");
  System.out.println(
```

[1] Like many of the listings in this book, I had to break up long lines to fit the page width. You might want to read through the code in the file SesameEmbeddedProxy.java using your favorite programming editor or IDE.

```
      "status for creating table \
        free_text_predicates = " + status);
  } catch (SQLException ex) {
    System.out.println(
      "Error trying to create table \\
        free_text_predicates: " + ex);
  }
```

Here, the variable *name* is the repository name. The following code snippet is the implementation of the wrapper method for registering a predicate so that triples using this predicate can be searched:

```
// call this method before adding triples
public void registerFreetextPredicate(String predicate) {
  try {
    predicate = fix_uri_format(predicate);
    java.sql.Statement stmt =
       database_connection.createStatement();
    ResultSet rs =
      stmt.executeQuery(
        "select * from free_text_predicates \\
             where predicate = '"+predicate+"'");
    if (rs.next() == false) {
      stmt.executeUpdate(
        "insert into free_text_predicates values \\
                          ('" + predicate+"')");
    }
  } catch (SQLException ex) {
    System.out.println("Error trying to write to \\
    table free_text_predicates: " + ex+"\n"+predicate);
  }
}
```

The private method *fix_uri_format* makes sure the URIs are wrapped in $<>$ characters and handles geolocation URIs. The following code is the implementation of the wrapper function for initializing the geolocation database table:

```
public void initializeGeoLocation(Double strip_width) {
  Triple.RDF_LOCATION =
    valueFactory.createURI(
         "http://knowledgebooks.com/rdf/location");
  System.out.println(
         "Initializing geolocation database...");
```

```
  this.strip_width = strip_width.floatValue();
  // create table geoloc if it does not already exist:
  try {
    java.sql.Statement stmt =
       database_connection.createStatement();
    int status =
      stmt.executeUpdate(
       "create table geoloc (geohash char(15), \\
           subject varchar(120), \\
           predicate varchar(120), \\
           lat_lon_object varchar(120), \\
           lat float, lon float)");
    System.out.println("status for creating \\
       table geoloc = " + status);
  } catch (SQLException ex) {
    System.out.println("Warning trying to \\
           create table geoloc (OK, table \\
           is already created): " + ex);
  }
}
```

The geolocation resolution (the argument strip_width) is not used in the Sesame wrapper and exists for compatibility with AllegroGraph.

5.2. Using the Embedded Lucene Library

The class *com.knowledgebooks.rdf.implementation.LuceneRdfManager* wraps the use of the embedded Lucene[2] text index and search library. Lucene is a state of the art indexing and search system that is often used by itself in an embedded mode or as part of larger projects like Solr[3] or Nutch[4]. Here is the implementation of this helper class:

```
public class LuceneRdfManager {
  public LuceneRdfManager(String data_store_file_root)
                                          throws Exception {
    this.data_store_file_root = data_store_file_root;
  }
```

[2]Lucene is a very useful library but any detailed coverage is outside the scope of this book. There is a short introduction on Apache's web site: http://lucene.apache.org/java/3_0_1/gettingstarted.html.

[3]Solr runs as a web service and adds sharding, spelling correction, and many other nice features to Lucene. I usually use Solr to implement search in Rails projects.

[4]I consider Nutch to be a "Google in a box" turnkey search system that scales to large numbers of servers. The Hadoop distributed map reduce system started as part of the Nutch project.

```
  public void addTripleToIndex(String subject,
                               String predicate,
                               String object)
                              throws IOException {
    File index_dir = new File(data_store_file_root +
                              "/lucene_index");
    writer =
     new IndexWriter(FSDirectory.open(index_dir),
                     new StandardAnalyzer(
                         Version.LUCENE_CURRENT),
                     !index_dir.exists(),
                     IndexWriter.MaxFieldLength.LIMITED);
    Document doc = new Document();
    doc.add(new Field("subject", subject,
                      Field.Store.YES,
                      Field.Index.NO));
    doc.add(new Field("predicate", predicate,
                      Field.Store.YES,
                      Field.Index.NO));
    doc.add(new Field("object", object,
                      Field.Store.YES,
                      Field.Index.ANALYZED));
    writer.addDocument(doc);
    writer.optimize();
    writer.close();
  }
  public List<List<String>>
         searchIndex(String search_query)
            throws ParseException, IOException {
    File index_dir =
      new File(data_store_file_root +
               "/lucene_index");
    reader = IndexReader.open(
        FSDirectory.open(index_dir), true);
    List<List<String>> ret =
        new ArrayList<List<String>>();
    Searcher searcher = new IndexSearcher(reader);

    Analyzer analyzer =
       new StandardAnalyzer(Version.LUCENE_CURRENT);
    QueryParser parser =
      new QueryParser(Version.LUCENE_CURRENT,
                      "object", analyzer);
    Query query = parser.parse(search_query);
```

```
    TopScoreDocCollector collector =
      TopScoreDocCollector.create(10, false);
    searcher.search(query, collector);
    ScoreDoc[] hits = collector.topDocs().scoreDocs;

    for (int i = 0; i < hits.length; i += 1) {
      Document doc = searcher.doc(hits[i].doc);
      List<String> as2 = new ArrayList<String>(20);
      as2.add(doc.get("subject"));
      as2.add(doc.get("predicate"));
      as2.add(doc.get("object"));
      ret.add(as2);
    }
    reader.close();
    return ret;
  }

  private String data_store_file_root;
  private IndexWriter writer;
  private IndexReader reader;
}
```

This code to use embedded Lucene is fairly straightforward, the only potentially tricky part being checking to see if a disk-based Lucene index directory already exists. It is important to call the constructor for class *IndexWriter* with the correct third argument value of false if the index already exists so we don't overwrite an existing index.

There is some inefficiency in both methods *addTripleToIndex* and *searchIndex* because I open and close the index as needed. For production work you would want to maintain an open index and serialize calls that use the index. The code is pedantic[5] as written but simple to understand.

5.3. Wrapup for Sesame Wrapper

I have tried to make the implementation of the Sesame wrapper functionally equivalent to the AllegroGraph wrapper. This goal is largely met although there are differences in the inferencing support between AllegroGraph and Sesame: both support RDFS inferencing (see Chapter 7) and AllegroGraph additionally supports some OWL (Web Ontology Language) extensions.

[5] My purpose is to teach you how to use Semantic Web and Linked Data technologies to build practical applications. I am trying to make the code examples as simple as possible and still provide you with tools that you can both experiment with and build applications with. I always write code as simple as possible and worry later about efficiency if it does not run fast enough.

The Scala, Clojure, and JRuby client examples from the last chapter also work as-is using the Sesame wrapper developed in this chapter.

You can also use the Sesame client APIs to access remote SPARQL endpoints but I do not cover them here because I write a portable SPARQL client library in Section 14.4 that we will use to access remote SPARQL endpoint web services in later examples.

Part III.

Semantic Web Technologies

6. RDF

The Semantic Web is intended to provide a massive linked data set for use by software systems just as the World Wide Web provides a massive collection of linked web pages for human reading and browsing. The Semantic Web is like the World Wide Web in that anyone can generate any content that they want. This freedom to publish anything works for the web because we use our ability to understand natural language to interpret what we read – and often to dismiss material that based upon our own knowledge we consider to be incorrect.

The core concept for the Semantic Web is data integration and use from different sources. As we will soon see, the tools for implementing the Semantic Web are designed for encoding data and sharing data from many different sources.

The Resource Description Framework (RDF) is used to encode information and the RDF Schema (RDFS) language defines properties and classes and also facilitates using data with different RDF encodings without the need to convert data to use different schemas. For example, no need to change a property name in one data set to match the semantically identical property name used in another data set. Instead, you can add an RDF statement that states that the two properties have the same meaning.

I do not consider RDF data stores to be a replacement for relational databases but rather something that you will use with databases in your applications. RDF and relational databases solve different problems. RDF is appropriate for sparse data representations that do not require inflexible schemas. You are free to define and use new properties and use these properties to make statements on existing resources. RDF offers more flexibility: defining properties used with classes is similar to defining the columns in a relational database table. You do not need to define properties for every instance of a class. This is analogous to a database table that can be missing columns for rows that do not have values for these columns (a sparse data representation). Furthermore, you can make ad hoc RDF statements about any resource without the need to update global schemas. We will use the SPARQL query language to access information in RDF data stores. SPARQL queries can contain optional matching clauses that work well with sparse data representations.

RDF data was originally encoded as XML and intended for automated processing. In this chapter we will use two simple to read formats called N-Triples and N3[1]. There

[1] N3 is a far better format to work with if you want to be able to read RDF data files and understand their contents. Currently AllegroGraph does not support N3 but Sesame does. I will usually use the N3

are many tools available that can be used to convert between all RDF formats so we might as well use formats that are easier to read and understand. RDF data consists of a set of triple values:

- subject - this is a URI
- predicate - this is a URI
- object - this is either a URI or a literal value

A statement in RDF is a triple composed of a subject, predicate, and object. A single resource containing a set of RDF triples can be referred to as an RDF graph. These resources might be a downloadable RDF file that you can load into AllegroGraph or Sesame, a web service that returns RDF data, or a SPARQL endpoint that is a web service that accepts SPARQL queries and returns information from an RDF data store.

While we tend to think in terms of objects and classes when using object oriented programming languages, we need to readjust our thinking when dealing with knowledge assets on the web. Instead of thinking about "objects" we deal with "resources" that are specified by URIs. In this way resources can be uniquely defined. We will soon see how we can associate different namespaces with URI prefixes – this will make it easier to deal with different resources with the same name that can be found in different sources of information.

While subjects will almost always be represented as URIs of resources, the object part of triples can be either URIs of resources or literal values. For literal values, the XML schema notation for specifying either a standard type like integer or string, or a custom type that is application domain specific.

You have probably read articles and other books on the Semantic Web, and if so, you are probably used to seeing RDF expressed in its XML serialization format: you will not see XML serialization in this book. Much of my own confusion when I was starting to use Semantic Web technologies ten years ago was directly caused by trying to think about RDF in XML form. RDF data is graph data and serializing RDF as XML is confusing and a waste of time when either the N-Triple format or even better, the N3 format are so much easier to read and understand.

Some of my work with Semantic Web technologies deals with processing news stories, extracting semantic information from the text, and storing it in RDF. I will use this application domain for the examples in this chapter. I deal with triples like:

- subject: a URI, for example the URL of a news article
- predicate: a relation like "a person's name" that is represented as a URI like

format when discussing ideas but use the N-Triple format as input for example programs and for output when saving RDF data to files.

<http://knowledgebooks.com/rdf/person/name>[2]

- object: a literal value like ”Bill Clinton” or a URI

We will always use URIs[3] as values for subjects and predicates, and use URIs or string literals as values for objects. In any case URIs are usually preferred to string literals because they are unique; for example, consider the two possible values for a triple object:

- ”Bill Clinton” - as a string literal, the value may not refer to President Bill Clinton.
- <http://knowledgebooks.com/rdf/person#BillClinton> - as a URI, we can later make this URI a subject in a triple and use a relation to specify that this particular person had the job of President of the United States.

We will see an example of this preferred use but first we need to learn the N-Triple and N3 RDF formats.

6.1. RDF Examples in N-Triple and N3 Formats

In the Introduction I proposed the idea that RDF was more flexible than Object Modeling[4] in programming languages, relational databases, and XML with schemas[5]. If we can tag new attributes on the fly to existing data, how do we prevent what I might call “data chaos” as we modify existing data sources? It turns out that the solution to this problem is also the solution for encoding real semantics (or meaning) with data: we usually use unique URIs for RDF subjects, predicates, and objects, and usually with a preference for not using string literals. I will try to make this idea more clear with some examples.

Any part of a triple (subject, predicate, or object) is either a URI or a string literal. URIs encode namespaces. For example, the containsPerson property is used as the value of the predicate in this triple; the last example could properly be written as:

[2] URIs, like URLs, start with a protocol like HTTP that is followed by an internet domain.

[3] Uniform Resource Identifiers (URIs) are special in the sense that they (are supposed to) represent unique things or ideas. As we will see in Chapter 9, URIs can also be ”dereferenceable” in that we can treat them as URLs on the web and ”follow” them using HTTP to get additional information about a URI.

[4] We will model classes (or types) using RDFS and OWL but the difference is that an object in an OO language is explicitly declared to be a member of a class while a subject URI is considered to be in a class depending only on what properties it has. If we add a property and value to a subject URI then we may immediately change its RDFS or OWL class membership.

[5] I think that there is some similarity between modeling with RDF and document oriented data stores like MongoDB or CouchDB where any document in the system can have any attribute added at any time. This is very similar to being able to add additional RDF statements that either add information about a subject URI or add another property and value that somehow narrows the ”meaning” of a subject URI.

```
http://knowledgebooks.com/ontology/#containsPerson
```

The first part of this URI is considered to be the namespace[6] for (what we will use as a predicate) "containsPerson." Once we associate an abbreviation like **kb** for **http://knowledgebooks.com/ontology/** then we can just use the QName ("quick name") with the namespace abbreviation; for example:

```
kb:containsPerson
```

Being able to define abbreviation prefixes for namespaces makes RDF and RDFS files shorter and easier to read.

When different RDF triples use this same predicate, this is some assurance to us that all users of this predicate subscribe to the same meaning. Furthermore, we will see in Section 7.1 that we can use RDFS to state equivalency between this predicate (in the namespace http://knowledgebooks.com/ontology/) with predicates represented by different URIs used in other data sources. In an "artificial intelligence" sense, software that we write does not understand a predicate like "containsPerson" in the way that a human reader can by combining understood common meanings for the words "contains" and "person" but for many interesting and useful types of applications that is fine as long as the predicate is used consistently.

Because there are many sources of information about different resources the ability to define different namespaces and associate them with unique URI prefixes makes it easier to deal with situations.

A statement in N-Triple format consists of three URIs (or string literals – any combination) followed by a period to end the statement. While statements are often written one per line in a source file they can be broken across lines; it is the ending period which marks the end of a statement. The standard file extension for N-Triple format files is *.nt and the standard format for N3 format files is *.n3.

My preference is to use N-Triple format files as output from programs that I write to save data as RDF. I often use either command line tools or the Java Sesame library to convert N-Triple files to N3 if I will be reading them or even hand editing them. You will see why I prefer the N3 format when we look at an example:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
<http://news.com/201234 /> kb:containsCountry "China"  .
```

[6] You have seen me use the domain knowledgebooks.com several times in examples. I have owned this domain and used it for business since 1998 and I use it here for convenience. I could just as well use example.com. That said, the advantage of using my own domain is that I then have the flexibility to make this URI "dereferenceable" by adding an HTML document using this URI as a URL that describes what I mean by "containsPerson." Even better, I could have my web server look at the request header and return RDF data if the requested content type was "text/rdf"

Here we see the use of an abbreviation prefix "kb:" for the namespace for my company KnowledgeBooks.com ontologies. The first term in the RDF statement (the subject) is the URI of a news article. When we want to use a URL as a URI, we enclose it in angle brackets – as in this example. The second term (the predicate) is "containsCountry" in the "kb:" namespace. The last item in the statement (the object) is a string literal "China." I would describe this RDF statement in English as, "The news article at URI http://news.com/201234 mentions the country China."

This was a very simple N3 example which we will expand to show additional features of the N3 notation. As another example, suppose that this news article also mentions the USA. Instead of adding a whole new statement like this:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
<http://news.com/201234 /> kb:containsCountry "China"  .
<http://news.com/201234 /> kb:containsCountry "USA"  .
```

we can combine them using N3 notation. N3 allows us to collapse multiple RDF statements that share the same subject and optionally the same predicate:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
<http://news.com/201234 /> kb:containsCountry "China" ,
                                              "USA" .
```

We can also add in additional predicates that use the same subject:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .

<http://news.com/201234 /> kb:containsCountry "China" ,
                                              "USA" .
    kb:containsOrganization "United Nations" ;
    kb:containsPerson "Ban Ki-moon" , "Gordon Brown" ,
                      "Hu Jintao" , "George W. Bush" ,
                      "Pervez Musharraf" ,
                      "Vladimir Putin" ,
                      "Mahmoud Ahmadinejad" .
```

This single N3 statement represents ten individual RDF triples. Each section defining triples with the same subject and predicate have objects separated by commas and ending with a period. Please note that whatever RDF storage system we use (we will be using AllegroGraph) it makes no difference if we load RDF as XML, N-Triple, of N3 format files: internally subject, predicate, and object triples are stored in the same way and are used in the same way.

I promised you that the data in RDF data stores was easy to extend. As an example, let us assume that we have written software that is able to read online news articles and create RDF data that captures some of the semantics in the articles. If we extend our program to also recognize dates when the articles are published, we can simply reprocess articles and for each article add a triple to our RDF data store using the N-Triple format to set a publication date[7].

```
<http://news.com/2034 /> kb:datePublished "2008-05-11" .
```

Furthermore, if we do not have dates for all news articles that is often acceptable depending on the application.

6.2. The RDF Namespace

You just saw an example of using namespaces when I used my own namespace <http://knowledgebooks.com/ontology#>.

When you define a name space you can assign any "Quick name" (QName, or abbreviation) to the URI that uniquely identifies a namespace if you are using the N3 format.

The RDF namespace <http://www.w3.org/1999/02/22-rdf-syntax-ns#> is usually registered with the QName **rdf:** and I will use this convention. The next few sections show the definitions in the RDF namespace that I use in this book.

6.2.1. rdf:type

The **rdf:type** property is used to specify the type (or class) of a resource. Notice that we do not capitalize "type" because by convention we do not capitalize RDF property names. Here is an example in N3 format (with long lines split to fit the page width):

```
@prefix rdf:
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

@prefix
 kb:
 <http://knowledgebooks.com/rdf/publication#> .

<http://demo_news/12931> rdf:type kb:article .
```

[7] This example is pedantic since we can apply XML Scehma (XSL) data types to literal string values, this could be more accurately specified as "2008-05-11"@http://www.w3.org/2001/XMLSchema#date

Here we are converting the URL of a news web page to a resource and then defining a new triple that specifies the web page resource is or type kb:article (again, using the QName **kb:** for my knowledgebooks.com namespace).

6.2.2. rdf:Property

The **rdf:Property** class is, as you might guess from its name, used to describe and define properties. Notice that "Property" is capitalized because by convention we capitalize RDF class names.

This is a good place to show how we define new properties, using a previous example:

```
@prefix
 kbcontains:
 <http://knowledgebooks.com/rdf/contains#> .

<http://demo_news/12931>
  kbcontains:person
  "Barack Obama" .
```

I might make an additional statement about this URI stating that it is a property:

```
kbcontains:person rdf:type rdf:Property .
```

When we discuss RDF Schema (RDFS) in Chapter 7 we will see how to create sub-types and sub-properties.

6.3. Dereferenceable URIs

We have been using URIs as unique identifiers representing either physical objects (e.g., the moon), locations (e.g., London England), ideas or concepts (e.g., Christianity), etc. Additionally, a URI is dereferenceable if we can follow the URI with a web browser or software agent to fetch information from the URI. As an example, we often use the URI

```
http://xmlns.com/foaf/0.1/Person
```

to represent the concept of a person. This URI is dereferenceable because if we use a tool like wget or curl to fetch the content from this URI then we get an HTML document for the FOAF Vocabulary Specification. Dereferenceable content could also be a RDFS or OWL document describing the URI, a text document, etc.

6.4. RDF Wrap Up

If you read the World Wide Web Consortium's RDF Primer (highly recommended) at http://www.w3.org/TR/REC-rdf-syntax/ you will see many other classes and properties defined that, in my opinion, are often most useful when dealing with XML serialization of RDF. Using the N-Triple and N3 formats, I find that I usually just use rdf:type and rdf:Property in my own modeling efforts, along with a few identifiers defined in the RDFS namespace that we will look at in the next chapter.

An RDF triple has three parts: a subject, predicate, and object.[8] By itself, RDF is good for storing and accessing data but lacks functionality for modeling classes, defining properties, etc. We will extend RDF with RDF Schema (RDFS) in the next chapter.

[8] AllegroGraph also stores a unique integer triple ID and a graph ID for partitioning RDF data and to support graph operations. While using the triple ID and graph ID can be useful, my own preference is to stick with using just what is in the RDF standard.

7. RDFS

The World Wide Web Consortium RDF Schema (RDFS) definition can be read at http://www.w3.org/TR/rdf-schema/ and I recommend that you use this as a reference because I will discuss only the parts of RDFS that are required for implementing the examples in this book. The RDFS namespace http://www.w3.org/2000/01/rdf-schema# is usually registered with the QName **rdfs:** and I will use this convention[1].

7.1. Extending RDF with RDF Schema

RDFS supports the definition of classes and properties based on set inclusion. In RDFS classes and properties are orthogonal. We will not simply be using properties to define data attributes for classes – this is different than object modeling and object oriented programming languages like Java. RDFS is encoded as RDF – the same syntax.

Because the Semantic Web is intended to be processed automatically by software systems it is encoded as RDF. There is a problem that must be solved in implementing and using the Semantic Web: everyone who publishes Semantic Web data is free to create their own RDF schemas for storing data; for example, there is usually no single standard RDF schema definition for topics like news stories, stock market data, people's names, organizations, etc. Understanding the difficulty of integrating different data sources in different formats helps to understand the design decisions behind the Semantic Web: the designers wanted to make it not only possible but also easy to use data from different sources that might use similar schema to define properties and classes. One common usage pattern is using RDFS to define two properties that both define a person's last name have the same meaning and that we can combine data that use different schema.

We will start with an example that also uses dRDFS an is an extension of the example in the last section. After defining **kb:** and **rdfs:** namespace QNames, we add a few additional RDF statements (that are RDFS):

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
```

[1] The actual namespace abbreviations that you use have no effect as long as you consistently use whatever QName values you set for URIs in the RDF statements that use the abbreviations.

```
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

kb:containsCity rdfs:subPropertyOf kb:containsPlace .
kb:containsCountry rdfs:subPropertyOf kb:containsPlace .
kb:containsState rdfs:subPropertyOf kb:containsPlace .
```

The last three lines that are themselves valid RDF triples declare that:

- The property containsCity is a subproperty of containsPlace.
- The property containsCountry is a subproperty of containsPlace.
- The property containsState is a subproperty of containsPlace.

Why is this useful? For at least two reasons:

- You can query an RDF data store for all triples that use property containsPlace and also match triples with property equal to containsCity, containsCountry, or containsState. There may not even be any triples that explicitly use the property containsPlace.
- Consider a hypothetical case where you are using two different RDF data stores that use different properties for naming cities: "cityName" and "city." You can define "cityName" to be a subproperty of "city" and then write all queries against the single property name "city." This removes the necessity to convert data from different sources to use the same Schema.

In addition to providing a vocabulary for describing properties and class membership by properties, RDFS is also used for logical inference to infer new triples, combine data from different RDF data sources, and to allow effective querying of RDF data stores. We will see examples of more RDFS features in Chapter 8 when we perform SPARQL queries.

7.2. Modeling with RDFS

While RDFS is not as expressive of a modeling language as the RDFS++[2] or OWL, the combination of RDF and RDFS is usually adequate for many semantic web applications. Reasoning with and using more expressive modeling languages will require increasingly more processing time. Combined with the simplicity of RDF and RDFS it is a good idea to start with less expressive and only "move up the expressivity scale" as needed.

[2]RDFS++ is a Franz extension to RDFS that adds some parts of OWL. I cover RDFS++ in some detail in the Lisp Edition of this book and mention some aspects of RDFS++ in Section 7.3, the Java Edition.

Here is a short example on using RDFS to extend RDF (assume that my namespace **kb:** and the RDFS namespace **rdfs:** are defined):

```
kb:Person rdf:type rdfs:Class .
kb:Person rdfs:comment "represents a human" .
kb:Manager rdf:type kb:Person .
kb:Manager rdfs:domain kb:Person .
kb:Engineer rdf:type kb:Person .
kb:Engineer rdfs:domain kb:Person .
```

Here we see the use of **rdfs:comment** used to add a human readable comment to the new class **kb:Person**. When we define the new classes **kb:Manager** and **kb:Engineer** we make them subclasses of **kb:Person** instead of the top level super class **rdfs:Class**. We will look at examples later in that that demonstrate the utility of models using class hierarchies and hierarchies of properties – for now it is enough to introduce the notation.

The **rdfs:domain** of an **rdf:property** specifies the class of the subject in a triple while **rdfs:range** of an **rdf:property** specifies the class of the object in a triple. Just as strongly typed programming languages like Java help catch errors by performing type analysis, creating (or using existing) good RDFS property and class definitions helps RDFS, RDFS++, and OWL descriptive logic reasoners to catch modeling and data definition errors. These definitions also help reasoning systems infer new triples that are not explicitly defined in a triple data store.

We continue the current example by adding property definitions and then asserting a triple that is valid given the type and property restrictions that we have defined using RDFS:

```
kb:supervisorOf rdfs:domain kb:Manager .
kb:supervisorOf rdfs:range kb:Engineer .

"Mary Johnson" rdf:type kb:Manager .
"John Smith'' rdf:type kb:Engineer .

"Mary Johnson" kb:supervisorOf "John Smith" .
```

If I tried to add a triple with “Mary Johnson” and “John Smith” reversed in the last RFD statement then an RDFS inference/reasoning system could catch the error. This example is not ideal because I am using string literals as the subjects in triples. In general, you probably want to define a specific namespace for concrete resources representing entities like the people in this example.

The property **rdfs:subClassOf** is used to state that all instances of one class are also instances of another class. The property **rdfs:subPropertyOf** is used to state that all

resources related by one property are also related by another; for example, given the following N3 statements that use string literals as resources to make this example shorter:

```
kb:familyMember rdf:type rdf:Property .
kb:ancestorOf rdf:type rdf:Property .
kb:parentOf rdf:type rdf:Property .

kb:ancestorOf rdfs:subPropertyOf kb:familyMember .
kb:parentOf rdfs:subPropertyOf kb:ancestorOf .

"Marry Smith" kb:parentOf "Sam" .
```

then the following is valid:

```
"Marry Smith" kb:ancestorOf "Sam" .
"Marry Smith" kb:familyMember "Sam" .
```

We have just seen that a common use of RDFS is to define additional application or data-source specific properties and classes in order to express relationships between resources and the types of resources. Whenever possible you will want to reuse existing RDFS properties and resources that you find on the web. For instance, in the last example I defined my own subclass **kb:person** instead of using the Friend of a Friend (FOAF) namespace's definition of person. I did this for pedantic reasons: I wanted to show you how to define your own classes and properties.

7.3. AllegroGraph RDFS++ Extensions

The *unofficial* version of RDFS/OWL called RDFS++ is a practical compromise between DL OWL and RDFS inferencing. AllegroGraph supports the following predicates:

- rdf:type – discussed in Chapter 6
- rdf:property – discussed in Chapter 6
- rdfs:subClassOf – discussed in Chapter 7
- rdfs:range – discussed in Chapter 7
- rdfs:domain – discussed in Chapter 7
- rdfs:subPropertyOf – discussed in Chapter 7

- owl:sameAs
- owl:inverseOf
- owl:TransitiveProperty

We will now discuss **owl:sameAs**, **owl:inverseOf**, and **owl:TransitiveProperty** to complete the discussion of frequently used RDFS predicates seen earlier in this Chapter.

7.3.1. owl:sameAs

If the same entity is represented by two distinct URIs **owl:sameAs** can be used to assert that the URIs refer to the same entity. For example, two different knowledge sources might might define different URIs in their own namespaces for President Barack Obama. Rather than translate data from one knowledge source to another it is simpler to equate the two unique URIs. For example:

```
kb:BillClinton rdf:type kb:Person .
kb:BillClinton owl:sameAs mynews:WilliamClinton
```

Then the following can be verified using an RDFS++ or OWL DL capable reasoner:

```
mynews:WilliamClinton rdf:type kb:Person .
```

7.3.2. owl:inverseOf

We can use **owl:inverseOf** to declare that one property is the inverse of another.

```
:parentOf owl:inverseOf :childOf .
"John Smith" :parentOf "Nellie Smith" .
```

There is something new in this example: I am using a "default namespace" for **:parentOf** and **:childOf**. A default namespace is assumed to be application specific and that no external software agents will need access to resources defined in the default namespace.

Given the two previous RDF statements we can infer that the following is also true:

```
"Nellie Smith" :childOf "John Smith" .
```

7.3.3. owl:TransitiveProperty

As its name implies **owl:TransitiveProperty** is used to declare that a property is transitive as the following example shows:

```
kb:ancestorOf a rdf:Property .
"John Smith"  kb:ancestorOf "Nellie Smith" .
"Nellie Smith" kb:ancestorOf "Billie Smith" .
```

There is something new in this example: in N3 you can use **a** as shorthand for **rdf:type**. Given the last three RDF statements we can infer that:

```
"John Smith" : kb:ancestorOf "Billie Smith" .
```

7.4. RDFS Wrapup

I find that RDFS provides a good compromise: it is simpler to use than the Web Ontology Language (OWL) and is expressive enough for many linked data applications. As we have seen, AllegroGraph supports RDFS++ which is RDFS with a few OWL extensions:

1. rdf:type
2. rdfs:subClassOf
3. rdfs:domain
4. rdfs:range
5. rdfs:subPropertyOf
6. owl:sameAs
7. owl:inverseOf
8. owl:TransitiveProperty

Since I only briefly covered these extensions you may want to read the documentation on Franz's web site[3].

Sesame supports RDFS "out of the box" and back end reasoners are available for Sesame that support OWL[4]. Sesame is likely to have OWL reasoning built in to the

[3] http://www.franz.com/agraph/support/learning/Overview-of-RDFS++.html

[4] You can download SwiftOWLIM or BigOWLIM at http://www.ontotext.com/owlim/ and use either as a SAIL backend repository to get OWL reasoning capability.

standard distribution in the future. My advice is to start building applications with RDF and RDFS with a view to using OWL as the need arises. If you are using AllegroGraph for your application development then certainly use the RDFS++ extensions if RDFS is too limited for your applications.

We have been using SPARQL in examples and in the next chapter we will look at SPARQL in some detail.

8. The SPARQL Query Language

SPARQL is a query language used to query RDF data stores. While SPARQL may initially look like SQL you will see that there are important differences because the data is graph-based so queries match graph patterns instead SQL's relational matching operations. So the syntax is similar but SPARQL queries graph data and SQL queries relational data in tables.

We have already been using SPARQL queries in examples in this book. I will give you more introductory material in this chapter before using SPARQL in larger example programs later in this book.

8.1. Example RDF Data in N3 Format

We will use the N3 format RDF file data/news.n3 for examples in this chapter. We use the N3 format because it is easier to read and understand. There is an equivalent N-Triple format file data/news.nt because AllegroGraph does not currently support loading N3 files. I created these files automatically by spidering Reuters news stories on the news.yahoo.com web site and automatically extracting named entities from the text of the articles. I used the Java Sesame library to convert the generated N-Triple file to N3 format. We will see similar techniques for extracting named entities from text in Chapter 11 when I develop utilities for using the Reuters Open Calais web services. We will also use my Natural Language Processing (NLP) library in Chapter 12 to do the same thing. In this chapter we use these sample RDF files that I have created using Open Calais and news articles that I found on the web.

You have already seen snippets of this file in Section 7.1 and I list the entire file here for reference ,edited to fit line width. You may find the file news.n3 easier to read if you are at your computer and open the file in a text editor so you will not be limited to what fits on a book page):

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

kb:containsCity rdfs:subPropertyOf kb:containsPlace .
```

```
kb:containsCountry rdfs:subPropertyOf kb:containsPlace .

kb:containsState rdfs:subPropertyOf kb:containsPlace .

<http://yahoo.com/20080616/usa_flooding_dc_16 />
    kb:containsCity "Burlington" , "Denver" ,
                    "St. Paul" ," Chicago" ,
                    "Quincy" , "CHICAGO" ,
                    "Iowa City" ;
    kb:containsRegion "U.S. Midwest" , "Midwest" ;
    kb:containsCountry "United States" , "Japan" ;
    kb:containsState "Minnesota" , "Illinois" ,
                     "Mississippi" , "Iowa" ;
    kb:containsOrganization "National Guard" ,
                     "U.S. Department of Agriculture" ,
                     "White House" ,
                     "Chicago Board of Trade" ,
                     "Department of Transportation" ;
    kb:containsPerson "Dena Gray-Fisher" ,
                      "Donald Miller" ,
                      "Glenn Hollander" ,
                      "Rich Feltes" ,
                      "George W. Bush" ;
    kb:containsIndustryTerm "food inflation" , "food" ,
                            "finance ministers" ,
                            "oil" .

<http://yahoo.com/78325/ts_nm/usa_politics_dc_2 />
    kb:containsCity "Washington" , "Baghdad" ,
                    "Arlington" , "Flint" ;
    kb:containsCountry "United States" ,
                       "Afghanistan" ,
                       "Iraq" ;
    kb:containsState "Illinois" , "Virginia" ,
                     "Arizona" , "Michigan" ;
    kb:containsOrganization "White House" ,
                            "Obama administration" ,
                            "Iraqi government" ;
    kb:containsPerson "David Petraeus" ,
                      "John McCain" ,
                      "Hoshiyar Zebari" ,
                      "Barack Obama" ,
                      "George W. Bush" ,
                      "Carly Fiorina" ;
    kb:containsIndustryTerm "oil prices" .
```

```
<http://yahoo.com/10944/ts_nm/worldleaders_dc_1 />
    kb:containsCity "WASHINGTON" ;
    kb:containsCountry "United States" , "Pakistan" ,
                       "Islamic Republic of Iran" ;
    kb:containsState "Maryland" ;
    kb:containsOrganization "University of Maryland" ,
                            "United Nations" ;
    kb:containsPerson "Ban Ki-moon" , "Gordon Brown" ,
                      "Hu Jintao" , "George W. Bush" ,
                      "Pervez Musharraf" ,
                      "Vladimir Putin" ,
                      "Steven Kull" ,
                      "Mahmoud Ahmadinejad" .

<http://yahoo.com/10622/global_economy_dc_4 />
    kb:containsCity "Sao Paulo" , "Kuala Lumpur" ;
    kb:containsRegion "Midwest" ;
    kb:containsCountry "United States" , "Britain" ,
                       "Saudi Arabia" , "Spain" ,
                       "Italy" , India" ,
                       ""France" , "Canada" ,
                       "Russia" , "Germany" , "China" ,
                       "Japan" , "South Korea" ;
    kb:containsOrganization "Federal Reserve Bank" ,
                            "European Union" ,
                            "European Central Bank" ,
                            "European Commission" ;
    kb:containsPerson "Lee Myung-bak" , "Rajat Nag" ,
                      "Luiz Inacio Lula da Silva" ,
                      "Jeffrey Lacker" ;
    kb:containsCompany "Development Bank Managing" ,
                       "Reuters" ,
                       "Richmond Federal Reserve Bank" ;
    kb:containsIndustryTerm "central bank" , "food" ,
                            "energy costs" ,
                            "finance ministers" ,
                            "crude oil prices" ,
                            "oil prices" ,
                            "oil shock" ,
                            "food prices" ,
                            "Finance ministers" ,
                            "Oil prices" , "oil" .
```

8.2. Example SPARQL SELECT Queries

In the following examples, we will look at queries but not the results. You have already seen results of SPARQL queries when we ran the AllegroGraph and Sesame wrapper examples.

We will start with a simple SPARQL query for subjects (news article URLs) and objects (matching countries) with the value for the predicate equal to $containsCountry$:

```
SELECT ?subject ?object
  WHERE {
    ?subject
    http://knowledgebooks.com/ontology#containsCountry>
    ?object .
  }
```

Variables in queries start with a question mark character and can have any names. Since we are using two free variables (*?subject* and *?object*) each matching result will contain two values, one for each of these variables.

We can make this last query easier to read and reduce the chance of misspelling errors by using a namespace prefix:

```
PREFIX kb:  <http://knowledgebooks.com/ontology#>
SELECT ?subject ?object
  WHERE {
      ?subject kb:containsCountry ?object .
  }
```

We could have filtered on any other predicate, for instance *containsPlace*. Here is another example using a match against a string literal to find all articles exactly matching the text "Maryland."

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
SELECT ?subject
WHERE { ?subject kb:containsState "Maryland" . }
```

We can also match partial string literals against regular expressions:

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
SELECT ?subject ?object
```

```
   WHERE {
     ?subject
     kb:containsOrganization
     ?object FILTER regex(?object, "University") .
   }
```

Prior to this last example query we only requested that the query return values for subject and predicate for triples that matched the query. However, we might want to return all triples whose subject (in this case a news article URI) is in one of the matched triples. Note that there are two matching triples, each terminated with a period:

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
SELECT ?subject ?a_predicate ?an_object
   WHERE {
    ?subject
    kb:containsOrganization
    ?object FILTER regex(?object, "University") .

    ?subject ?a_predicate ?an_object .
   }
DISTINCT
ORDER BY ?a_predicate  ?an_object
LIMIT 10
OFFSET 5
```

When WHERE clauses contain more than one triple pattern to match, this is equivalent to a Boolean "and" operation. The DISTINCT clause removes duplicate results. The ORDER BY clause sorts the output in alphabetical order: in this case first by predicate (containsCity, containsCountry, etc.) and then by object. The LIMIT modifier limits the number of results returned and the OFFSET modifier sets the number of matching results to skip.

We are finished with our quick tutorial on using the SELECT query form. There are three other query forms that I will now briefly[1] cover:

- CONSTRUCT – returns a new RDF graph of query results
- ASK – returns Boolean true or false indicating if a query matches any triples
- DESCRIBE – returns a new RDF graph containing matched resources

[1] I almost always use just SELECT queries in applications.

8.3. Example SPARQL CONSTRUCT Queries

A SPARQL CONSTRUCT query acts like a SELECT query in that part of an RDF graph is matched. For CONSTRUCT queries, the matching subgraph is returned.

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
CONSTRUCT {kb:StateOfMaryland kb:isDiscussedIn ?subject }
WHERE { ?subject kb:containsState "Maryland" . }
```

The output graph would only contain one RDF statement because only one of our test news stories mentioned the state of Maryland:

```
kb:StateOfMaryland
  kb:isDiscussedIn
  <http://yahoo.com/10944/ts_nm/worldleaders_dc_1 /> .
```

8.4. Example SPARQL ASK Queries

SPARQL ask queries check the validity of an RDF statement (possibly including variables) and returns ”yes” or ”no” as the query result. In a similar example to the CONSTRUCT query, here I ask if there are any articles that discuss the state of Maryland:

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
ASK { ?subject kb:containsState "Maryland"  }
```

8.5. Example SPARQL DESCRIBE Queries

Currently the SPARQL standard leaves the output from DESCRIBE queries as only partly defined and implementaton specific. A DESCRIBE query is similar to a CONSTRUCT query in that it returns information about resources in queries. The following example should return a graph showing information of all triples using the resource matched by the variable ?subject:

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
DECRIBE ?subject
WHERE { ?subject kb:containsState "Maryland" . }
```

8.6. Wrapup

This chapter ends the background material on Semantic Web Technologies. The remaining chapters in this book will be examples of gathering useful linked data and using it in applications.

9. Linked Data and the World Wide Web

It has been a decade since Tim Berners-Lee started writing about “version 2” of the World Wide Web: the Semantic Web. His new idea was to augment HTML anchor links with typed links using RDF data. As we have seen in detail in the last several chapters, RDF is encoded as data triples with the parts of each triple identified as the **subject**, **predicate**, and **object**. The **predicate** identifies the type of link between the **subject** and the **object** in a RDF triple.

You can think of a single RDF graph as being hosted in one web service, SPARQL endpoint service, or a downloadable set of RDF files. Just as the value of the web is greatly increased with relevant links between web pages, the value of RDF graphs is increased when they contain references to triples in other RDF graphs. In theory, you could think of all linked RDF data that is reachable on the web as being a single graph but in practice graphs with billions of nodes are difficult to work with. That said, handling very large graphs is an active area of research both in university labs and in industry.

URIs refer to things, acting as a unique identifier. An important idea is that URIs in linked data sources can also be ”dereferenceable:” a URI can serve as a unique identifier for the Semantic Web and if you follow the link you can find HTML, RDF or any document type that might better inform both human readers and software agents. Typically, a dereferenceable URI is ”followed” by using the HTTP protocol’s GET method.

The idea of linking data resources using RDF extends the web so that both human readers and software agents can use data resources. In Tim Berners-Lee’s 2009 TED talk on Linked Data he discusses the importance of getting governments, companies and individuals to share Linked Data and to not keep it private. He makes the great point that the world has many challenges (medicine, stabilizing the economy, energy efficiency, etc.) that can benefit from unlocked Linked Data sources.

9.1. Linked Data Resources on the Web

There are already many useful public Linked Data sources, with more being developed. Some examples are:

1. DBpedia contains the ”info box” data automatically collected from Wikipedia (see Chapter 14).
2. FOAF (Friend of a Friend) Ontology for specifying information about people and their social and business relationships.
3. GeoNames (http://www.geonames.org/) links place names to DBpedia (see Chapter 15).
4. Freebase (http://freebase.com) is a community driven web portal that allows people to enter facts as structured data. It is possible to query Freebase and get results as RDF. (See Chapter 13).

We have already used the FOAF RDFS definitions in examples in this book[1] and we will DBpedia, GeoNames, and Freebase in later chapters.

9.2. Publishing Linked Data

Leigh Dodds and Ian Davis have written an online book ”Linked Data Patterns”[2] that provides useful patterns for defining and using Linked Data. I recommend their book as a more complete reference than this short chapter.

I have used a few reasonable patterns in this book for defining RDF properties, some examples being:

```
<http://knowledgebooks.com/ontology/containsPlace>
<http://knowledgebooks.com/ontology/containsCity>
<http://knowledgebooks.com/rdf/discusses/person>
<http://knowledgebooks.com/rdf/discusses/place>
```

It is also good practice to name resources automatically using a root URI followed by a unique ID based on the data source; for example: a database row ID or a Freebase ID.

```
<http://knowledgebooks.com/rdf/datasource/freebase/20121>
```

[1] As an example, for people’s names, addresses, etc.
[2] Available under a Creative Commons License at http://patterns.dataincubator.org/book/

```
<http://knowledgebooks.com/rdf/datasource/psql/ \\
testdb/testtable/21198>
```

For all of these examples (properties and resources) it would be good practice to make these URIs dereferenceable.

9.3. Will Linked Data Become the Semantic Web?

There has not been much activity building large systems using Semantic Web technologies. That said, I believe that RDF is a natural data format to use for making statements about data found on the web and I expect the use of RDF data stores to increase. The idea of linked data seems like a natural extension: making URIs dereferenceable lets people follow URIs and get additional information on commonly used RDFS properties and resources. I am interested in Natural Language Processing (NLP) and it seems reasonable to expect that intelligent agents can use natural (human) language dereferenced descriptions of properties and resources.

9.4. Linked Data Wrapup

I have defined the relevant terms for using Linked Data in this short chapter and provided references for further reading and research. Much of the rest of this book is comprised of Linked Data application examples using some utilities for information extraction and processing with existing data sources.

Part IV.

Utilities for Information Processing

10. Library for Web Spidering

There are many good web spidering libraries available. Additionally, I offer my own Java implementation in this chapter and examples using this library in Clojure, Scala, and JRuby. We will use this library in the remainder of this book for fetching information from web pages.

10.1. Parsing HTML

For the examples in this book, I want to extract plain text from web pages, even though for some applications using HTML markup can help determine and discard unwanted text from sidebar menus, etc. There are several good open source HTML parsers. I like the Jericho parser[1] because it has a high level API that makes it simple to extract both plain text and links from HTML.

The following snippets from the file WebSpider.java show how to use Jericho:

```
import net.htmlparser.jericho.*;
 ...
    URL url = new URL(url_str);
    URLConnection connection = url.openConnection();
    connection.setAllowUserInteraction(false);
    InputStream ins = url.openStream();
    Source source = new Source(ins);
    TextExtractor te = new TextExtractor(source);
    String text = te.toString();
    List<StartTag> anchorTags =
        source.getAllStartTags("a ");
    ListIterator iter = anchorTags.listIterator();
    // .. process href attribute from each anchor tag
```

It will be useful having linked URLs to fetch linked web pages. In the API for my library you can specify a starting URL and the maximum number of pages to return.

[1] http://jericho.htmlparser.net

10.2. Implementing the Java Web Spider Class

The complete implementation for the web spider class is in the file WebSpider.java in the package com.knowledgebooks.info_spiders. The following snippets show how I manage a queue of web URLs to visit:

```
String host = new URL(root_url).getHost();
List<String> urls = new ArrayList<String>();
Set<String> already_visited = new HashSet<String>();
urls.add(root_url);
int num_fetched = 0;
while (num_fetched < max_returned_pages &&
       !urls.isEmpty()) {
  try {
    String url_str = urls.remove(0);
    if (url_str.toLowerCase().indexOf(host) > -1 &&
        url_str.indexOf("https:") == -1 &&
        !already_visited.contains(url_str)) {
        already_visited.add(url_str);
        URL url = new URL(url_str);
        URLConnection connection =
            url.openConnection();
        // .. process the HTML data from this web page
```

The WebSpider class also needs to be able to handle relative links on a web page. The following code handles absolute and relative links:

```
Attribute link = attr.get("href");
String link_str = link.getValue(); // absolute URL
if (link_str.indexOf("http:") == -1) { // relative URL
    String path = url.getPath();
    if (path.endsWith("/"))
       path = path.substring(0, path.length() - 1);
    int index = path.lastIndexOf("/");
    if (index > -1) path = path.substring(0, index);
    link_str =
        url.getHost() + "/" + path + "/" + link_str;
    link_str = "http://" +
               link_str.replaceAll("///", "/").
                        replaceAll("//", "/");
}
```

10.3. Testing the WebSpider Class

The following code snippet shows how to use the WebSpider class:

```
WebSpider ws =
   new WebSpider("http://www.knowledgebooks.com", 20);
for (List<String> ls : ws.url_content_lists) {
   String url =ls.get(0);
   String text = ls.get(1);
   System.out.println("\n\n\n----URL:\n"+
                      url+"\n    content:\n"+text);
}
```

Each web page is represented by a list of two strings: the page absolute URL and the plain text extracted from the web page. In a pure Java application, I would implement a simple POJO (Plain Old Java Object) class to hold the retrn values. However, since I am most likely to also use this utility class in Clojure and Scala applications, it makes interfacing to those languages a little easier returning a list of list of strings.

10.4. A Clojure Test Web Spider Client

The method **url_content** returns a Java **List**<**List**<**String**>> so I map the function **seq** to the Java result to get a list of lists, the inner list containing a string URL and a string for the plain text web page contents:

```
(import '(com.knowledgebooks.info_spiders WebSpider))

(defn get-pages [starting-url max-pages]
  (let [ws (new WebSpider starting-url max-pages)]
    (map seq (.url_content_lists ws))))

(println (get-pages "http://www.knowledgebooks.com" 2))
```

The output looks like this (with some text removed for brevity):

```
(("http://www.knowledgebooks.com"
    "Knowledgebooks.com: AI Technology for ...")
  ("http://www.knowledgebooks.com/demo.jsp"
    "KB_bundle Demonstration  ..."))
```

10.5. A Scala Test Web Spider Client

The following Scala code snippet calls the Java APIs and leaves the results as **List**<**List**<**String**>>:

```
import com.knowledgebooks.info_spiders.WebSpider

object TestScalaWebSpider {
  def main(args: Array[String]) {
    val results =
      new WebSpider("http://www.knowledgebooks.com", 2)
    println(results.url_content_lists.get(0))
    println(results.url_content_lists.get(1))
  }
}
```

The output is:

```
[http://www.knowledgebooks.com,
 Knowledgebooks.com: AI Technology for ..." ]
[http://www.knowledgebooks.com/demo.jsp,
 KB_bundle Demonstration ..."]
```

10.6. A JRuby Test Web Spider Client

The JRuby example loads all jar files in the lib directory and the knowledgebooks.jar file and then directly calls my Java API:

```
require 'java'
(Dir.glob("lib/*.jar")).each do |fname|
  require fname
end
require "knowledgebooks.jar"
require 'pp'

results =
  com.knowledgebooks.info_spiders.WebSpider.new(
       "http://www.knowledgebooks.com", 2)
pp results.url_content_lists
```

It is necessary to first create the knowledgebooks.jar file before running the example in this section:

```
$ make
```

The output is:

```
$ jruby test/test_ruby_web_spider.rb
[["http://www.knowledgebooks.com",
  "Knowledgebooks.com: AI Technology for ..."],
 ["http://www.knowledgebooks.com/demo.jsp",
  "KB_bundle Demonstration ..."]]
```

10.7. Web Spider Wrapup

For complex web spidering applications I use the Ruby utilities scRUBYt! and Watir[2] that provide fine control over choosing which parts of a web page to extract. For simpler cases when I need all of the text on spidered web pages I use my own library.

[2] I cover both of these tools in my book "Scripting Intelligence, Web 3.0 Information Gathering and Processing" [Apress 2009]

11. Library for Open Calais

The Open Calais web services can be used to create semantic metadata from plain text. OpenCalais can extract proper names (people, locations, companies, etc.) as well as infer relationships and facts. We will be storing this metadata as RDF and using it for several example applications in the remainder of this book.

The Open Calais web services are available for free use with some minor limitations. This service is also available for a fee with additional functionality and guaranteed service levels. We will use the free service in this chapter.

The Thomson Reuters company bought ClearForest, the developer of Open Calais. You can visit the web site http://www.opencalais.com/ to get a free developers key, documentation and code samples for using Open Calais.

You will need to apply for a free developers key. On my development systems I define an environment variable for the value of my key using the following (the key shown is not a valid key):

```
export OPEN_CALAIS_KEY=po2ik1a312iif985f9k
```

The example source file for my utility library is **OpenCalaisClient.java** in the Java package **com.knowledgebooks.info_spiders**.

11.1. Open Calais Web Services Client

The Open Calais web services return RDF payloads serialized as XML data.

For our purposes, we will not use the returned XML data and instead parse the comment block to extract named entities that Open Calais identifies. There is a possibility in the future that the library in this section may need modification if the format of this comment block changes (it has not changed in several years).

I will not list all of the code in OpenCalaisClient.java but we will look at some of it. I start by defining two constant values, the first depends on your setting of the environment variable OPEN_CALAIS_KEY:

```
  String licenseID = System.getenv("OPEN_CALAIS_KEY");
```

The web services client function is fairly trivial: we just need to make a RESTful web services call and extract the text form the comment block, parsing out the named entities and their values. Before we look at some code, we will jump ahead and look at an example comment block; understanding the input data will make the code easier to follow:

```
<!--Relations: PersonCommunication,
               PersonPolitical,
               PersonTravel

Company: IBM, Pepsi
Country: France
Person: Hiliary Clinton, John Smith
ProvinceOrState: California-->
```

We will use the **java.net.URL** and **java.net.URLConnection** classes to make REST style calls to the Open Calais web services. I shortened a few lines in the flowing listing, so also refer to the Java source file.

```
    Hashtable<String, List<String>> ret =
         new Hashtable<String, List<String>>();
    String paramsXML = "<c:params xmlns:c=\"http://...;
    StringBuilder sb =
      new StringBuilder(content.length() + 512);
    sb.append("licenseID=").append(licenseID);
    sb.append("&content=").append(content);
    sb.append("&paramsXML=").append(paramsXML);
    String payload = sb.toString();
    URLConnection connection =
      new URL("http://api.opencalais.com...").
             openConnection();
    connection.addRequestProperty("Content-Type",
            "application/x-www-form-urlencoded");
    connection.addRequestProperty("Content-Length",
                String.valueOf(payload.length()));
    connection.setDoOutput(true);
    OutputStream out = connection.getOutputStream();
    OutputStreamWriter writer =
        new OutputStreamWriter(out);
    writer.write(payload);
    writer.flush();
```

```
    // get response from Open Calais server:
    String result = new Scanner(
                connection.getInputStream()).
                useDelimiter("\\Z").next();
    result = result.replaceAll("&lt;", "<").
                    replaceAll("&gt;", ">");
    //System.out.println(result);
    int index1 = result.indexOf("terms of service.-->");
    index1 = result.indexOf("<!--", index1);
    int index2 = result.indexOf("-->", index1);
    result = result.substring(index1 + 4, index2 - 1 + 1);
    String[] lines = result.split("\\n");
    for (String line : lines) {
      int index = line.indexOf(":");
      if (index > -1) {
        String relation = line.substring(0, index).trim();
        String[] entities = line.substring(index + 1).
                                trim().split(",");
        for (int i = 0, size = entities.length;
             i < size; i++) {
            entities[i] = entities[i].trim();
        }
        ret.put(relation, Arrays.asList(entities));
    }
  }
  return ret;
```

The file TestOpenCalaisClient.java shows how to use the OpenCalaisClient utility class:

```
String content = "Hillary Clinton likes to remind ...";
Map<String, List<String>> results =
  new OpenCalaisClient().
      getPropertyNamesAndValues(content);
for (String key : results.keySet()) {
  System.out.println("  " + key + ": " +
                     results.get(key));
}
```

The following shows the output:

```
Person: [Al Gore, Doug Hattaway, Hillary Clinton]
Relations: [EmploymentRelation, PersonTravel]
```

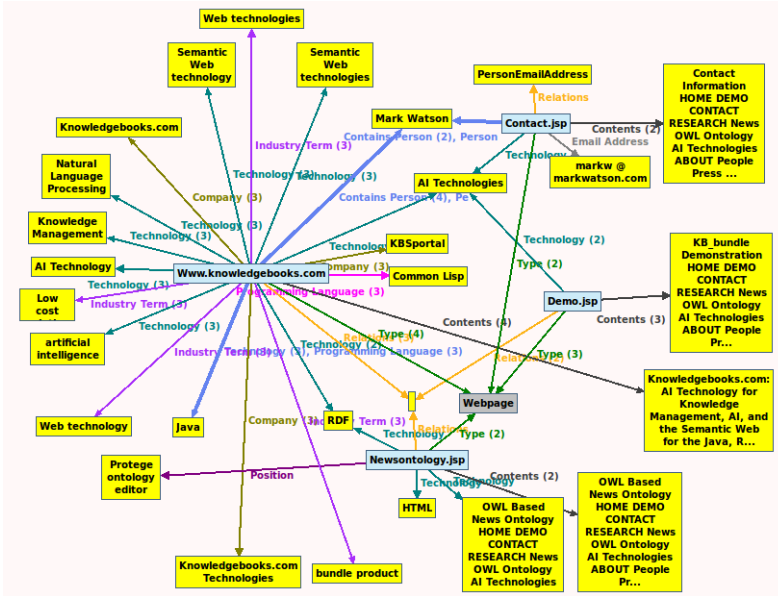

Figure 11.1.: Generated RDF viewed in Gruff

```
City: [San Francisco]
Country: [France, Spain, United States]
ProvinceOrState: [Texas]
```

11.2. Using OpenCalais to Populate an RDF Data Store

We will use the utilities developed in the last section for using the Open Calais web services in this section to populate an RDF data store. This example will be simple to implement because I am using the web spider utilities from Chapter 10 and either the AllegroGraph wrapper (Chapter 4) or the Sesame wrapper (Chapter 5). I will spider a few pages from my knowedgebooks.com web site, use the Open Calais web service to identify entities and relations contained in the spidered web pages, and then do two things: write generated RDF to a file and also store generated RDF in a data store and perform a few example SPARQL queries.

As an example of the type of RDF data that we will pull from my knowledgebooks.com web site, look at Figure 11.1 that shows generated RDF for two spidered web pages in the Gruff RDF viewer.

The following code snippet shows the collection of page content from my web site:

```
//RdfServiceProxy proxy =
//  new AllegroGraphServerProxy();
RdfServiceProxy proxy = new SesameEmbeddedProxy();
proxy.deleteRepository("knowledgebooks_repo");
proxy.createRepository("knowledgebooks_repo");
proxy.registerFreetextPredicate(
     "http://knowledgebooks.com/rdf/contents");
WebSpider ws =
  new WebSpider("http://www.knowledgebooks.com", 2);
```

Here I have connected to an RDF server, created a fresh repository, registered the predicate http://knowledgebooks.com/rdf/contents to trigger indexing the text in all triples that use this predicate and finally, fetched (spidering) two pages from my web site.

The following code snippet from the source file OpenCalaisWebSpiderToRdfFile.java creates a print writer for saving data to a file and then loops through the page data returned from the web spider utility class:

```
PrintWriter out =
    new PrintWriter(new FileWriter("out.nt"));
for (List<String> ls : ws.url_content_lists) {
  String url =ls.get(0);
  String text = ls.get(1);
  Map<String, List<String>> results =
      new OpenCalaisClient().
          getPropertyNamesAndValues(text);
  out.println("<"+url+"> "+
    "<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> \\
     <http://knowledgebooks.com/rdf/webpage> .");
  out.println("<"+url+"> " +
    "<http://knowledgebooks.com/rdf/contents> \"" +
     text.replaceAll("\"", "'") + "\" .");
  if (results.get("Person")!=null)
    for (String person : results.get("Person")) {
     out.println("<"+url+"> " +
       "<http://knowledgebooks.com/rdf/containsPerson> \""
       + person.replaceAll("\"", "'") + "\" .");
  }
  for (String key : results.keySet()) {
    for (Object val : results.get(key)) {
      out.println("<"+url+"> " +
```

```
          "<http://knowledgebooks.com/rdf/" +
          key + "> \"" + val + "\" .");
      }
    }
  }
  out.close();
```

There is little new code in this example because I am using the results of the **WebSpider** class (Chapter 10) and my Open Calais client class. I am using these results to create RDF statements linking original URLs with **containsPerson** and **contents** properties to string literal values.

The following code snippet is similar to the last one but here I am writing data to an RDF data store instead of a flat text file:

```
for (List<String> ls : ws.url_content_lists) {
  String url =ls.get(0);
  String text = ls.get(1);
  Map<String, List<String>> results =
      new OpenCalaisClient().
          getPropertyNamesAndValues(text);
  proxy.addTriple(url, Triple.RDF_TYPE,
    "http://knowledgebooks.com/rdf/webpage");
  proxy.addTriple("<"+url+">",
    "http://knowledgebooks.com/rdf/contents", "\"" +
    text.replaceAll("\"", "'") + "\"");
  for (String key : results.keySet()) {
    for (Object val : results.get(key)) {
       proxy.addTriple(url,
         "http://knowledgebooks.com/rdf",
         "\"" + text.replaceAll("\"", "'") + "\"");
    }
  }
}
```

We can print all triples in the data store using this code snippet:

```
System.out.println("\n\nSample queries:\n");
List<List<String>> results =
  proxy.query("SELECT ?s ?p ?o  WHERE {?s ?p ?o .}");
for (List<String> result : results) {
   System.out.println("All triples result: " + result);
}
```

The following snippet shows how to perform text search in the RDF data store (this works with both the AllegroGraph and my Sesame wrappers):

```
  List<List<String>> results = proxy.textSearch("Lisp");
  for (List<String> result : results) {
    System.out.println(
        "Wild card text search result: " + result);
  }
```

and output from both the last two code snippets is (most output is not shown and I split lines to fit the page width):

```
All triples result:
  [http://www.knowledgebooks.com,
   http://www.w3.org/1999/02/22-rdf-syntax-ns#type,
   http://knowledgebooks.com/rdf/webpage]
All triples result:
  [http://www.knowledgebooks.com,
  http://knowledgebooks.com/rdf/contents,
  Knowledgebooks.com: AI Technology for Knowledge \\
  Management, AI, and the Semantic Web for the Java,\\
  Ruby, and Common Lisp Platforms ...]
Wild card text search result:
  [<http://www.knowledgebooks.com>,
   <http://knowledgebooks.com/rdf/contents>,
  Knowledgebooks.com: AI Technology for Knowledge \\
  Management, AI, and the Semantic Web for the Java,\\
  Ruby, and Common Lisp Platforms ...]
```

11.3. OpenCalais Wrap Up

Since AllegroGraph (and my wrapper using a Sesame back end) supports indexing and search of any text fields in triples, the combination of using triples to store specific entities in a large document collection with full search, AllegroGraph or Sesame can an tool to mange large document repositories.

"Documents" can be any source of text identified with a unique URI: web pages, word processing documents, blog entries, etc.

I will show you my Natural Language Processing (NLP) library in the next chapter. My library does not perform as well as Open Calais for entity extraction but has other

features like automatic summarizing and calculating a short list of key terms that are useful for searching for similar material using search engines like Google or Bing.

12. Library for Entity Extraction from Text

I have been working on Natural Language Processing (NLP) software since the 1980s. In recent years I have, frankly, been using the Open Calais system (covered in Chapter 11) more than my own KnowledgeBooks.com software because Open Calais performs better for entity extraction. That said, I still find it useful to be able to perform entity extraction and text summarization in a local (non-web services) library. I also find my own library is easier to extend as I did recently to add the ability to determine search terms that would be likely to get you back to a given page using a search engine (Section 12.1.5).

12.1. KnowledgeBooks.com Entity Extraction Library

You can find a simplified version of my KnowledgeBooks.com source code in the software distribution for this book[1] in the Java packages com.knowledgebooks.nlp and com.knowledgebooks.nlp.util. If you want to experiment with this code then I will leave it to you to read through the source code. Here we will just take a quick look at the public APIs of the most important Java classes.

12.1.1. Public APIs

The *Document* class is constructed either by passing the constructor a string of text to process or a list of strings. This class provides behavior of providing both word tokens and sentence boundaries. Here are the APIs that you might find useful in your own programs:

```
public int getNumWords()
public String getWord(int wordIndex)
```

[1] I reduced the size of my full NLP library by about two-thirds, leaving the most useful parts and hopefully making it easier for you to experiment with the code.

```
public int getNumSentences()
public IPair getSentenceBoundaryFromWordIndex(int wordIndex)
public IPair getSentenceBoundary(int sentenceIndex)
public String getSentence(int index)
```

12.1.2. Extracting Human and Place Names from Text

The class *ExtractNames* is a top-level utility class to extract human and place names from text. Initializing an instance of this class has some overhead for the first instance created because the file data/propername.ser needs to be read into memory and static hash tables are created. The public APIs that you may want to use in your applications are:

```
public ScoredList[] getProperNames(String s)
```

A *ScoredList* instance contains a list of strings, each with an associated numeric score. The method *getProperNames* returns an array of two instances of *ScoredList* (the first for human names and the second for place names).

There are other APIs for testing to see if strings are human names or place names. Here is an example of using this class:

```
ExtractNames extractNames = new ExtractNames();
ScoredList[] ret = extractNames.getProperNames(
   "George Bush played golf. President George W. Bush \\
    went to London England and Mexico and then see \\
    Mary Smith in Moscow. President Bush will \\
    return home Monday.");
 System.out.println("Human names: " +
           ret1[0].getValuesAsString());
 System.out.println("Place names: " +
           ret1[1].getValuesAsString());
```

Output for this example looks like:

```
Human names: George Bush:1, President George W . Bush:1,
Mary Smith:1, President Bush:1
Place names: London:1, Mexico:1, Moscow:1
```

12.1.3. Automatically Summarizing Text

When I am dealing with documents that contain a lot of text, I often like to calculate a summary of the text for display purposes. The class *KeyPhraseExtractionAndSummary* processes text to return both a list of key phrases and a summary.

```
String s = "Sales of 10 cotton cloth and raw silk \\
            cocoons  are down in England and France \\
            due to competition from India. Cotton is \\
            easy to wash. President Bush, wearing a \\
            Strouds shirt, and Congress are concerned \\
            about US cotton and riso and Riso sales. \\
            Airline traffic is down this year.";
KeyPhraseExtractionAndSummary e =
   new KeyPhraseExtractionAndSummary(s);
int num = e.getNumPhrases();
for (int i=0; i<num; i++) {
  System.out.println("" + e.getScore(i) + " : "
                     + e.getPhrase(i));
}
System.out.println("\nSummary:\n"+e.getSummary());
```

The output is:

```
9262.499 : President Bush, wearing a Strouds shirt, \\
and Congress are concerned about US cotton and riso \\
and Riso sales. 6987.4995 : Sales of 10 cotton cloth \\
and raw silk cocoons are down in England and France \\
due to competition from India.

Summary:
President Bush, wearing a Strouds shirt, and Congress \\
are concerned about US cotton and riso and Riso sales. \\
Sales of 10 cotton cloth and raw silk cocoons are down \\
in England and France due to competition from India.
```

The algorithm that I use for key phrase and summarization is strongly dependent on the ability to classify text. I use the Java class *AutoTagger* to assign a list of rated categories for text. Categories are assigned by looking words up in word frequency count hash tables, one hash table per category. The raw word counts per category data is in the file data/tags.xml. This file contains little data (about 5000 words for all categories) for fast initialization and low memory use (because the hash tables are

stored in memory). I use two techniques in my projects for making the auto tagging (or auto categorization) and therefore summarization more accurate:

1. I use much more single word data, often over 100,000 words.
2. I also collect statistics on word pairs, find the word pairs most often used together in categories, and add this to single word frequency results.

12.1.4. Classifying Text: Assigning Category Tags

If you want to just auto tag text and not summarize it then use the following code snippet as an example:

```
AutoTagger test = new AutoTagger();
List<NameValue<String, Float>> results =
  test.getTags("The President went to Congress to
     argue for his tax bill before leaving on a vacation
     to Las Vegas to see some shows and gamble.");
for (NameValue<String, Float> result : results) {
  System.out.println(result);
}
```

The output is:

```
[NameValue: news_economy : 1.0]
[NameValue: news_politics : 0.84]
[NameValue: news_weather : 0.8]
[NameValue: health_exercise : 0.32]
[NameValue: computers_microsoft : 0.32]
[NameValue: computers : 0.24]
[NameValue: religion_islam : 0.24]
```

My NLP code has been reworked over a ten-year period and is not in a ”tidy state” but I made some effort to pull out just some useful bits that you may find useful, especially if you customize the data in the tags.xml file for your own applications.

12.1.5. Finding the Best Search Terms in Text

This is an interesting problem: given a web page, determine a few words that would likely get you back to the page using a search engine. Or, given text on a web page, determine key words for searching for similar information.

In Chapter 13 we will use this to find relevant objects in Freebase that match entities we extract from text.

The algorithm is faily simple: first, I will use the *AutoTagger* class to find the rated category tags for text. I will then repeat the core calculation in the *AutoTagger* class keeping track of how much each word contributes evidence to the most likely category tags. You can find the code in the source file ExtractSearchTerms.java. We will look at the most interesting parts of this code here.

I start by getting the weighted category tags, and creating lists of words and tags in the input text:

```
public ExtractSearchTerms(String text) {
  // this code is not so efficient since I first need
  // to get the best tags for the input text, then go
  // back and keep track of which words provide the
  // most evidence for selecting these tags.
  List<NameValue<String, Float>> tagResults =
                      new AutoTagger().getTags(text);
  Map<String,Float> tagRelevance =
                      new HashMap<String,Float>();
  for (NameValue<String, Float> nv : tagResults) {
      tagRelevance.put(nv.getName(), nv.getValue());
  }
  List<String> words = Tokenizer.wordsToList(text);
  int number_of_words=words.size();
  Stemmer stemmer = new Stemmer();
  List<String> stems =
      new ArrayList<String>(number_of_words);
  for (String word : words)
      stems.add(stemmer.stemOneWord(word));
```

Next, I loop over all words in the input text, discard stop (or "noise") words, and see if the stem for each word is relevant to any of the tag/classification types. I am looking for the individual words that most strongly help to tag/classify this text:

```
  int number_of_tag_types =
      AutoTagger.tagClassNames.length;
  float[] scores = new float[number_of_words];
  for (int w=0; w<number_of_words; w++) {
    if (NoiseWords.checkFor(stems.get(w))== false) {
      for (int i = 0; i < number_of_tag_types; i++) {
        Float f =
          AutoTagger.hashes.get(i).get(stems.get(w));
```

```
          if (f != null) {
            Float tag_relevance_factor =
             tagRelevance.get(AutoTagger.tagClassNames[i]);
            if (tag_relevance_factor != null) {
              scores[w] += f * tag_relevance_factor;
            }
          }
        }
      }
    }
```

The individual words that most strongly help to tag/classify the input text are saved as the recommended search terms:

```
    float max_score=0.001f;
    for (int i=0; i<number_of_words; i++)
      if (max_score < scores[i]) max_score = scores[i];
    float cutoff = 0.2f * max_score;
    for (int i=0; i<number_of_words; i++) {
      if (NoiseWords.checkFor(stems.get(i))==false) {
        if (scores[i] > cutoff)
             bestSearchTerms.add(words.get(i));
      }
    }
  }
  public List<String> getBest() {
   return bestSearchTerms;
  }
  private List<String> bestSearchTerms =
     new ArrayList<String>();
```

Here is the test code in test/TestExtractSearchTerms.java:

```
  String s = "The President went to Congress to argue \\
    for his tax bill passed into law before leaving   \\
    on a vacation to   Las Vegas to see some shows    \\
    and gamble. However, too many Senators are against\\
    this spending bill.";
  ExtractSearchTerms extractor = new ExtractSearchTerms(s);
  System.out.println("Best search terms " + extractor.getBest())
```

The output is:

```
Best search terms [tax, passed, law, spending]
```

In applications, it is very important to create a good stop (or "noise") word list. This list will be very dependent on the type of information your application is processing. For example, you would use different stop word lists for sports vs. news information processing applications. I specified my test stop word list in the source file *NoiseWords.java* in the package *com.knowledgebooks.nlp.util*.

12.2. Examples Using Clojure, Scala, and JRuby

If you are using Java then feel free to skip the following three short sections. I find myself often reusing Java libraries in a more concise programming language like Clojure, Scala, or JRuby. I wrote the language wrappers in the next three sections for my own use and you may also find them to be useful. [2]

12.2.1. A Clojure NLP Example

It is simple to wrap my Java NLP classes for easier use in Clojure programs. Here is the Clojure wrapper that I use:

```
(ns nlp_clojure)

(import
 '(com.knowledgebooks.nlp
   AutoTagger
   KeyPhraseExtractionAndSummary
   ExtractNames)
 '(com.knowledgebooks.nlp.util
   NameValue
   ScoredList))

(def auto-tagger (AutoTagger.))
(def name-extractor (ExtractNames.))

(defn get-auto-tags [text]
  (seq (map to-string (.getTags auto-tagger text))))
(defn get-names [text]
```

[2] For those of you who have read any of my previous books, you know that I mostly write about topics from my own research and for consulting work for customers. I "re-purpose" the libraries that I create for these projects when I write.

```
  (let [[names places]
     (.getProperNames name-extractor text)]
    [(seq (.getStrings names))
     (seq (.getStrings places))]))
(defn get-summary [text]
  (.getSummary
    (new KeyPhraseExtractionAndSummary text)))

;; utility:
(defn to-string [obj] (.toString obj))
```

Here is a small Clojure test script:

```
(use 'nlp_clojure)

(println (get-auto-tags "The President went to Congress"))
(println (get-names
  "John Smith talked with Carol Jones in London last week."))
(println (get-summary "The Columbia Slough is a narrow..."))
```

```
(news:1.0 news_war:0.8 news_politics:0.8
 computers_ai_textmining:0.6 religion_islam:0.6)
[(John Smith Carol Jones) (London)]
One of the nations largest freshwater urban wetlands, \\
Smith and Bybee Wetlands Natural Area, shares the lower \\
slough watershed with a sewage treatment plant, marine \\
terminals, a golf course, and a car racetrack.  The \\
Columbia Slough is a narrow waterway, about 19 miles \\
( 31 km ) long, in the floodplain of the Columbia River \\
in the U.S. state of Oregon.
```

12.2.2. A Scala NLP Example

Here is the Scala wrapper that I use for my Java NLP library:

```
package nlp_scala

import com.knowledgebooks.nlp.{AutoTagger,
      KeyPhraseExtractionAndSummary,
      ExtractNames}
import com.knowledgebooks.nlp.util.{NameValue, ScoredList}
```

```
class NlpScala {
  val auto_tagger = new AutoTagger
  val name_extractor = new ExtractNames
  def get_auto_tags(s : String) = {
    // return a Scala List containing instances
    // of java.lang.String:
    auto_tagger.getTagsAsStrings(s).toArray.toList
  }
  def get_names(s : String) = {
    val result : java.util.List[java.util.List[String]] =
       name_extractor.getProperNamesAsStrings(s)
    // return a List 2 elements: first is a list of human
    // name strings, second a list of place name strings:
    List((result.get(0).toArray.toList,
      result.get(1).toArray.toList))
  }
}
```

Here is a short test program:

```
import nlp_scala.NlpScala

object TestScalaNlp{
  def main(args: Array[String]) {
    var test = new NlpScala
    val results =  test.get_auto_tags("President Obama \\
       went to Congress to talk about taxes")
    println(results)
    val names = test.get_names("Bob Jones is in Canada \\
       and is then meeting John Smith in London")
    println(names)
  }
}
```

The output from this test is:

```
List(news_economy:1.0, news_politics:0.95454544,
    health_exercise:0.36363637, news:0.22727273)
List((List(Bob Jones:1, John Smith:1),
    List(Canada:1, London:1)))
```

12.2.3. A JRuby NLP Example

Here is the JRuby wrapper that I use for my Java NLP library (with some long lines broken to fit the page width):

```
require 'java'
(Dir.glob("lib/*.jar") +
 Dir.glob("lib/sesame-2.2.4/*.jar")).each do |fname|
  require fname
end
require "knowledgebooks.jar"

class NlpRuby
  def initialize
    @auto_tagger = \\
      com.knowledgebooks.nlp.AutoTagger.new
    @extract_names = \\
      com.knowledgebooks.nlp.ExtractNames.new
  end
  def get_tags text
    @auto_tagger.getTags(text).collect do |name_value|
      [name_value.getName, name_value.getValue]
    end
  end
  def get_proper_names text
    @extract_names.getProperNames(text). \\
        collect do |scored_list|
      scored_list.getStrings.zip(scored_list.getScores)
    end
  end
end
```

Here is a short test program:

```
require 'src/nlp_ruby'
require 'pp'

nlp = NlpRuby.new
tags = nlp.get_tags("The President went to \\
Congress to argue for his tax bill before \\
leaving on a vacation to Las Vegas to see \\
some shows and gamble.")
pp tags
```

```
names = nlp.get_proper_names("John Smith went \\
to France and Germany with Sam Jones.")
pp names
```

The output from this test is:

```
# last names=100427, # first names=5287
[["news_economy", 1.0],
 ["news_politics", 0.839999973773956],
 ["news_weather", 0.800000011920929],
 ["health_exercise", 0.319999992847443],
 ["computers_microsoft", 0.319999992847443],
 ["computers", 0.239999994635582],
 ["religion_islam", 0.239999994635582]]
[[["John Smith", 1], ["Sam Jones", 1]],
 [["France", 1], ["Germany", 1]]]
```

12.3. Saving Entity Extraction to RDF and Viewing with Gruff

This example will be very similar to the example using Open Calais (Chapter 11). I will use my NLP library as described in this chapter with the web spider tools from Chapter 10. However, this example will be a little more complicated because I will generate per web page properties for:

1. People mentioned on the web page
2. Places mentioned on the web page
3. Short summary of text on the web page
4. Automatically generated tags for topics found on the web page

I will also calculate web page similarity and generate properties for this as I did in Chapter 11. With the generated RDF data loaded into either AllegroGraph or Sesame we will be able to perform queries to find, for example:

1. All web pages that mention a specific person or place
2. All web pages in a specific category

The example code for this section can be found in the file *KnowledgeBooksNlpGenerateRdfPropertiesFromWebPages.java* in the examples directory. Before looking at a

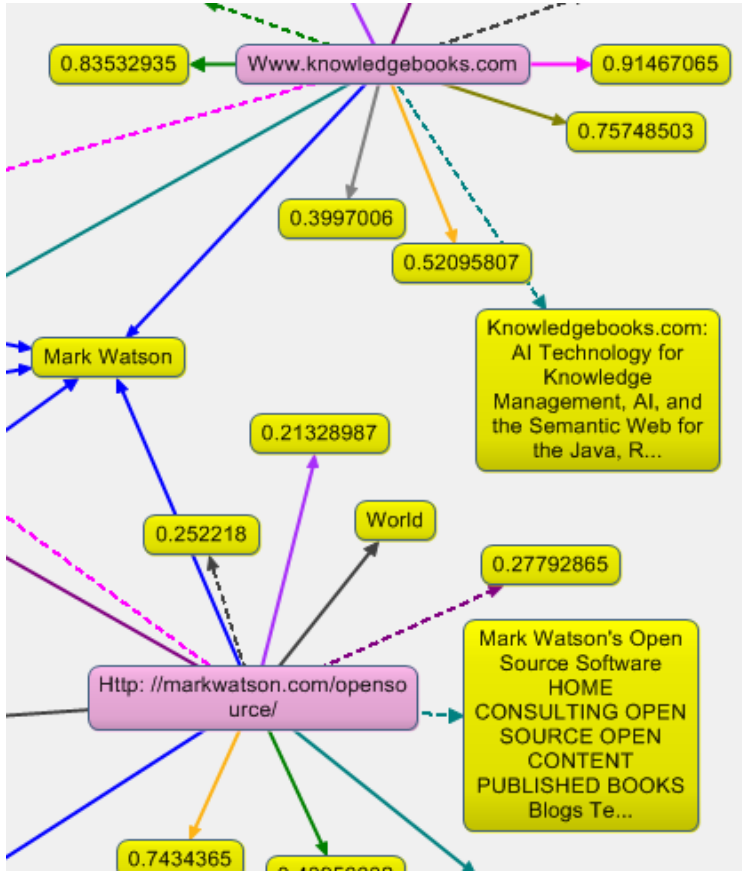

Figure 12.1.: RDF generated with KnowledgeBooks NLP library viewed in Gruff. Arrows represent RDF properties.

few interesting bits of this example code, you can see an example of the type of RDF data that we will pull from my knowledgebooks.com and markwatson.com web sites look at Figure 12.1 that shows generated RDF for two spidered web pages in the Gruff RDF viewer.

The example class *KnowledgeBooksNlpGenerateRdfPropertiesFromWebPages* reads a configuration file containing web sites to spider and how many pages to fetch from each site; here is an example configuration file that you can find in testdata/websites.txt:

```
http://www.knowledgebooks.com 4
http://markwatson.com 4
```

The class constructor takes a path to a configuration file and a PrintWriter object used to output RDF N-Triple data:

```
public KnowledgeBooksNlpGenerateRdfPropertiesFromWebPages(
                 String config_file_path, PrintWriter out)
                                        throws IOException {
  this.out = out;
  extractNames = new ExtractNames();
  autoTagger = new AutoTagger();
  List<String> lines =
    (List<String>)FileUtils.readLines(
                     new File(config_file_path));
  for (String line : lines) {
    Scanner scanner = new Scanner(line);
    scanner.useDelimiter(" ");
    try {
      String starting_url = scanner.next();
      int spider_depth = Integer.parseInt(scanner.next());
      spider(starting_url, spider_depth);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

The method *spider* does most of the real work, starting with spidering a web site and looping through the returned URLs and page content as a text string:

```
  WebSpider ws = new WebSpider(starting_url, spider_depth);
  for (List<String> ls : ws.url_content_lists) {
    String url = ls.get(0);
    String text = ls.get(1);
```

I then get the people, places, and classification tags for each web page:

```
    ScoredList[] names =
       extractNames.getProperNames(text);
    ScoredList people = names[0];
    ScoredList places = names[1];
    List<NameValue<String, Float>> tags =
       autoTagger.getTags(text);
```

Generating RDF N-Triples for the people, places, and clasi=sification tags is simple; for example, here I am generating RDF to record that web pages contain place names:

```
    for (String place : places.getStrings()) {
      out.println("<" + url +
        "> <http://knowledgebooks.com/rdf/containsPlace> \""
        + place.replaceAll("\"", "'") + "\" .");
    }
```

Private method *process_interpage_shared_properties* is used by method *spider* to output RDF data with predicates

```
<http://knowledgebooks.com/rdf/high_similarity>
<http://knowledgebooks.com/rdf/medium_similarity>
<http://knowledgebooks.com/rdf/low_similarity>
```

that are assigned based on the number of common classification tags shared by two web pages.

12.4. NLP Wrapup

My NLP library can be used instead of Open Calais or in conjunction with Open Calais to supply additional functionality. The RDF generating example at the end of this chapter will be expanded later in Chapter 16 after we look at techniques for using Freebase, DBpedia, and GeoNames in the next three chapters. The example in this chapter will be expanded in Chapter 16 to also use these additional information sources.

13. Library for Freebase

Freebase is a public data source created by the MetaWeb Corporation. Freebase is similar to Wikipedia because users of Freebase add data that is linked to other data in Freebase. If you are not already familiar with Freebase then I suggest you spend some time experimenting with the web interface (http://freebase.com) before working through this chapter. As a developer make sure that you eventually look at the developer documentation at http://www.freebase.com/docs/data because I will only cover the aspects of Freebase that I need for the example applications in this book.

13.1. Overview of Freebase

Objects stored in Freebase have a unique object ID assigned to them. It makes sense to use this ID as part of a URI when generating URIs to use as RDF resources. We talked about dereferenceable URIs in Section 6.3. The RDF for the object representing me on Freebase can be obtained by dereferencing:

```
http://rdf.freebase.com/rdf/ \\
guid.9202a8c04000641f80000000146fb902
```

Objects in Freebase are tagged with one or more types. For example, if I search for myself and fetching HTML output using a URI like:

```
http://www.freebase.com/search?query=Mark+Watson+consultant
```

then I see that I am assigned to three types: Person, Author, and Software Developer. If I want JSON formatted results then I can use:

```
http://www.freebase.com/api/service/search?query= \\
Mark+Watson+author
```

A full reference of API arguments is http://www.freebase.com/view/en/api_service_search and Table 13.1 shows the arguments that I most frequently use.

If you try either of the two previous queries (returning either HTML or JSON) you will see several results. If I want results for just people, I can try either of:

Table 13.1.: Subset of Freebase API Arguments

Argument	Argument type	Format	Default value
query	required	string	
type	optional	string	/location/citytown
limit	optional	integer	20
start	optional	integer	0

```
http://www.freebase.com/search?type=/people/person \\
&query=Mark+Watson+consultant

http://www.freebase.com/api/service/search \\
?type=/people/person&query=Mark+Watson
```

to return HTML or JSON results for type /people/person. If you try the second of these queries in a web browser you can see the raw JSON payload. The JSON output will look something like:

```
  "status": "200 OK",
  "code": "/api/status/ok",
  "result":
   [
    {"alias": ["Mark Watson"],
     "article":
       {"id": "/guid/9202a8c04000641f80000000146fb98f"},
     "guid": "#9202a8c04000641f80000000146fb902",
     "id": "/guid/9202a8c04000641f80000000146fb902",
     "image": null, "name": "Mark Louis Watson",
     "relevance:score": 20.821449279785156,
     "type": [
       {"id": "/common/topic", "name": "Topic"},
       {"id": "/people/person", "name": "Person"},
       {"id": "/book/author", "name": "Author"},
       {"id": "/computer/software_developer",
         "name": "Software Developer"}]
    }
  ],
  "transaction_id":
    "cache;cache01.p01.sjc1:811;2010-02-28T20:53:4Z;087"
}
```

Here the result array contains only one hash table result. Hopefully this example will motivate you to learn to use Freebase as an information source for Semantic Web

applications. Notice that the example RDF query my guid that was the first example in this section uses the GUID value returned in this last search example.

In the next section we will look at the MQL query language that uses JSON to specify queries.

13.1.1. MQL Query Language

We saw in the last section how to use REST style web service calls to return HTML or JSON search results. I recommend that you eventually read through the MQL documentation at http://www.freebase.com/docs. For now, I am going to show you some MQL examples that will be sufficient for the example code later in this chapter. The Freebase documentation refers to the query syntax as filling in the blanks and I think that this is a good description. For example, here is some JSON from the last data snippet with some hash values replaced with "null" to match single values and [] to match multiple values:

```
[{
  "type" : "/people/person",
  "name" : "Mark Louis Watson",
  "id" : null
}]
```

You can test MQL queries using the input form on http://www.freebase.com/app/queryeditor. The last example MQL returns my guid:

```
{
  "code":          "/api/status/ok",
  "result": [{
    "id":   "/guid/9202a8c04000641f80000000146fb902",
    "name": "Mark Louis Watson",
    "type": "/people/person"
  }],
  "status":        "200 OK",
  "transaction_id": "cache;cache01.p01.sjc1:8101;2010-03-01T20:13
}
```

There is only one person named "Mark Louis Watson" in Freebase (as I write this) but if I replace my full name with just "Mark Watson" then I get 7 results. As another example, this MQL query gets an array of computer book authors:

```
{ "id": [],
```

```
  "type":"/book/author",
  "type":"/computer/software_developer"
}
```

MQL's powerful tree matching query processing works very well if you know what types are available to match against.

13.1.2. Geo Search

Freebase contains geo search functionality. First, there are many Freebase types that represent physical locations; for example:

```
[{
  "name" : "Flagstaff",
  "type" : "/location/citytown",
  "id": null
}]
```

returns two results:

```
{
  "code":           "/api/status/ok",
  "result": [
    {
      "id":   "/en/flagstaff",
      "name": "Flagstaff",
      "type": "/location/citytown"
    },
    {
      "id":   "/en/flagstaff_maine",
      "name": "Flagstaff",
      "type": "/location/citytown"
    }
  ],
  "status":         "200 OK",
  "transaction_id":
    "cache;cache01.p01.sjc1:8101;2010-03-01T20:25:47Z;0004"
}
```

The first result is the one I expected (Flagstaff is a city about one hour north of where I live in Arizona) and the second result was a surprise. I prefer to use the REST-based geo search APIs; an example query:

```
http://www.freebase.com/api/service/geosearch? \\
location_type=/location/citytown&location=Flagstaff
```

Using the REST geo search API, I get only one result, Flagstaff, the city to the north of where I live (we will use this output in the next section so you will want to refer back to this later):

```
{
  "features": [
    {
      "geometry": {
        "coordinates": [
          -111.6506,
          35.1981
        ],
        "id": "#9202a8c04000641f800000000114e2b9",
        "type": "Point"
      },
      "id": "#9202a8c04000641f800000000006e342",
      "properties": {
        "/common/topic/image": [
          {
            "guid": "#9202a8c04000641f80000000049146fd",
            "id": "/wikipedia/images/commons_id/7036927",
            "index": null,
            "type": "/common/image"
          }
        ],
        "guid": "#9202a8c04000641f800000000006e342",
        "id": "/en/flagstaff",
        "name": "Flagstaff",
        "type": [
          "/common/topic",
          "/location/location",
          "/location/citytown",
          "/location/dated_location",
          "/location/statistical_region",
          "/metropolitan_transit/transit_stop",
          "/film/film_location",
          "/location/hud_county_place",
          "/location/hud_foreclosure_area"
        ]
      },
      "type": "Feature"
```

```
    }
  ],
  "type": "FeatureCollection"
}
```

There are nine types assigned to Flagstaff, Arizona, that you could separately query values for. Using MQL and Freebase is a large topic and we have already covered enough for the example programs later in this book. I am going to finish up with one more example that might provide you with some ideas for your own projects, finding businesses of a specific type near Freebase locations. Here I am asking for a maximum of two restaurants within five miles of Flagstaff:

```
http://www.freebase.com/api/service/geosearch? \\
location=/en/flagstaff&type=/dining/restaurant& \\
within=5&limit=2
```

The JSON payload returned is:

```
{
  "features": [
    {
      "geometry": {
        "coordinates": [
          -111.649189,
          35.197632
        ],
        "id": "#9202a8c04000641f8000000003e273b0",
        "type": "Point"
      },
      "id": "#9202a8c04000641f8000000003e273af",
      "properties": {
        "/common/topic/image": [

        ],
        "guid": "#9202a8c04000641f8000000003e273af",
        "id": "/en/alpine_pizza",
        "name": "Alpine Pizza",
        "type": [
          "/dining/restaurant",
          "/common/topic",
          "/business/business_location",
          "/business/employer"
        ]
```

```
      },
      "type": "Feature"
    },
    {
      "geometry": {
        "coordinates": [
          -111.661408,
          35.18331
        ],
        "id": "#9202a8c04000641f8000000003e273ef",
        "type": "Point"
      },
      "id": "#9202a8c04000641f8000000003e27554",
      "properties": {
        "/common/topic/image": [

        ],
        "guid": "#9202a8c04000641f8000000003e27554",
        "id": "/en/busters_restaurant_bar",
        "name": "Busters Restaurant & Bar",
        "type": [
          "/dining/restaurant",
          "/common/topic",
          "/business/business_location",
          "/business/employer"
        ]
      },
      "type": "Feature"
    }
  ],
  "type": "FeatureCollection"
}
```

Some people criticize the Semantic Web for not having a sufficient number of public linked data sources - I suspect that these people have never used Freebase or DBPedia (Chapter 14). For the remainder of this chapter, we will look at some programming examples using Freebase.

13.2. Freebase Java Client APIs

MetaWeb provides client APIs in several languages. There is a copy of their API Java code (released under a MIT style license) in the package *com.freebase* in the software

distribution for this book.

As you have seen so far in this chapter, any programming language with both network libraries for client REST HTTP requests and for handling JSON data can be used to access Freebase linked data.

Freebase provides an open source Java API for both handling JSON and Freebase web service calls. I will use this library in all of the following examples. There are three Java source files that we will use:

1. src/com/knowledgebooks/info_spiders/FreebaseClient.java wraps the Freebase Java APIs.
2. examples/FreebaseToRdf.java is a convenience wrapper that performs both keyword search and geolocation lookups.
3. examples/EntityToRdfHelpersFreebase.java is developed in Chapter 16 and will be used to match entities in input text to article GUIDs in Freebase.

Most of the code snippets in this section are all in the file FreebaseToRdf.java in the examples directory. I will perform the same queries that I showed in the last section to avoid having to list the lengthly JSON output. To get started, the following code snippet searches Freebase for "Mark Louis Watson author consultant":

```
String q = "Mark Louis Watson author consultant";
Freebase freebase = Freebase.getFreebase();
JSON results =
  freebase.search(q, new HashMap<String,String>());
System.out.println(results.toString());
```

In the last section I listed the JSON output returned from a geo search on "Flagstaff" and you might want to take another look at that output before looking at the following code snippet that picks apart this JSON to get at the latitude and longitude coordinates of Flaggstaff:

```
String location = "Flagstaff";
JSON results =
  freebase.geosearch(location,
                     new HashMap<String,String>());
System.out.println("Test geo search:\n" +
                   results.toString());
System.out.println("Test geo search result:\n" +
                   results.get("result").toString());
System.out.println("Test geo search features:\n" +
    results.get("result").get("features").toString());
```

```
  System.out.println("Test geo search first feature:\n" +
      results.get("result").get("features").get(0). \\
      toString());
  System.out.println("Test geo search geometry:\n" +
      results.get("result").get("features").get(0). \\
        get("geometry").toString());
  System.out.println("Test geo search coordinates:\n" +
      results.get("result").get("features").get(0). \\
        get("geometry").get("coordinates").toString());
```

You may have noticed that when I search for ”Flagstaff” using the geo search API I get only one result. This is a limitation of the geo search Java library. As an example, if I search for ”Berkeley” using the search API then I get a large number of results. However, a search for ”Berkeley” using the geo search API returns only the first result that happens to correspond to the University of California at Berkeley. You can get around this limitation of the geo search API by being specific enough in your location to get the result that you are looking for, in this example search for ”City of Berkeley.”

The output for the geo search example code for ”Flagstaff” looks like (id is shortened and the lines are chopped to fit page width):

```
Test geo search:
{"result":{"features":[{"id":"#9202a04641f800006e342", ...
Test geo search result:
{"features":[{"id":"#9202a04641f800006e342", ...
Test geo search features:
[{"id":"#9202a04641f800006e342","properties": ...
Test geo search first feature:
{"id":"#9202a04641f800006e342","properties":{"id": ...
Test geo search geometry:
{"id":"#9202a04641f800006e342","type":"Point", \\
"coordinates":[-111.6506,35.1981]}
Test geo search coordinates:
[-111.6506,35.1981]
```

While JSON is excellent for what it was designed to do (being a native Javascript data format) I personally find dealing with JSON in Java applications to be a nuisance. When writing applications using JSON, I start out as I did in this last code snippet by printing out a sample JSON payload and writing code snippets to pull it apart to get the information I need. For a given type of query, I then write a small wrapper to extract the information I need which is what I did in the following Java code snippets, starting with a local (non-public) utility class to contain latitude and longitude coordinates:

```
class LatLon {
```

```
  public double lat;
  public double lon;
  public LatLon(double lat, double lon) {
    this.lat = lat; this.lon = lon;
  }
  public String toString() {
    return "<Lat: " + lat + ", Lon: " + lon + ">";
  }
}
```

Then, I added a few methods to the example class *FreebaseToRdf*:

```
public class FreebaseToRdf {
  public FreebaseToRdf() {
    this.freebase = Freebase.getFreebase();
  }
  public JSON search(String query) {
    return search(query,
                  new HashMap<String,String>());
  }
  public JSON search(String query,
                     Map<String,String> options) {
    return freebase.search(query, options);
  }
  public LatLon geoSearchGetLatLon(String location) {
    return geoSearchGetLatLon(location,
                        new HashMap<String,String>());
  }
  public LatLon geoSearchGetLatLon(String location,
                                   Map<String,String> options) {
    JSON results =
      freebase.geosearch(location,
                new HashMap<String,String>());
    JSON coordinates =
         results.get("result").get("features").
              get(0).get("geometry").get("coordinates");
    return new LatLon(
        Double.parseDouble(""+coordinates.get(0)),
        Double.parseDouble(""+coordinates.get(1)));
  }

  public JSON geoSearch(String location) {
    return geoSearch(location,
                new HashMap<String,String>());
```

```
  }
  public JSON geoSearch(String location,
                        Map<String,String> options) {
    return freebase.geosearch(location,options);
  }

  private Freebase freebase;
}
```

If you want to pass search options, then add your key/value option values:

```
  Map<String,String> options =
      new HashMap<String,String>();
  options.put("limit", "5");
  JSON results = freebase.search("Java", options);
```

This wrapper library is really too simple because most of the APIs still return raw JSON. If you use this example code in your own applications then you will probably want to write custom extractors like *geoSearchGetLatLon()* to extract whatever specific data that your application needs from the raw JSON data.

13.3. Combining Web Site Scraping with Freebase

I am going to use the Java NLP utility class from Section 12.1.5 to find relevant search terms for Freebase. The basic idea is to scrape an arbitrary web page, use the KnowledgeBooks entity extraction library (or you could use Open Calais) to find names and places, and find more information about these names and places on Freebase. The trick is to find search terms in the original input text and add these terms to the Freebase query string. (Note: I will do the same using DBPedia in Chapter 14.)

The file *examples/WebScrapingAndFreebaseSearch.java* contains the example code for this section. The main function spiders a web site and loops on each fetched page. The processing steps for each page include: use the class *ExtractSearchTerms* to get relevant search terms, extract person and place names from the page, and call the utility method process for both people and places found on the web page:

```
static public void main(String[] args) throws Exception {
  PrintWriter out =
    new PrintWriter(new FileWriter("out.nt"));
  WebSpider ws = new WebSpider("http://markwatson.com", 2);
```

```
  for (List<String> ls : ws.url_content_lists) {
    String url =ls.get(0);
    String text = ls.get(1);
    // Get search terms for this web page's content:
    ExtractSearchTerms extractor =
        new ExtractSearchTerms(text);
    System.out.println("Best search terms " +
            extractor.getBest());
    // Get people and place names in this web
    // page's content:
    ScoredList[] ret =
        new ExtractNames().getProperNames(text);
    List<String> people = ret[0].getStrings();
    List<String> places = ret[1].getStrings();
    System.out.println("Human names: " + people);
    System.out.println("Place names: " + places);
    // Use Freebase to get more information about
    // these people and places:
    processPeople(out, url, text, "person", people,
                  extractor.getBest());
    processPlaces(out, url, "place", places);
  }
  out.close();
}
```

Most of the functionality seen here has already been implemented in my KnowledgeBooks NLP library. The new code for this example is in the utility method *process*. The trick that I use in *process* is fairly simple, given the goal of finding the correct Freebase article or data object corresponding to the human and place names contained in a web page. The class *ExtractSearchTerms* (see Section 12.1.5) auto-classifies the input text and keeps track of which words in the text provide the most evidence for assigning the categories. These words are the recommended search terms.

The problem is that for a typical Freebase article on a very similar topic, many of the search terms will not appear. The trick I use is in using the method *take* to choose (with some randomness) a subset of words in the list of recommended search terms. I start out by taking subsets almost as large as the set of recommended search terms. For each subset, I perform a Freebase search looking for a search match over a specified threshold. If I do not find a good match, I gradually reduce the size of the subset of extracted search terms until I get either a good match or give up after several iterations. Certainly, this method does not always find relevant Freebase articles or objects for a human or place, but it often does. For your own applications, you can experiment with the threshold value, decreasing it to get more results but more ”false positives” or increasing it to get fewer results but reducing the number of ”false positives.” You can take a look at the implementation of *take* in the Java source file. I use another utility

method *blankNodeURI* to assign URIs for blank nodes used in the generated RDF.

Here is some generated RDF that shows the use of blank nodes:

```
<http://www.knowledgebooks.com>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://knowledgebooks.com/rdf/webpage> .
<http://www.knowledgebooks.com>
  <http://knowledgebooks.com/rdf/contents>
  "Knowledgebooks.com: AI Technology for Knowledge ..." .
<http://www.knowledgebooks.com>
  <http://knowledgebooks.com/rdf/discusses/person>
  _:person_63659_10000 .

_:person_63659_10000
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type">
  <http://knowledgebooks.com/rdfs/entity/person> .

_:person_63659_10000
  <http://xmlns.com/foaf/0.97/name>
  "Mark Watson" .

_:person_63659_10000
  <http://knowledgebooks.com/rdf/freebase/id>
  "guid/9202a8c04000641f80000000146fb902" .

<http://www.knowledgebooks.com>
  <http://knowledgebooks.com/rdf/discusses/place>
  _:place_69793_10001 .
_:place_69793_10001
  <http://knowledgebooks.com/rdf/name/>
  "Flagstaff" .
_:place_58099_10001
  <http://knowledgebooks.com/rdf/location/>
  "-111.65+35.19"@http://knowledgebooks.com/rdf/latlon .
```

If you want to defreference the Freebase GUID seen in the last listing, append it to the base URI "http://www.freebase.com/view/" and if you want the RDF data then replace the first "/" character in the GUID with a "." append it to the URI "http://rdf.freebase.com/rdf/". For this example these two URIs to get the derefenced HTML and the RDF data for this GUID are:

```
http://www.freebase.com/view/guid/9202a8c04000641f8000 \\
0000146fb902
```

```
http://rdf.freebase.com/rdf/guid.9202a8c04000641f8000 \\
0000146fb902
```

13.4. Freebase Wrapup

Freebase is an excellent resource for augmenting information from other data sources. The overview in this chapter and the code examples should give you a good start in using Freebase in your own applications.

14. SPARQL Client Library for DBpedia

This Chapter will cover the development of a general purpose SPARQL client library and also the use of this library to access the DBpedia SPARQL endpoint.

DBpedia is a mostly automatic extraction of RDF data from Wikipedia using the metadata in Wikipedia articles. You have two alternatives for using DBpedia in your own applications: using the public DBpedia SPARQL endpoint web service or downloading all or part of the DBpedia RDF data and loading it into your own RDF data store (e.g., AllegroGraph or Sesame).

The public DBpedia SPARQL endpoint URI is http://dbpedia.org/sparql. For the purpose of the examples in this book we will simply use the public SPARQL endpoint but for serious applications I suggest that you run your own endpoint using the subset of DBpedia data that you need..

The public DBpedia SPARQL endpoint is run using the Virtuoso Universal Server (http://www.openlinksw.com/). If you want to run your own your own DBpedia SPARQL endpoint you can download the RDF data files from http://wiki.dbpedia.org and use the open source version of Virtuoso, Sesame, AllegroGraph, or any other RDF data store that supports SPARQL queries.

14.1. Interactively Querying DBpedia Using the Snorql Web Interface

When you start using DBpedia, a good starting point is the interactive web application that accepts SPARQL queries and returns results. The URL of this service is:

```
http://dbpedia.org/snorql
```

Figure 14.1 shows the DBpedia Snorql web interface showing the results of one of the sample SPARQL queries used in this section.

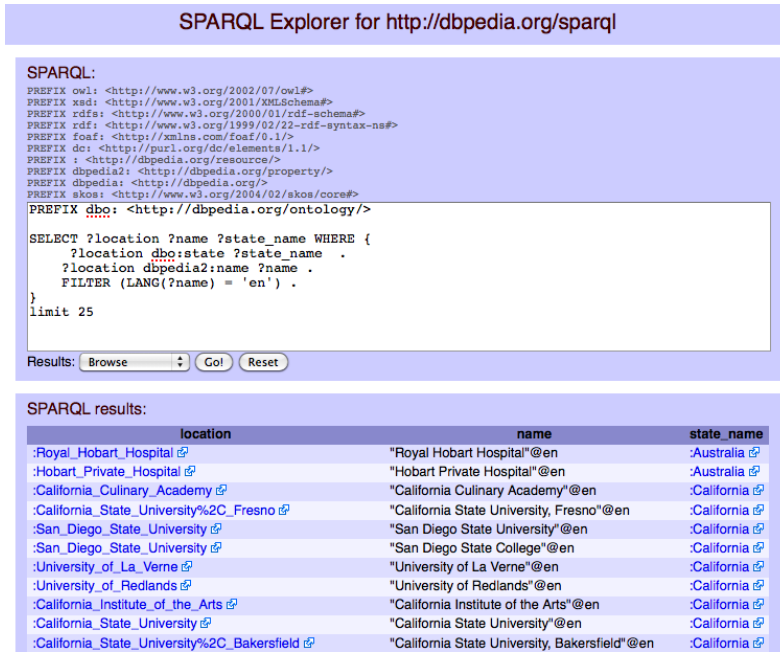

Figure 14.1.: DBpedia Snorql Web Interface

A good way to become familiar with the DBpedia ontologies used in these examples is to click the links for property names and resources returned as SPARQL query results, as seen in Figure 14.1. Here are three different sample queries that you can try:

```
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?s ?p WHERE {
     ?s ?p <http://dbpedia.org/resource/Berlin> .
}
ORDER BY ?name
```

```
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?s ?p WHERE {
     ?s dbo:state ?p  .
}
limit 25
```

```
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>
```

```
SELECT ?location ?name ?state_name WHERE {
       ?location dbo:state ?state_name  .
       ?location dbpedia2:name ?name .
       FILTER (LANG(?name) = 'en') .
}
limit 25
```

The http://dbpedia.org/snorql SPARQL endpoint web application is a great resource for interactively exploring the DBpedia RDF datastore. We will look at an alternative browser in the next section.

14.2. Interactively Finding Useful DBpedia Resources Using the gFacet Browser

The gFacet browser allows you to find RDF resources in DBpedia using a search engine. After finding matching resources you can then dig down by clicking on individual search results.

You can access the gFacet browser using this URL:

```
http://www.gfacet.org/dbpedia/
```

Figures 14.2 and 14.3 show a search example where I started by searching for "Arizona parks," found five matching resources, clicked the first match "Parks in Arizona," and then selected "Dead Horse State Park."[1]

14.3. The lookup.dbpedia.org Web Service

We will use Georgi Kobilarov's DBpedia lookup web service to perform free text search queries to find data in DBpedia using free text search. If you have a good idea of what you are searching for and know the commonly used DBpedia RDF properties then using the SPARQL endpoint is convenient. However, it is often simpler to just perform a keyword search and this is what we will use the lookup web service for. We will later see the implementation of a client library in Section 14.5. You can find documentation on the REST API at http://lookup.dbpedia.org/api/search.asmx?op=KeywordSearch. Here is an example URL for a REST query:

[1] This is a park near my home where I go kayaking and fishing.

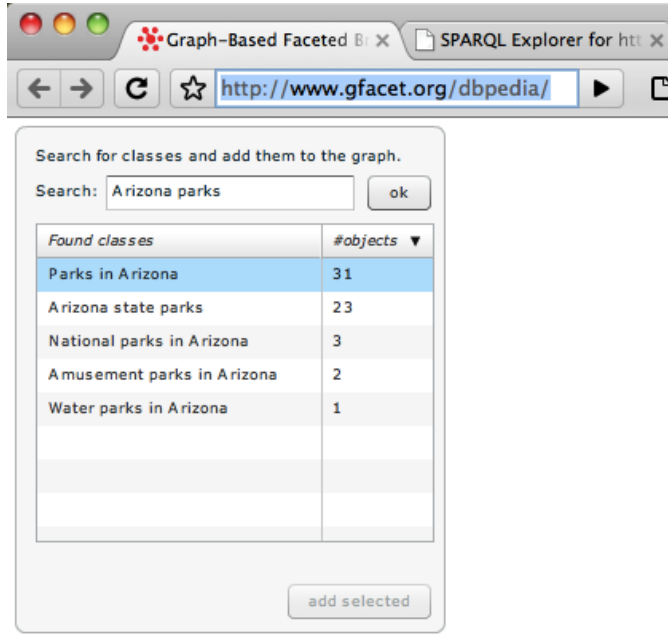

Figure 14.2.: DBpedia Graph Facet Viewer

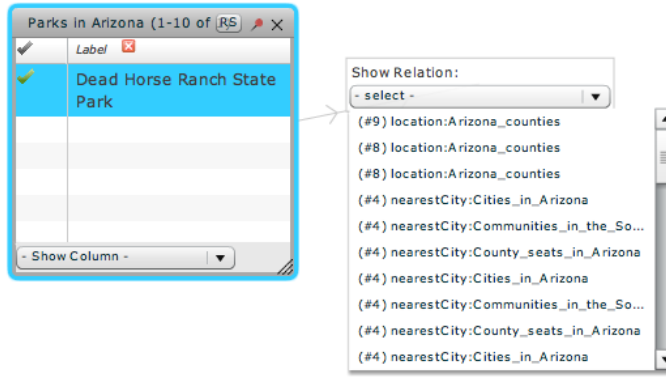

Figure 14.3.: DBpedia Graph Facet Viewer after selecting a resource

```
http://lookup.dbpedia.org/api/search.asmx/KeywordSearch? \\
QueryString=Flagstaff\&QueryClass=XML\&MaxHits=10
```

As you will see in Section 14.5, the search client needs to filter results returned from the lookup web service since the lookup service returns results with partial matches of search terms. I prefer to get only results that contain all search terms.

The following sections contain implementations of a SPARQL client and a free text search lookup client.

14.4. Implementing a Java SPARQL Client Library

There are three Java files in the software for this book that you can use for general SPARQL clients and specifically to access DBpedia:

1. src/com/knowledgebooks/rdf/SparqlClient.java is general purpose library for accessing SPARQL endpoint web services.
2. src/com/knowledgebooks/info_spiders/DBpediaLookupClient.java is a utility for accessing Georgi Kobilarov's DBpedia lookup web service.
3. examples/EntityToRdfHelpersDbpedia.java is developed in Chapter 16 and will be used to match entities in text to URIs in DBpedia.

The SPARQL endpoints that we will be using return XML data containing variable bindings for a SPARQL query. You can find the implementation in the file SparqlClient.java. The class *SparqlClient* extends the default XML Parsing SAX class DefaultHandler:

```
public class SparqlClient extends DefaultHandler {
```

I use the Apache Commons Library to set up and make an HTTP request to the endpoint and then pass the response input stream to a SAX parser::

```
public SparqlClient(String endpoint_URL, String sparql)
          throws Exception {
  HttpClient client = new HttpClient();
  client.getHttpConnectionManager().getParams().
    setConnectionTimeout(10000);

  String req = URLEncoder.encode(sparql, "utf-8");
  HttpMethod  method = new GetMethod(endpoint_URL +
```

```
  method.setFollowRedirects(false);
  try {
    client.executeMethod(method);
    InputStream ins = method.getResponseBodyAsStream();
    SAXParserFactory factory =
      SAXParserFactory.newInstance();
    SAXParser sax = factory.newSAXParser();
    sax.parse(ins, this);
  } catch (HttpException he) {
    System.err.println("Http error connecting to '" +
                   endpoint_URL + "'");
  } catch (IOException ioe) {
    System.err.println("Unable to connect to '" +
                  endpoint_URL + "'");
  }
  method.releaseConnection();
}
```

The SAX callback handlers use four member variables to record the state of variable bindings in the returned XML payload:

```
private List<Map<String, String>> variableBindings =
  new ArrayList<Map<String, String>>();
private Map<String, String> tempBinding = null;
private String tempVariableName = null;
private String lastElementName = null;
```

In the next section I'll show you a sample SPARQL query to find people who were born in California. I am going to jump ahead here and show you some of the returned XML data so the implementation of the SAX handler methods will be clear (notice that some identical variable bindings are returned - we will not keep duplicates):

```
 <head>
  <variable name="name"/>
  <variable name="birth"/>
  <variable name="person"/>
 </head>
 <results distinct="false" ordered="true">
  <result>
   <binding name="name">
     <literal>Busby Berkeley</literal>
   </binding>
```

```
   <binding name="person">
     <uri>http://dbpedia.org/resource/Busby_Berkeley</uri>
   </binding>
  </result>
  <result>
   <binding name="name">
     <literal>Busby Berkeley</literal>
   </binding>
   <binding name="person">
     <uri>http://dbpedia.org/resource/Busby_Berkeley</uri>
   </binding>
  </result>
  <result>
   <binding name="name">
     <literal>Emily Osment</literal>
   </binding>
   <binding name="person">
     <uri>http://dbpedia.org/resource/Emily_Osment</uri>
   </binding>
  </result>
```

The three SAX callback helpers are simple: I just record state of where I am in the XML tree and collect variable names and their bindings:

```
public void startElement(String uri,
                         String localName,
                         String qName,
                         Attributes attributes)
                            throws SAXException {
    if (qName.equalsIgnoreCase("result")) {
        tempBinding = new HashMap<String, String>();
    }
    if (qName.equalsIgnoreCase("binding")) {
        tempVariableName = attributes.getValue("name");
    }
    lastElementName = qName;
}

public void endElement(String uri,
                       String localName,
                       String qName)
                          throws SAXException {
    if (qName.equalsIgnoreCase("result")) {
        if (!variableBindings.contains(tempBinding))
```

```
            variableBindings.add(tempBinding);
    }
}

public void characters(char[] ch,
                       int start, int length
                          ) throws SAXException {
    String s = new String(ch, start, length).trim();
    if (s.length() > 0) {
        if ("literal".equals(lastElementName))
          tempBinding.put(tempVariableName, s);
        if ("uri".equals(lastElementName))
          tempBinding.put(tempVariableName, "<" + s + ">");
    }
}
```

The implementation of the SPARQL client was fairly simple. Most of the work was in parsing the returned XML response from the web service and extracting the information that we needed.

In the next few sections we will use this SPARQL library with Java, JRuby, Clojure, and Scala examples.

14.4.1. Testing the Java SPARQL Client Library

The *SparqlClient* constructor requires a SPARQL endpoint and a SPARQL query:

```
import com.knowledgebooks.rdf.SparqlClient;
import java.util.Map;

public class TestSparqlClient {
  public static void main(String[]args) throws Exception {
    String sparql =
"PREFIX foaf: <http://xmlns.com/foaf/0.1/>\n" +
"PREFIX dbpedia2: <http://dbpedia.org/property/>\n" +
"PREFIX dbpedia: <http://dbpedia.org/>\n" +
"SELECT ?name ?person WHERE {\n" +
"     ?person dbpedia2:birthPlace \n" +
"        <http://dbpedia.org/resource/California> .\n" +
"     ?person foaf:name ?name .\n" +
"}\n" +
"LIMIT 10\n";
     SparqlClient test =
```

```
        new SparqlClient("http://dbpedia.org/sparql",
                         sparql);
      for (Map<String, String> bindings :
           test.variableBindings()) {
        System.out.print("result:");
        for (String variableName : bindings.keySet()) {
          System.out.print("  " + variableName +
              ":" + bindings.get(variableName));
        }
        System.out.println();
      }
    }
}
```

The method *variableBindings* returns a *List<Map<String, String>>*. The maps contain two keys, ”person” and ”name,” as seen in the output from this example (lines have been split to fit page width):

```
result:
  person:<http://dbpedia.org/resource/Busby_Berkeley>
  name:Busby Berkeley
result:
  person:<http://dbpedia.org/resource/Emily_Osment>
  name:Emily Osment
 ...
```

14.4.2. JRuby Example Using the SPARQL Client Library

The file knowledgebooks.jar is created using the top level Makefile. If you make any changes to the Java code in the directory src then you need to rerun ”make” before using this JRuby wrapper src/sparql_client_ruby.rb:

```
require 'java'
(Dir.glob("lib/*.jar") +
 Dir.glob("lib/sesame-2.2.4/*.jar")).each do |fname|
  require fname
end
require "knowledgebooks.jar"

class SparqlClientRuby
  def self.query endpoint_UL, sparql
    proxy = com.knowledgebooks.rdf.SparqlClient.new(
```

```
                            endpoint_UL, sparql)
    proxy.variableBindings().collect do |var_binding|
      x = {}
      var_binding.key_set.each {|var| x[var] =
                                    var_binding[var]}
      x
    end
  end
end
```

Here is a short test program that uses this wrapper:

```
require 'src/sparql_client_ruby'
require 'pp'

sparql =
"""PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
SELECT ?name ?person WHERE {
     ?person dbpedia2:birthPlace
             <http://dbpedia.org/resource/California> .
     ?person foaf:name ?name .
}
LIMIT 10
"""

query_results =
   SparqlClientRuby.query("http://dbpedia.org/sparql", sparql)

pp query_results
```

This test program should be run from the top level directory:

```
$ jruby test/test_sparql_client_ruby.rb
[{"person"=>"<http://dbpedia.org/resource/Busby_Berkeley>",
  "name"=>"Busby Berkeley"},
 {"person"=>"<http://dbpedia.org/resource/Emily_Osment>",
  "name"=>"Emily Osment"},
 {"person"=>"<http://dbpedia.org/resource/Gordon_Moore>",
  "name"=>"Gordon Moore"},
  ...
]
```

14.4.3. Clojure Example Using the SPARQL Client Library

The Clojure wrapper in the file src/sparql_client_clojure.clj is short:

```
(ns sparql_client_clojure)

(import '(com.knowledgebooks.rdf SparqlClient))

(defn convert-to-map [vb]
  (into {} vb))

(defn sparql-query [endpoint_uri query]
  (seq (map convert-to-map
         (.variableBindings
            (new SparqlClient endpoint_uri query)))))
```

The function *convert-to-map* converts a Java HashMap containing Java Entry objects into a Clojure PersistentArrayMap containing Clojure MapEntry objects. The test program is in the file test/test-sparql-clojure.clj which I show here broken into two code snippets interspersed with the program output:

```
(use 'sparql_client_clojure)

(def sparql
"PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
SELECT ?name ?person WHERE {
     ?person dbpedia2:birthPlace
        <http://dbpedia.org/resource/California> .
     ?person foaf:name ?name .
}
LIMIT 10
")

(def results
  (sparql-query "http://dbpedia.org/sparql" sparql))

(println results)

(println (first results))

(println (class (first (first results))))
(println (class (first results)))
```

The purpose of the last two print statements is to show you the classes of the returned seq and elements in the seq (not all output is shown):

```
({person <http://dbpedia.org/resource/Busby_Berkeley>,
  name Busby Berkeley}
 {person <http://dbpedia.org/resource/Emily_Osment>,
  name Emily Osment}
 ...)
{person <http://dbpedia.org/resource/Busby_Berkeley>,
 name Busby Berkeley}
clojure.lang.MapEntry
clojure.lang.PersistentArrayMap
```

The next part of the file test-sparql-clojure.clj:

```
(println ((first results) "name"))
(println ((first results) "person"))

(doseq [result results]
  (println (str "Result:\n  person URI: " (result "person")
    "\n  person name: " (result "name") ".")))
```

produces this output (not all output is shown):

```
Busby Berkeley
<http://dbpedia.org/resource/Busby_Berkeley>
Result:
  person URI: <http://dbpedia.org/resource/Busby_Berkeley>
  person name: Busby Berkeley.
Result:
  person URI: <http://dbpedia.org/resource/Emily_Osment>
  person name: Emily Osment.
 ...
```

14.4.4. Scala Example Using the SPARQL Client Library

For a Scala example, I am directly calling the Java APIs from Scala instead of writing a separate wrapper:

```
import com.knowledgebooks.rdf.SparqlClient
```

```
object TestScalaSparqlClient {
  def main(args: Array[String]) {
    val sparql =
"""PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
SELECT ?name ?person WHERE {
     ?person dbpedia2:birthPlace
         <http://dbpedia.org/resource/California> .
     ?person foaf:name ?name .
}
LIMIT 10
"""
    val results =
      new SparqlClient("http://dbpedia.org/sparql", sparql)
    println(results.variableBindings)
  }
}
```

The output from this Scala code snippet is:

```
[{person=<http://dbpedia.org/resource/Busby_Berkeley>,
  name=Busby Berkeley},
 {person=<http://dbpedia.org/resource/Emily_Osment>,
  name=Emily Osment},
  ...
]
```

14.5. Implementing a Client for the lookup.dbpedia.org Web Service

The DBpedia lookup service[2] indexes the data stored in DBpedia and performs free text search. This service returns results that contain a subset of the search terms so I will filter the results discarding results that do not contain every search term.

The source file is com/knowledgebooks/info_spiders/DBpediaLookupClient. The lookup web service supports RSET style requests like:

```
http://lookup.dbpedia.org/api/search.asmx/KeywordSearch?\
QueryString=Flagstaff&QueryClass=XML&MaxHits=10
```

[2] Implemented by Georgi Kobilarov

You can look at the source file for details; here we will look at just the public APIs I implemented:

```
public class DBpediaLookupClient extends DefaultHandler {
  public DBpediaLookupClient(String query)
       throws Exception {
    ...
  }
  public List<Map<String, String>> variableBindings() {
    ...
  }
}
```

I extend the XML parsing *DefaultHandler* class to extract results from the XML data returned by the lookup web service by adding XML event handling methods *startElement*, *endElement*, and *characters*.

The file test/TestDBpediaLookupClient.java is an example using the class *DBpediaLookupClient*:

```
import com.knowledgebooks.info_spiders.DBpediaLookupClient;
import java.util.Map;

public class TestDBpediaLookupClient {
  public static void main(String[] args)
      throws Exception {
    DBpediaLookupClient lookup =
      new DBpediaLookupClient("City of Flagstaff Arizona");
    for (Map<String, String> bindings :
         lookup.variableBindings()) {
      System.out.println("result:");
      for (String variableName : bindings.keySet()) {
        System.out.println("  " + variableName +
              ":" + bindings.get(variableName));
      }
      System.out.println();
    }
  }
}
```

Here is the output:

```
result:
```

```
Description:Flagstaff is a city located in northern
  Arizona, in the southwestern United States. In
  July 2006, the city's estimated population was
  58,213. The population of the Metropolitan
  Statistical Area was estimated at 127,450 in 2007.
  It is the county seat of Coconino County. The city
  is named after a Ponderosa Pine flagpole made
  by a scouting party from Boston (known as the
  "Second Boston Party") to celebrate the United
  States Centennial on July 4, 1876.
Label:University towns in the United States
URI:http://dbpedia.org/resource/Category:\
       University_towns_in_the_United_States
```

14.6. DBpedia Wrap Up

Like Freebase, DBpedia is a good source of information contributed by many individuals and organizations. You have seen two techniques for finding information: using SPARQL queries and free text search. Because the information in DBpedia is extracted from Wikipedia it contains a wide variety of subjects but there is no guarantee that any information you find is accurate.

In the next Chapter we will look at another free source of data: the GeoNames database and web service.

15. Library for GeoNames

GeoNames (http://www.geonames.org/) is a geographic information database. The raw data is available under a Creative Commons Attribution license. There is a free web service and a commercial web service. For production environments you will want to use the commercial service but for development purposes and for the examples for this book I use the free service[1].

15.1. GeoNames Java Library

The creators and maintainers of the GeoNames project have a Java library that you can download as a JAR file. I include this JAR file in the *lib* directory and wrap their library with my utility class in the package *com.knowledgebooks.info_spiders* and the class name is *GeoNamesClient*. The package *com.knowledgeBooks.nlp.util* contains a data container class *GeoNamesData* that I use in my applications and in the book examples.

15.1.1. GeoNamesData

I created the data container class *GeoNamesData* to hold the data returned from GeoNames web service calls.

The following listing shows some of the implementation of this data container class. Note that I am not following the JavaBean conventions of private data with public get/set methods[2]. This allows more concise applications, with field name access as you can do in Ruby or Scala.

```
import org.geonames.Toponym;
```

[1] The geonames.org web service is limited to 2000 queries per hour from any single IP address. Commercial support is available, or, with some effort, you can also run GeoNames on your own server with some effort. There are, for example, a few open source Ruby on Rails projects that use the Geonames data files and provide a web service interface.

[2] This is not the best Java programming style but I sometimes do this for data-only classes. I think that sometimes simplicity is better than formalism for its own sake.

```
public class GeoNameData {
    public enum GeoType  {CITY, COUNTRY, STATE,
                          RIVER, MOUNTAIN, UNKNOWN};
    public int geoNameId = 0;
    public GeoType geoType = GeoType.UNKNOWN;
    public String name = "";
    public double latitude = 0;
    public double longitude = 0;
    public String countryCode = "";
    public GeoNameData(Toponym toponym) {
        geoNameId = toponym.getGeoNameId();
        latitude = toponym.getLatitude();
        longitude = toponym.getLongitude();
        name = toponym.getName();
        ...
    }
    public GeoNameData() { }
    public String toString() {
        return "[GeoNameData: " + name + ", type: " +
                  geoType + ... "]";
    }
}
```

The utility class in the next section fills instances of this class by making web service calls to GeoNames.

15.1.2. GeoNamesClient

Using the wrapper class *GeoNamesClient* you can get geographic information for cities, countries, states, rivers and mountains. The following listing shows part of the implementation of my wrapper for the standard Java GeoNames client library. You can refer to the source file for the rest of the implementation. The method *helper* makes the web service calls and is used by the individual methods to fetch city, country, etc. geographic information.

```
package com.knowledgebooks.info_spiders;

import com.knowledgebooks.nlp.util.GeoNameData;
import org.geonames.*;
import java.util.ArrayList;
import java.util.List;

public class GeoNamesClient {
```

```
    public GeoNamesClient() { }
    private List<GeoNameData>
        helper(String name, String type)
                        throws Exception {
        List<GeoNameData> ret =
           new ArrayList<GeoNameData>();
        ToponymSearchCriteria searchCriteria =
           new ToponymSearchCriteria();
        searchCriteria.setStyle(Style.LONG);
        searchCriteria.setQ(name);
        ToponymSearchResult searchResult =
              WebService.search(searchCriteria);
        for (Toponym toponym :
             searchResult.getToponyms()) {
          if (toponym.getFeatureClassName() != null &&
              toponym.getFeatureClassName(). \\
                 toString().indexOf(type) > -1 &&
              toponym.getName().indexOf(name) > -1 &&
              valid(toponym.getName())) {
            ret.add(new GeoNameData(toponym));
          }
        }
        return ret;
    }
    private boolean valid(String str) {
        ...
        return true;
    }
    public List<GeoNameData>
      getCityData(String city_name)
                        throws Exception {
        return helper(city_name, "city");
    }
    public List<GeoNameData>
      getCountryData(String country_name)
                           throws Exception {
        return helper(country_name, "country");
    }
    ...
}
```

15.1.3. Java Example Client

The client APIs are easy to use. In the following example, I am using the free GeoNames web service so I (politely) pause for two seconds between web service calls.

```
import com.knowledgebooks.info_spiders.GeoNamesClient;

public class TestGeoNamesClient {
  public static void main(String[] args) throws Exception {
    GeoNamesClient test = new GeoNamesClient();
    System.out.println(test.getCityData("Paris"));
    pause();
    System.out.println(test.getCountryData("Canada"));
    pause();
    System.out.println(test.getStateData("California"));
    pause();
    System.out.println(test.getRiverData("Amazon"));
    pause();
    System.out.println(test.getMountainData("Whitney"));
  }
  private static void pause() {
    try { Thread.sleep(2000);
    } catch (Exception ignore) { }
  }
}
```

Here is a small part of the output from the test program:

```
[[GeoNameData: Paris, type: CITY, country code: FR,
               ID: 2988507, latitude: 48.85341,
               longitude: 2.3488],
 [GeoNameData: Paris, type: CITY, country code: CA,
               ID: 6942553, latitude: 43.2,
               longitude: -80.383333],
 ...]
[[GeoNameData: Canada, type: COUNTRY, country code: CA,
               ID: 6251999, latitude: 60.0,
               longitude: -96.0]]
...
```

15.2. GeoNames Wrap Up

The GeoNames web service is a great source of information on cities, countries, rivers, mountains, etc. We have finished studying linked data and learning how to access them. In the next part of this book we will look at an example application using almost everything covered so far.

16. Generating RDF by Combining Public and Private Data Sources

We will use most of what you have learned in this book in the example application developed in this chapter. A major theme has been using RDF data stores instead of alternatives like relational databases or document data stores like MongoDB and CouchDB. For applications that might be better suited to a document style data store, you can still use the ideas and most of the code in this chapter but you would lose the flexibility of being able to use RDF data from different sources and using different namespaces and schemas.

The example application in this chapter produces an RDF data file containing information from scraped web pages and linked data references from Freebase, DBpedia, and a local relational database.[1]

16.1. Motivation for Automatically Generating RDF

Your company or the organization that you work for has private data sources that can be data mined for information. Using public data sources to increase the value of your private data is really one of the main purposes of this book. That said, I can't very well use your data in a book example so you will have to use some imagination to extend the ideas presented here to your own business. We will use a simple sample database and the relational database to RDF tool D2R that I cover in Appendices A and B where I show you how to set up the D2R Server that wraps existing databases and provides a SPARQL endpoint for read-only access.

The sample application written for this chapter is "hand crafted" in the sense that I am looking for specific types of Semantic information and there is custom code to handle this information. This is a practical approach that is much simpler than trying

[1] For this example, I did not generate RDF using data from GeoNames because I am already getting geographic data from Freebase. If you don't need access to Freebase in your applications, then using GeoNames is the easiest way to get geographic information.

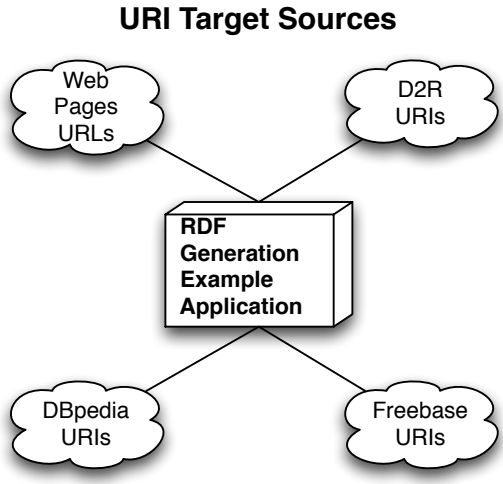

Figure 16.1.: Data Sources used in this example application

to write something that is very general. Indeed, general systems may require artificial intelligence that is currently beyond the state-of-the-art. So, I suggest that you choose types of information that are important for your business, find external data sources that can increase the value of your data, and then iterate by adding more information sources and custom processing to generate additional RDF statements as needed.

In addition to augmenting your proprietary data by merging it with public linked data sources, there can be advantages to selectively publishing some of your own data via both SPARQL endpoint and web interfaces. This chapter is about merging your own data with public data, thereby increasing the value of both.

Generated RDF data with ancillary public RDF sources can be used, for example, in a web application much as you would use the data in a relational database.[2][3]

In the sample application developed in this chapter I am going to use a small test relational database (see Appendix B) and a list of external reference web sites. Building on previous book code examples, I will use Freebase and DBpedia to augment data in our own relational database and the reference web sites as seen in Figure 16.1. The generated output will be a RDF graph that spans information in all of these data sources.

[2] The documentation for AllegroGraph contains a sample Ruby on Rails example that uses the AllegroGraph RDF data store instead of a relational database.

[3] One of my customers uses RDF data instead of a relational database. Also, my KBSportal.com project uses and RDF data store, a relational database, and MongoDB.

Generating RDF data makes something simple that would be more difficult using a database. We are looking for relationships between content in different data sources and we desire an understanding of these relationships. As an example, we will generate RDF that maps relationships between specific people's names, place names, organizations, etc. to information sources like database rows, web pages, etc. With RDF, we then get "for free" the ability to analyze the reverse relationships using SPARQL queries.[4]

With RDF we can build on both our own RDF data stores and third party data stores, later add more data sources and use new properties (or relationships) between data to extend our application. My vision for the Semantic Web is a flexible framework of linked data that can be used and extended by adding data sources, new RDFS properties, and new Ontologies. The result is a very agile development process that continually builds on what is already available.

16.2. Algorithms used in Example Application

The example program in this chapter uses two meta-data information sources: a list of available databases and a list of web sites to spider. We will use the simple test database from Appendix A and a list of my web sites as example input data.

The input to this example application is a text file with one web URL per line followed by the maximum number of pages to retrieve for the URL's domain and the example relational database that I set up in Appendix B. Here is the first sample configuration file that I used while developing the example application in this chapter:

```
http://www.knowledgebooks.com 2
http://markwatson.com 2
```

Here is the second configuration file (the second line in this file is continued on three additional lines to fit the page width):

```
localhost 2020
customers vocab:customers_city/place     \\
          vocab:customers_state/place    \\
          vocab:customers_name/person    \\
          vocab:customers_name/organization
```

[4] This is a general benefit of graph databases that is shared with more general graph data stores like Neo4j and in-memory graph data structures. A key benefit of RDF data stores is the ability to perform SPARQL queries, eliminating the need for custom coding.

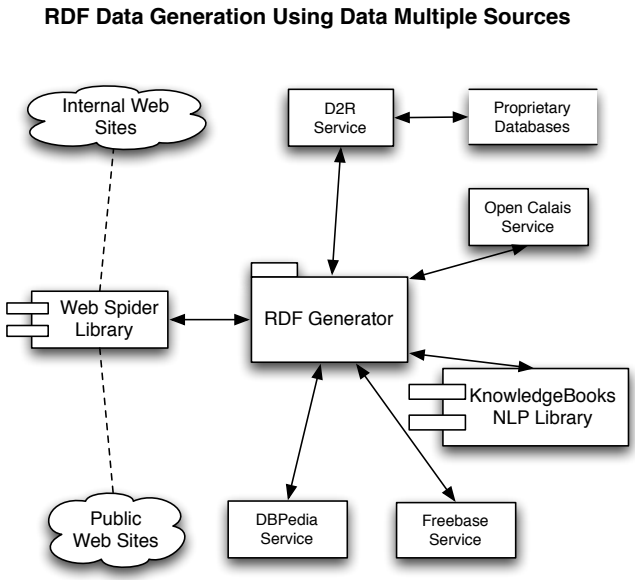

Figure 16.2.: Architecture for RDF generation from multiple data sources

The first line contains the D2R server host name or IP address followed by the port number for using the SPARQL endpoint.

We want our application to read this configuration file, perform web spidering using the utilities from Chapter 10. The text for each page is processed using the Open Calais web services and generates RDF for identifying properties on each single web page as seen in this example:

```
<http://www.knowledgebooks.com>
    <http://knowledgebooks.com/rdf/ProgrammingLanguage>
    "Common Lisp" .
<http://www.knowledgebooks.com>
    <http://knowledgebooks.com/rdf/ProgrammingLanguage>
    "Java" .
<http://www.knowledgebooks.com>
    <http://knowledgebooks.com/rdf/Technology>
    "AI Technology" .
<http://www.knowledgebooks.com>
    <http://knowledgebooks.com/rdf/Technology>
    "Knowledge Management" .
<http://www.knowledgebooks.com>
```

```
    <http://knowledgebooks.com/rdf/Technology>
    "Natural Language Processing" .
```

We also generate RDF containing information on how semantically similar web pages are as seen in this example:

```
<http://markwatson.com>
   <http://knowledgebooks.com/rdf/medium_similarity>
   <http://www.knowledgebooks.com> .
```

16.3. Implementation of the Java Application for Generating RDF from a Set of Web Sites

If you want to have the code to look at while reading through this chapter, the source code of the main class for this application is in the file examples/RdfDataGenerationApplication.java[5] if you want to have the code to look at while reading through this chapter.[6]

As seen in Figure 16.3 the main application class RdfDataGenerationApplication spiders a list of target web sites and then uses the three helper classes EntityToRdfHelpersFreebase, EntityToRdfHelpersDbpedia, and EntityToD2RHelpers to find auxiliary data on Freebase, DBpedia, and a local database wrapped with D2R. Most of the complexity in this application is in these three helper classes that are similar to the code we used before in Chapters and 13 and 14.

16.3.1. Main application class RdfDataGenerationApplication

The constructor for this class reads both configuration files, and calls the method *spider* on each URL in the configuration file. As we will see shortly, this method performs the work of processing each web page and using the web page contents to find links in other data sources like Freebase, DBpedia, and the relational database (wrapped using D2R as a SPARQL endpoint).

The method *process_interpage_shared_properties* creates RDF statements relating the data sources already processed. The main application class discussed in this section

[5] This example uses code from previous examples, most notably in OpenCalaisGenerateRdfPropertiesFromWebPages.java. I made no attempt to factor out bits of common code since I expect that you will pull out code that you need into your applications (within limits of my commercial software license or the AGPL).

[6] Recommended!

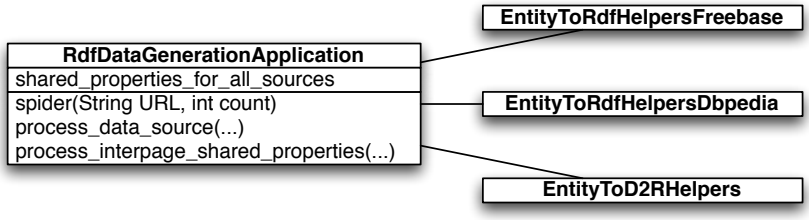

Figure 16.3.: The main application class RdfDataGenerationApplication with three helper classes

uses the three utility classes seen in Figure 16.3. We will look at these classes in the next three sections.

```
public RdfDataGenerationApplication(
             String web_sites_config_file,
             String database_config_file,
             PrintWriter out) throws Exception {
  this.out = out;
  this.database_config_file = database_config_file;
  // process web sites:
  List<String> lines =
       (List<String>)FileUtils.readLines(
                     new File(web_sites_config_file));
  for (String line : lines) {
    Scanner scanner = new Scanner(line);
    scanner.useDelimiter(" ");
    try {
      String starting_url = scanner.next();
      int spider_depth =
                     Integer.parseInt(scanner.next());
      spider(starting_url, spider_depth);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
  // after processing all 4 data sources, add more RDF
  // statements for inter-source properties:
  process_interpage_shared_properties();
  out.close();
}
```

The method *spider* uses the utility class *WebSpider* that I developed in Chapter 10 to get the text for the specified web pages. The input argument *spider_depth* controls how many links are processed. The hash map *for_shared_properties* is used to record properties on web pages between different calls to the method *process_data_source*.

```
private void spider(String starting_url, int spider_depth)
             throws Exception {
  WebSpider ws =
       new WebSpider(starting_url, spider_depth);
  Map<String, Set<String>> for_shared_properties =
       new HashMap<String, Set<String>>();
  for (List<String> ls : ws.url_content_lists) {
    String url = ls.get(0);
    String text = ls.get(1);
    process_data_source(url, text, for_shared_properties);
  }
}
```

The first part of method *process_data_source* uses the Open Calais client that I wrote in Chapter 11[7] to extract entity names in input text and create RDF statements inking these entities to the original source URIs.

```
private void process_data_source(String uri,
             String text,
             Map<String,
             Set<String>> for_shared_properties)
                 throws Exception {
  // OpenCalais:
  Map<String, List<String>> results =
      new OpenCalaisClient().
         getPropertyNamesAndValues(text);
  out.println("<" + uri + ""> " +
    "<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> \\
    <http://knowledgebooks.com/rdf/webpage> .");
  out.println("<" + uri + "> " +
    "<http://knowledgebooks.com/rdf/contents> \"" +
    text.replaceAll("\"", "'") + "\" .");
  if (results.get("Person") != null) {
    for (String person : results.get("Person")) {
      out.println("<" + uri + "> " +
        "<http://knowledgebooks.com/rdf/containsPerson>"
        + " \"" + person.replaceAll("\"", "'") + "\" .");
```

[7] You could substitute my NLP library from Chapter 12

```
    }
  }
  for (String key : results.keySet()) {
    for (Object val : results.get(key)) {
      if (("" + val).length() > 0) {
        String property =
          "<http://knowledgebooks.com/rdf/" + key + ">";
        out.println("<" + uri +
          "> <http://knowledgebooks.com/rdf/" + key +
          "> \"" + val + "\" .");
        HashSet<String> hs =
          (HashSet<String>)for_shared_properties.
                   get(property);
        if (hs == null) hs = new HashSet<String>();
        hs.add("\"" + val + "\"");
        for_shared_properties.put(
           "<http://knowledgebooks.com/rdf/" +
           key + ">", hs);
      }
    }
  }
```

The next part of method *process_data_source* uses the search term extractor from my NLP library from Chapter 12 to determine reasonable search terms for the input text. The goal is to determine search terms that would, if entered into a search engine, lead to the original content. This is useful for associating data in external sources like Freebase to entities found in the input text. As an example, if you looked up my name "Mark Watson" on Freebase you would be likely to find articles about me and many other people with my name. However, if you added relevant search terms from my web site like "Java," "artificial intelligence," etc. then you would be very likely to find information about me rather than someone else with my name.

```
// Find search terms in text:
ExtractSearchTerms extractor =
  new ExtractSearchTerms(text);
System.out.println("Best search terms " +
     extractor.getBest());
// Get people and place names in this
// web page's content:
ScoredList[] ret =
  new ExtractNames().getProperNames(text);
List<String> people = ret[0].getStrings();
List<String> places = ret[1].getStrings();
System.out.println("Human names: " + people);
```

```
    System.out.println("Place names: " + places);
```

Now we will use the helper class *EntityToRdfHelpersFreebase* to find (hopefully) relevant Freebase articles that we can link to the original URI:

```
    // Freebase:
    EntityToRdfHelpersFreebase.processPeople(
      out, uri, text, "person", people,
      extractor.getBest());
    EntityToRdfHelpersFreebase.processPlaces(
      out, uri, "place", places);
```

We will now use the same entity names and search terms to (hopefully) find relevant information on DBpedia using the helper class *EntityToRdfHelpersDbpedia*:

```
    // DBpedia:
    EntityToRdfHelpersDbpedia.processEntity(
      out, uri, "person", people, extractor.getBest(),
      processed_DBpedia_queries);
    EntityToRdfHelpersDbpedia.processEntity(
      out, uri, "place", places, extractor.getBest(),
      processed_DBpedia_queries);
```

We now use the utility class *EntityToD2RHelpers* to find data in a relational database that (might) pertain to the entities that we found on the original web pages:

```
    // process databases with D2R SPARQL endpoint
    // front ends:
    new EntityToD2RHelpers(
      uri, database_config_file, people, places, out);
```

Finally, we want to store shared property links for the current web page in the class data map *shared_properties_for_all_sources* that will be used by the method *process_interpage_shared_properties* that is called by the class constructor:

```
    shared_properties_for_all_sources.put(
      uri, for_shared_properties);
  }
```

The method *process_interpage_shared_properties* uses shared properties between web page URIs to determine if two web pages are considered to be similar enough to make an RDF statement about their similarity:

```
  private void process_interpage_shared_properties()
         throws Exception {
    Set<String> unique_urls =
      shared_properties_for_all_sources.keySet();
    for (String url_1 : unique_urls) {
      for (String url_2 : unique_urls) {
        if (url_1.equals(url_2) == false) {
          float url_similarity =
            score_mapset(
              shared_properties_for_all_sources.get(url_1),
              shared_properties_for_all_sources.get(url_2));
          if (url_similarity > 12f) {
            out.println("<" + url_1 + "> " +
          "<http://knowledgebooks.com/rdf/high_similarity>"
              + " <" + url_2 + "> .");
          } else if (url_similarity > 8f) {
            out.println("<" + url_1 + "> " +
          "<http://knowledgebooks.com/rdf/medium_similarity>"
              + "<" + url_2 + "> .");
          } else if (url_similarity > 5f) {
            out.println("<" + url_1 + "> " +
          "<http://knowledgebooks.com/rdf/low_similarity>"
              + "<" + url_2 + "> .");
          }
        }
      }
    }
  }
```

The method *score_mapset* simply compares two map sets based on the number of common keys:

```
  private float score_mapset(
    Map<String, Set<String>> set_1,
    Map<String, Set<String>> set_2) {
    float ret = 0f;
    for (String property_1 : set_1.keySet()) {
      Set<String> s1 = set_1.get(property_1);
      Set<String> s2 = set_2.get(property_1);
      if (s2 != null) {
        ret += score_sets(s1, s2);
      }
    }
    return ret;
```

```
  }

  private float score_sets(Set<String> l_1, Set<String> l_2) {
    float ret = 0f;
    for (String s : l_1) {
      if (l_2.contains(s)) ret += 1f;
    }
    return ret;
  }
```

We will look briefly at the implementation of the three helper classes seen in Figure 16.3 in the next three sections.[8]

16.3.2. Utility class EntityToRdfHelpersFreebase

The file examples/EntityToRdfHelpersFreebase.java is the implementation of a utility for matching lists of peoples names and place names with Freebase articles. A list of possible search terms for the original web page contents is used to minimize the number of "false positive" matches. Using my example web site knowledgebooks.com, a list of search terms like "semantic web" can help identify my name in Freebase, discarding articles that mention other people with my name.

This implementation is similar to what I showed you in Chapter 13. You can look at the source code to see the following processing steps:

1. Create an instance of the utility class *Freebase*.
2. Randomly choose a subset of the search terms for the original web page and search for articles on Freebase with either a human or place name that also contains a subset of these search terms.
3. Link any Freebase articles that contain both an entity name from the web page and the random subset of search terms.

When I originally implemented this class I used all available search terms and rarely found matching articles. However, by choosing random subsets of the search terms I would often find relevant material on Freebase to link to. I iteratively decrease the number of search terms in different random subsets and stop when I find either a relevant article or when the number of search terms to be chosen gets below a threshold value of three search terms.

[8] You should refer to the source code files since the following code descriptions will be a brief overview.

16.3.3. Utility class EntityToRdfHelpersDbpedia

The file examples/EntityToRdfHelpersDbpedia.java is the implementation of a utility for matching lists of peoples names and place names with data in DBpedia. A list of possible search terms for the original web page contents is used to minimize the number of ”false positive” matches. Using my example web site knowledgebooks.com, a list of search terms like ”semantic web” can help identify my name in DBpedia, discarding data that mentions other people with my name.

This implementation is similar to what I showed you in Chapter 14. You can look at the source code to see the following processing steps:

1. Use an instance of the utility class *DBpediaLookupClient*
2. Randomly choose a subset of the search terms for the original web page and search for articles on DBpedia with either a human or place name that also contains a subset of these search terms.
3. Link any DBpedia URIs that contain both an entity name from the web page and the random subset of search terms.

16.3.4. Utility class EntityToD2RHelpers

The file examples/EntityToD2RHelpers.java is the implementation of a utility for matching lists of peoples names and place names with data in a relational database. Unlike the classes in the last two sections, I do not use a list of possible search terms for the original web page contents is used to minimize the number of ”false positive” matches. If the entity type is of type person then I match against entity types defined in the database configuration file; for example, consider this entry in the test configuration file:

```
customers
  vocab:customers_city/place          \\
  vocab:customers_state/place         \\
  vocab:customers_name/person         \\
  vocab:customers_name/organization   \\
```

Here I am specifying that the column *customers_name* refers to type person so I would search the data in this column to match ”people entities” found by either Open Calais or my NLP library. Similarly, *customers_city* is of entity type ”place” so I will construct a SPARQL query like the following with ”place” replaced with the place name determined during entity extraction form the original web page and ”property_and_entity_type[0]” replaced by ”vocab:customers_city”:

```
PREFIX vocab: <http://localhost:2020/vocab/resource/>
SELECT ?subject ?name WHERE {
     ?subject  property_and_entity_type[0] ?name
     FILTER regex(?name, place) .
}
LIMIT 10
```

The implementation of this class is simple because all we need to do is to match the types of entities found on the original web page with the type ”person or ”place” and get the property name from the database config file. Since it is generally useful making SPARQL queries to data sources I am going to list the part of this class that processes ”place” type entities (you can read the source code for the similar code for processing ”person” type entities):

```
public class EntityToD2RHelpers {
  public EntityToD2RHelpers(
      String uri, String config_file,
      List<String> people, List<String> places,
      PrintWriter out)   throws Exception {
    // In this example, I am assuming that the D2R
    // server is running on localhost:2020
    List<String> lines =
      (List<String>) FileUtils.readLines(
         new File(config_file));
    String d2r_host_and_port = lines.remove(0);
    String [] info = d2r_host_and_port.split(" ");
    for (String line : lines) {
      Scanner scanner = new Scanner(line);
      scanner.useDelimiter(" ");
      String d2r_type = scanner.next();
      while (scanner.hasNext()) {
        String term = scanner.next();
        String [] property_and_entity_type = term.split("/");

        if (property_and_entity_type[1].equals("place")) {
          for (String place : places) {
            // perform SPARQL queries to D2R server:
            String sparql =
"PREFIX vocab: <http://localhost:2020/vocab/resource/>\n" +
"SELECT ?subject ?name WHERE {\n" +
"     ?subject " + property_and_entity_type[0] + " ?name \n" +
" FILTER regex(?name, \"" + place + "\") .\n" +
"}\n" +
"LIMIT 10\n";
```

```
            SparqlClient test =
              new SparqlClient(
                "http://localhost:2020/sparql", sparql);
            for (Map<String, String> bindings :
                 test.variableBindings()) {
              System.out.print("D2R result:" + bindings);
              if (bindings.keySet().size() > 0) {
                String blank_node = blankNodeURI("place");
                out.println(blank_node + " " +
             "<http://knowledgebooks.com/rdf/placeName> \""
                   + place.replaceAll("\"", "'") + "\" .");
                out.println("<" + uri + "> " +
           "<http://knowledgebooks.com/rdf/containsPlace> "
                   + blank_node + " .");
                out.println(blank_node +
            " <http://knowledgebooks.com/rdf/d2r_uri> \"" +
                   bindings.get("subject") + "\" .");
              }
            }
          }
        }
      }
    }
    out.close();
  }
  private static String blankNodeURI(String name_type) {
    return "_:dr" + name_type + blank_node_count++;
  }
  static long blank_node_count = 1;
}
```

After running this example application we get an RDF text file that we can load into Gruff for viewing or load into an RDF data store like AllegroGraph or Sesame for performing SPARQL queries.

I showed you in Chapter 11 an example of writing generated DRF data to a text file or calling the AllegroGraph or Sesame wrappers to add RDF statements programatically. The example in this chapter writes RDF text files but can be trivially modified to add RDF triples programmatically to an RDF datastore.[9]

Figure 16.4 is similar to Figure 12.1 and shows additional data sources from Freebase, DBpedia, and a local database with a D2R SPARQL endpoint wrapper.

[9] I leave this as an exercise for the reader.

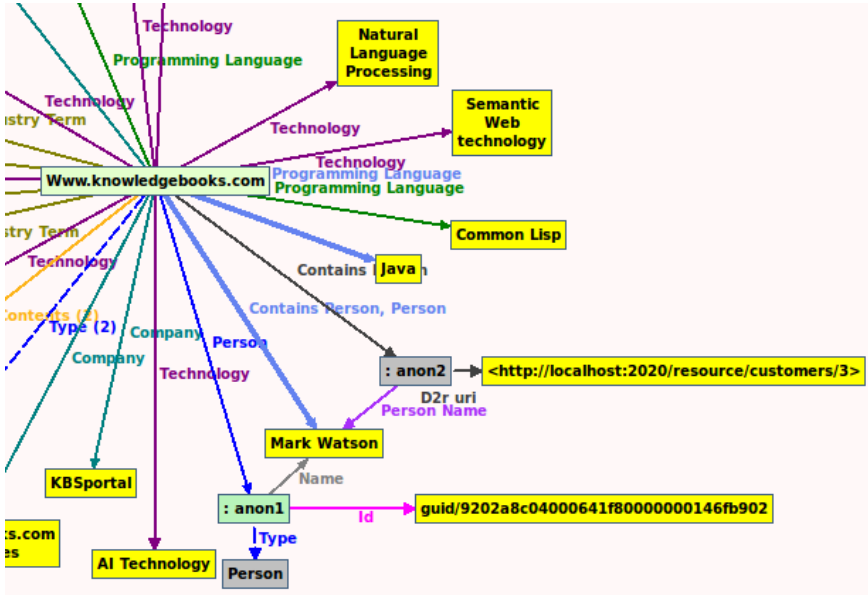

Figure 16.4.: Viewing generated RDF using Gruff

16.4. Sample SPARQL Queries Using Generated RDF Data

I am using the Franz AGWebView product[10] for the examples in this section. AGWebView should be started setting an administrator account so that you can create a new local repository and add the gen_rdf.nt RDF triple file:

```
./agwebview --account me:pass
```

If you visit http://localhost:8080 and login with account ”me” and password ”pass” then create a new repository and load the file gen_rdf.nt. Figures 16.5 and 16.6 show AGWebView, first with a SPARQL query and then showing the results of clicking on the blank node *_:bE8ADA5B4x2*.

[10] AGWebView is included with the AllegroGraph v4.0 server installation. See http://www.franz.com/agraph/agwebview for more information.

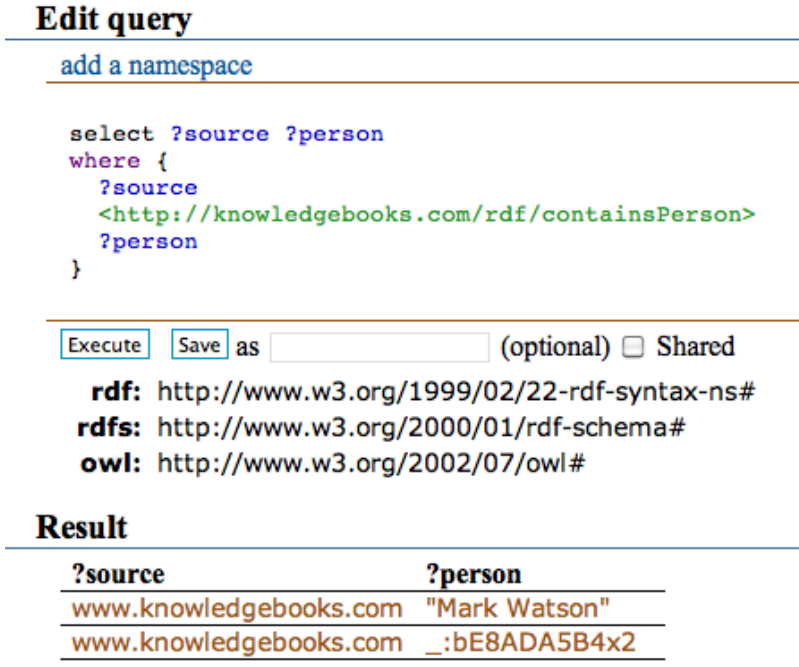

Figure 16.5.: Viewing generated RDF using AGWebView

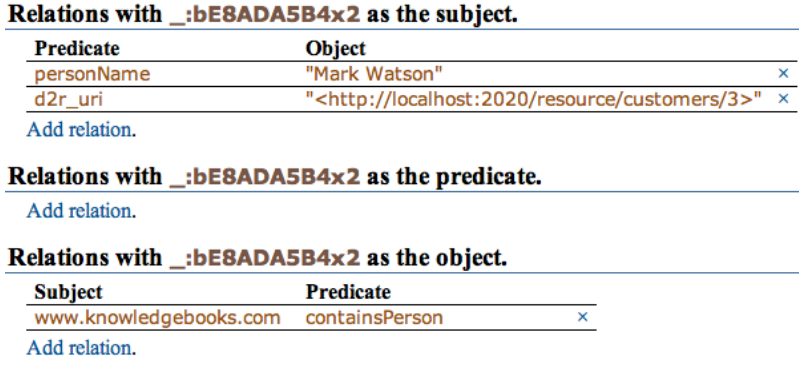

Figure 16.6.: Browsing the blank node *_:bE8ADA5B4x2*

The following listing shows the query seen in Figure 16.5 to find all web page URLs that contain a person's name:

```
select ?source ?person
where {
   ?source
   <http://knowledgebooks.com/rdf/containsPerson>
   ?person
}
```

The SPARQL query results are:

```
?source                 ?person
--------                 -------
www.knowledgebooks.com "Mark Watson"
www.knowledgebooks.com _:bE8ADA5B4x2
```

Notice in these results that one of the matching "names" is a blank node that is being browsed in Figure 16.6. For the rest of this section, I will show you a few more example queries that you can try yourself in AGWebView. Here I am looking for all web page URLs that discuss the industry term "Web technologies:"

```
select ?source
where {
  ?source
  <http://knowledgebooks.com/rdf/IndustryTerm>
  "Web technologies" .
}
```

The following query finds the URLs and names for all companies and the technologies that are mentioned on the each company's web site:

```
select ?company_url ?company_name ?technology
where {
   ?company_url
     <http://knowledgebooks.com/rdf/Company>
     ?company_name .
   ?company_url
     <http://knowledgebooks.com/rdf/Technology>
     ?technology .
}?
```

The following query finds all company URLs, company names, names of people working for the company, and optionally the Freebase GUID for people.

```
select ?company_url ?company_name
       ?person_name ?person_freebase_id
where {
   ?company_url
     <http://knowledgebooks.com/rdf/Company>
     ?company_name .
   ?company_url
     <http://knowledgebooks.com/rdf/discusses/person>
     ?person .
   ?person
     <http://xmlns.com/foaf/0.97/name>
     ?person_name .
   OPTIONAL {
     ?person
       <http://knowledgebooks.com/rdf/freebase/id>
       ?person_freebase_id .
   }
}?
```

16.5. RDF Generation Wrapup

Information is only useful if you can find it. Relational databases are fine for locating information in structured tables. However, even if you can effectively use a local data source you still get more use out of it if you can find relationships between your local data and external data sources. The example in this chapter provides you with a good starting point for linking your data to external sources by making RDF statements that relate data from different sources.

17. Wrapup

As we have explored in this book, RDF and RDFS are notations for making statements about sources of information. RDF data stores usually provide some form of logical inference (RDFS, RDFS++, and perhaps OWL) that allows us to further make statements about existing RDF statements. This is the secret sauce for using data from multiple sources without having to perform data conversion.

I have chosen to use two good software stacks in this book: AllegroGraph and Sesame. While there are good alternatives, learning how to use either or both AllegroGraph and Sesame should be sufficient for you to start experimenting with Semantic Web and Linked Data technologies, gradually integrating them into your work as you become comfortable using them.

Information collection and processing has always been an interest of mine, dealing with information in free text, semi-structured data like the HTML on web pages, and more structured information in relational databases. I hope that you enjoyed my personal take on these topics.

My expectation is that we will see a network effect benefit organizations that use Semantic Web technologies. A FAX machine is not valuable if there are only a few in the world. Web sites would lose much of their interest and practical value if they seldom contained links to other web sites. In a similar fashion Lined Data sources become more useful as they link to other data sources.

A. A Sample Relational Database

The file data/business.sql contains the SQL data to create three tables and enter some test data. This database is used in the example in Chapter 16.

Tables A.1, A.2 and A.3 show the column names and sample data for each table. We will use these tables to set up D2R in Appendix appendix:d2r and for the example program in Chapter 16.

On OS X or Linux you can create the database and populate it with test data using:

```
createdb -U postgres business_test
psql -U postgres business_test < data/business.sql
```

This test database has a simplified schema that very loosely models customers, products, and orders that connect orders with customers. Here is a listing of data/business.sql that is I used with PosgreSQL[1] specific:

```
create table customers (id int PRIMARY KEY,
     name char(30), city char(30), state char(30),
     country char(30));

create table products (id int PRIMARY KEY,
     name char(40), price float);

create table orders (id int PRIMARY KEY,
    product_id int REFERENCES products (id),
    customer_id int REFERENCES customers (id),
    number int);

insert into customers values (1, 'IBM', 'Armonk',
                              'New York', 'USA');
insert into customers values (2, 'Oracle',
                              'Redwood Shores',
                              'California', 'USA');
insert into customers values (3, 'Mark Watson',
```

[1] If you are using MySQL, this should also work fine for you.

Table A.1.: Customers Table

id	name	city	state	country
1	IBM	Armonk	New York	USA
2	Oracle	Redwood Shores	California	USA

Table A.2.: Products Table

id	name	price
1	KnowledgeBooks NLP Library	500.0
2	Spider Library	20.0

```
                              'Sedona',
                              'Arizona', 'USA');

insert into products values (1,
    'KnowledgeBooks NLP Library', 500.0);
insert into products values (2, 'Spider Library', 20.0);

insert into orders values (1, 1, 1, 1);
insert into orders values (2, 2, 1, 2);
insert into orders values (3, 1, 2, 4);
```

Table A.3.: Orders Table

id	product_id	customer_id	number
1	1	1	1
2	2	1	2
3	1	2	4

B. Using the D2R Server to Provide a SPARQL Endpoint for Relational Databases

D2R was written by Chris Bizer (who also started the DBpedia project) and Richard Cyganiak to add a SPARQL endpoint to existing relational databases. You can download the D2R server from http://www4.wiwiss.fu-berlin.de/bizer/d2rq/ and follow the installation instructions.

D2R can be used in one of two modes:

- Let D2R create the mapping for your database tables automatically based on table and column names.
- Create your own table and column names mappings to RDF predicates.

The first method is much simpler so that is where you should should start. Besides simplicity of setting up D2R with a database using the first method, you may find the RDF schema to be easier to use because you are already familiar with the table and column names.

B.1. Installing and Setting Up D2R

Download the latest version of D2R and un-tar it to a convenient location. If you are using MySQL or PostgreSQL there is no further installation required. If you are using another database then you need to put a JDBC driver JAR file in your D2R installation's *lib* directory.

B.2. Example Use of D2R with a Sample Database

Before you can wrap a database as a SPARQL endpoint, you need to create a mapping file. The following command (should be typed all on one line) will generate a mapping file named business_mapping.n3 from the PostgreSQL database business_test:

```
$ generate-mapping                    \\
     -o business_mapping.n3          \\
     -d org.postgresql.Driver          \\
     -u postgres -p <password>     \\
     jdbc:postgresql:business_test
```

You can then run the SPARQL endpoint service:

```
d2r-server business_maping.n3
```

The mapping file is generated by looking at primary and foreign key database schemas. D2R assigns URIs to rows in the three sample tables created in Appendix A and some examples will show you the pattern used. Here are URIs for all customers, products, and orders:

```
<http://localhost:2020/resource/customers/1>
<http://localhost:2020/resource/customers/2>
<http://localhost:2020/resource/products/1>
<http://localhost:2020/resource/products/2>
<http://localhost:2020/resource/orders/1>
<http://localhost:2020/resource/orders/2>
<http://localhost:2020/resource/orders/3>
```

A great feature is that these URIs are deferenceable. For example Figure B.1 shows the result of derencing the URI for order with *id* equal to 3. Notice the REST style URLs used in D2R.

The web application provided by D2R can be accessed using the URL:

```
http://localhost:2020/
```

You can also use an interactive SPARQL interface using the URL for the Snorql web interface[1]:

[1] Recommended!

Figure B.1.: Screen shot of D2R web interface

```
http://localhost:2020/snorql
```

If you are running D2R with the database created in Appendix A then try this query:

```
PREFIX vocab: <http://localhost:2020/vocab/resource/>
SELECT ?subject ?name WHERE {
   ?subject vocab:customers_name ?name
       FILTER regex(?name,  "Mark Watson" )
}
LIMIT 10
```

I develop the utility class EntityToRdfHelpersDbpedia in Chapter 16 that attempts to match entities line people's names and places found in text with URIs in DBpedia.