Chandos Information Professional Series

CP
CHANDOS
PUBLISHING

A Librarian's Guide to Graphs, Data and the Semantic Web

James Powell and Matthew Hopkins

A Librarian's Guide to Graphs, Data and the Semantic Web

James Powell and Matthew Hopkins
Los Alamos National Laboratory

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Chandos Publishing is an imprint of Elsevier

Chandos Publishing is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA
Langford Lane, Kidlington, OX5 1GB, UK

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

ISBN: 978-1-84334-753-8

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2015939547

Contents

This page intentionally left blank

About the authors

James Powell was born and raised in Virginia. He attended and later worked at Virginia Tech, where he honed his unique aptitude for both computer science and technical communications. About a decade ago, he moved to the southwest for a job at Los Alamos National Laboratory. He is a software developer specializing in information retrieval technologies and has contributed to a diverse array of projects at the Lab including social network applications, search engines, just-in-time information retrieval tools, Semantic Web applications, applied machine learning, and various projects aimed at managing research data for protein structures, infectious disease models, and data from NASA's Kepler mission characterizing variable stars.

Today, James lives and writes from his home in Santa Fe, New Mexico and works as a Research Technologist in the Research Library of Los Alamos National Laboratory.

Matthew Hopkins was born in Richmond, Virginia. He received his undergraduate degree from the University of Virginia and his Masters in Library Science from the University of North Carolina at Chapel Hill.

He lives with his family in Los Alamos, New Mexico, where he works at the Research Library of Los Alamos National Laboratory.

This page intentionally left blank

Preface and Acknowledgments

This book is in part an expanded version of a paper published in the ALA journal *Information Technology and Libraries* in December 2011 entitled "Graphs in Libraries: A Primer." This paper was written by myself and Matthew Hopkins, along with Daniel Alcazar, Robert Olendorf, Tamara McMahon, Amber Wu, and Linn Collins. Even though many library services are based upon or derived from network analysis, there weren't a lot of publications about graph theory and its applications in library and information science. This paper provided a good introduction to the topic, but ultimately a 5000 word feature article could do little more than scratch the surface. Several of us decided it might be worth expanding this paper into a book. We approached Chandos Publishing with a proposal and they suggested we also include material about the Semantic Web. This was a tall order, but we also knew it was spot on because the Semantic Web is an application of graph theory to information organization. We knew that if we pulled this off, we would bring together library science and the fledgling field of complexity in a way that would have a direct impact on future library services. Our book would empower librarians to speak the same language – and leverage the same insights – as complexity scientists.

Our first task was to explain where these technologies came from and why they were of interest to the library community. Our next challenge was to introduce graph theory without scaring off our audience, because graph theory is in fact a product of mathematics, and yet it is an accessible topic even for the nonmathematical inclined. Building on that foundation, we introduced many of the core concepts of the Semantic Web, including the triple, RDF graphs, ontologies, various Semantic Web standards, as well as reasoning and search. Then we surveyed the ways graph theory has been used to explore problems in many disciplines. Finally this we explored various ontologies that glue together Semantic Web graphs representing all the data that libraries accumulate about services and collections. The remainder of the book focuses on graph and Semantic Web "tools of the trade" including authoring, visualization, and storage and retrieval systems. The book ends with a pair of case studies, one based on Semantic Web technologies, the other based on pure graph modeling and graph analytic technologies. Our goal was to provide librarians and the technical staff who support them with strong conceptual foundations as well as examples of data models, rules, searches, and simple code that uses graphs and semantic data to support and augment various information retrieval tasks. We think this book has achieved this goal.

Writing a book is a huge undertaking, placing great demands and not a small amount of stress on those of us who attempt it. I couldn't have pulled off this

project without my brilliant co-author Matthew Hopkins, nor the support of friends, coworkers, and family. I owe special thanks to Linn Collins, who encouraged us to pursue this project, and if time and circumstances had permitted, would have been one of our coauthors. I worked for Linn for more than 5 years, and she was a brilliant and supportive boss, the kind of person you'll have the chance to work for once or twice in your career, if you're lucky. Linn was the expert user who supplied many requirements for InfoSynth, the Semantic Web system described in one of the case studies in Chapter 23. The other case study is based on a project called EgoSystem. The director of Los Alamos National Laboratory (LANL), Dr. Charles McMillan, engaged the library directly in discussions that lead to the requirements for EgoSystem. That project fell to the Digital Library Research and Prototyping team, which is led by Herbert Van de Sompel. I consider myself doubly fortunate because like Linn Collins, Herbert is another great mentor as well as a very supportive boss. It has been while working for him that I gained the confidence to pursue this book project. For EgoSystem, Herbert assembled a group that included myself, and another member of his team, a software engineer able to master any programming language in a vanishingly short amount of time, Harihar (Harish) Shankar. We also had the especially good fortune to engage a LANL alum well known in the graph community, Marko Rodriguez, on both the design and implementation of EgoSystem. Marko is brilliant and energetic, and he lives and breathes graph theory. What I learned from working with Marko and the rest of the Ego project team greatly influenced this book. The four of us spent countless hours in a chilly conference room in front of a chalk board hashing out a property graph model that would address many practical – and a few novel, – requirements for the system. The LANL Research Library director Dianna (Dee) Magnoni, who joined LANL midway through the EgoSystem project, and became a strong advocate for its continued development. She was also an enthusiastic supporter of this book.

There were other fortuitous happenstances that helped this project along. For more than a year, Linn and I met with colleagues from the University of New Mexico, including Dr. Diana Northrup and Johann van Reenan, at the Santa Fe Institute to discuss methods for managing research data sets related to the study of karst terrain. Perched on a rocky hillside amongst a dwarf forest of junipers, pinyon pines, and a smattering of cacti, the Santa Fe Institute is a small campus consisting of several contemporary southwestern adobe structures that blend into the rusty hills of crumbled granite. This well-hidden think tank overlooks the city of Santa Fe and has expansive views of the Rio Grande valley, the Jemez mountains, and Los Alamos to the west. It was a beautiful, inspiring, and catalyzing half-way point for us to convene. I remember perusing a number of books and papers lining the walls opposite the glass enclosed open air courtyard adjacent to the Institute's small library. That's where I first encountered the mysterious field of complexity science that seemed to concern itself with everything. A few years later, that led to me participating as a student in the Institute's first MOOC-based course, Introduction to Complexity. Although I have never met author and course instructor Melanie Mitchell personally, I feel I owe her a debt of gratitude because it was while taking

her course that I came to realize how graph theory fit in the larger context of the study of complex systems. Sometimes all it takes is one class to change your view of the entire world and for me, this was the class.

No doubt most authors experience at least a few uncomfortable moments of that special kind of despair that accompanies the task of writing a book. I found that even with a completed manuscript, the finish line can still be a long way off. And so, I owe a special debt of gratitude to one person in particular who helped me get to that finish line: Joel Sommers. The authors of any book seeking to survey and tutor the reader about a technology frets about completeness and accuracy. But I also wanted this book to be engaging. With that goal in mind, I tried to introduce each chapter with a novel application of graph theory or an interesting anecdote from computer science. Joel provided many razor-sharp edits that were right on target, and he made valuable suggestions for how I could improve some of these narratives with what he called "joyous (if melodramatic) injections of adventurism." I can't thank Joel, nor my colleagues at LANL enough for all their help and encouragement. I also greatly appreciate my coauthor Matthew's willingness to stick with this project. Without your help, this book would definitely not have happened. With your help, this book emerged as much more than the sum of its parts.

James Powell
Santa Fe, New Mexico
February 3, 2015

This page intentionally left blank

Introduction

Single file, rarely out of step with one another, a large contingent of ants marches almost as a single pulsing organism. In intimate proximity to one another, they make their way toward the remnants of a careless human's lunch. From above it is hard to see the stream of tiny ants emerging from a crater-like mound of sand.

As individuals, they are not gifted with keen sight. And despite such close quarters, they don't feel crowded, rushed, or claustrophobic. They effortlessly climb obstacles that would at our scale seem to be insurmountable boulders. They build nests, fights off invaders, and lifts massive items many times its own weight all in support of the collective.

Yet they have no leader. The colony is self-organizing. Like a road crew repainting the stripes along a stretch of highway, they constantly refresh the trail that guides their parade until it is no longer needed. Their highway is a forage line, marked by a pheromone trail. Some random ant discovered the food and excitedly established the trail for others to follow. They make quick work of the breadcrumbs and then attend to other matters. Some ants are soldiers, some are harvesters, some are diggers, and others just clean, switching roles as needed. You'll find no multitasking here.

Their behavior may be simple, but the aggregate results are complex. Ants can quickly adapt to the challenges of the local environment, achieving things as a community that would be impossible for a single ant. They do so by following simple rules, playing simple roles, and occasionally submitting to the whims of chance. They are a perfect example of what scientists call a complex system.

Ants endure and thrive in a dynamic world full of unknowns even though they have no hope of comprehending that world in its entirety. Simple rules guide individuals to perform whatever task is most pressing for their collective survival in the moment. An essential ingredient of those rules is randomness. Food foraging starts out as a random, directionless activity. When ants find a food source, they make pheromone trails to the source. The trails dissipate unless they are reinforced. When the food is gone, the pheromone trail fades away, and the ants usually retreat to their nest or move on to another source of food. Randomness takes over again. We used to believe that the universe was deterministic, that if you knew all the initial conditions and all the rules that govern it, then you could know everything that would ever happen. But ants know better. They don't plan, they react.

In our universe, the best models we have come up with to anticipate what can or might happen are based on probabilities. Inherent in probabilities is an element of randomness. Ants do perfectly fine by individually focusing on discrete, simple tasks. They react rather than acting under centralized direction, without anticipating

anything. The colony as a whole knows how to survive. But if some ants stray from the colony, the results can be disastrous for them.

The ant mill phenomenon is illustrative of how important it is for individual ants to remain connected to their colony. If a few ants lose track of their forage trail, they begin to follow one another. Pretty soon they form a circle and they will march around and around until they all die. As a collective, ants manifest complex behaviors that ensure their survival. This behavior is more sophisticated than one would assume possible given what is known about an individual component. Emergence is the term for this phenomenon, and it is characteristic of complex systems (Figure I.1).

The study of complex systems is relatively new, and there are many details around the edges that are a bit hazy. Even the precise definition of a complex system is a subject of debate among those who study them. Complex systems exist all around us and they inhabit the space between organized simplicity (simple, recurring patterns) and chaos. Complex doesn't necessarily imply complicated in the sense that the word complicated is often used. An ant isn't a complicated animal, but an ant colony is a complex system. Complicated systems are not necessarily complex. A car is quite complicated under the hood, but the components are highly individualized and play distinct roles for which they were specifically designed. There seems to be a tipping point somewhere between very simple and completely chaotic where complexity is manifest. In other words, an ant and its relationship to other ants is only as complex as it needs to be in order for an ant colony to survive and adapt to its environment.

Why the sudden interest in complex systems? Well, in the twentieth century, science hit a wall. The era when a single individual laboring tirelessly for a lifetime was able to understand and make significant contributions in a field was drawing to a close. For centuries, progress had come from the process of reductionism. Reductionism supposes that a complex phenomenon can ultimately be understood by comprehending its constituent components. But reductionism was reaching its limits. New knowledge increasingly depended on a considerable foundation of existing knowledge within and across disciplines. Some phenomena mysteriously manifested capabilities that were not suggested by an understanding of their parts. A great deal had been achieved without the benefit of modern computing capabilities. But it was inevitable that we'd reach a threshold where it was beyond a single person's ability to move a field forward. Reductionism was yielding lower returns. Scientific advancement began to depend more and more on large data sets, simulations, modeling, statistical analysis, machine learning, and these advancements suggested that there may be some unknown, poorly understood self-organizing principles that played a crucial role in some natural systems. Our ability to study complex systems is due in no small part to the advances in computing over the last few decades. And one of the techniques we use to understand complex systems is to model them as graphs.

In this book we talk a lot about models and modeling. A graph is a type of model. A model is a representation of some other thing. A paper airplane is a model that bears some resemblance to an actual airplane and can even fly. The advantage of a paper airplane is that you can make and fly your own without killing anyone.

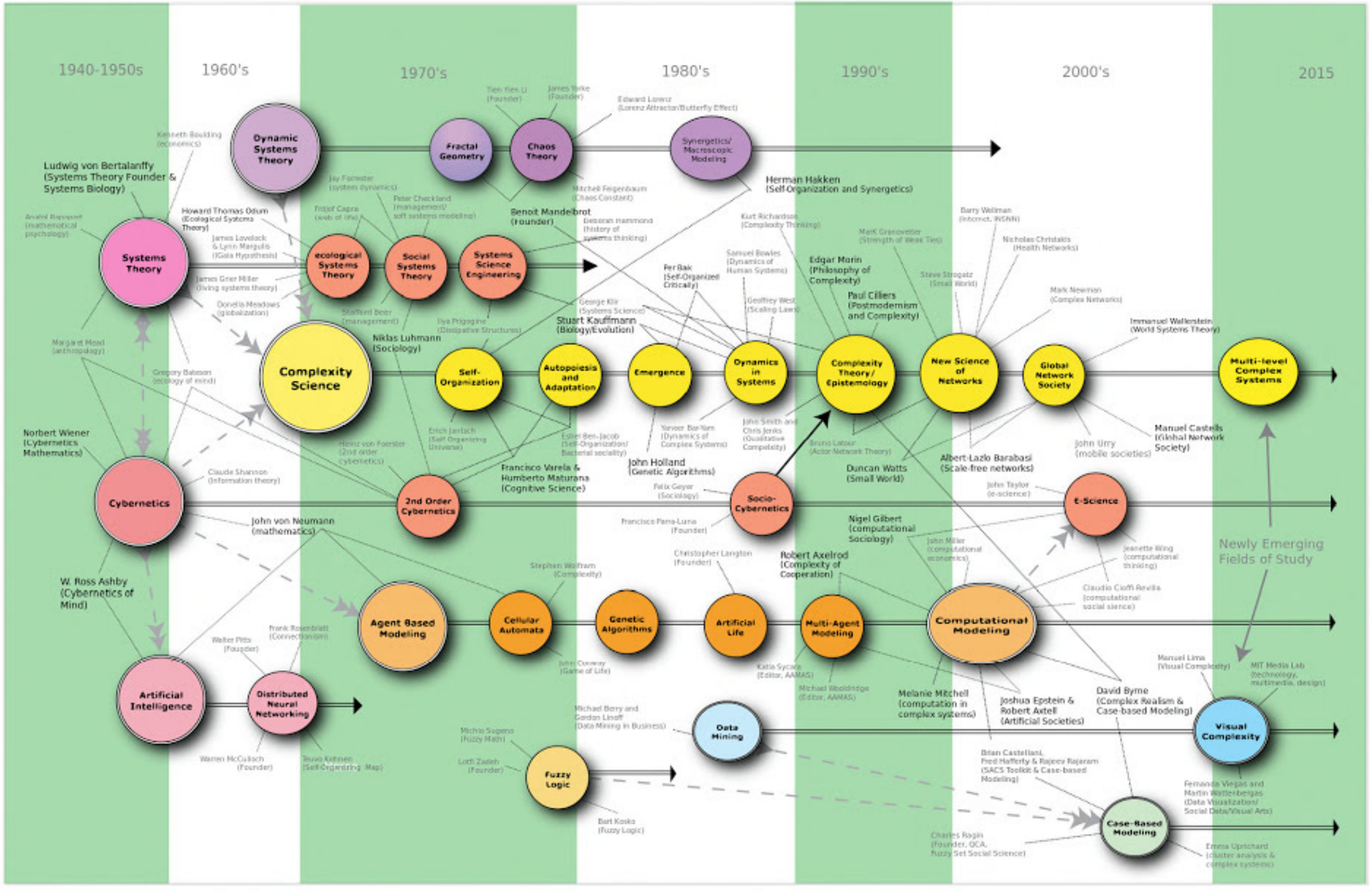

Figure I.1 A history of complexity science.
Source: Castellani, B. (2012). <http://commons.wikimedia.org/wiki/File:Complexity_Map.svg >.

Models are good for exploring aspects of a thing, and for testing ideas about it without doing any harm. Mathematicians often represent their models as formulae and they breathe life into a model by assigning values to its variables.

Graphs are not merely models; they are mathematical models. This type of graph is not the same as a bar or pie chart you're likely familiar with. It is a model of things and their relationships with one another. This model can simultaneously capture complexity and simplicity. A graph model of a complex system can reveal how it has achieved balance within itself without overcompensating, and with its environment, without eventually rendering that environment unlivable.

Visually, a graph is most often represented as a collection of dots with lines between them. The dots are an abstraction for the things that exist in a system, process, or knowledge space, and the lines are relationships among them. Graph theory is the field of mathematics that is concerned with analyzing and exploring graphs. Here is the formula that mathematicians use to describe a graph (Figure I.2):

$$\mathbf{G} = (\mathbf{V}, \mathbf{E})$$

Graphs have been used to gain a better understanding of many phenomena. For example, there was a long-standing mystery in biology regarding the energy requirements of an organism as its size increased. Intuitively, one would guess that if you doubled the size of an animal, it would require twice as much food. But it turns out that's not the case: a 75% increase is all that's required. How can that be? The solution to this mystery lies in the answer to the question: what do a city, a leaf, and a lung have in common? Geoffrey West of the Santa Fe Institute explains:

> *The key lies in the generic mathematical properties of networks. Highly complex self-sustaining systems . . . require close integration of many constituent units that require an efficient supply of nutrients and the disposal of waste products. . . .this servicing — via, for instance, circulatory systems in organisms, or perhaps transport systems in cities — is accomplished through optimized, space-filling, fractal-like branching networks whose dynamical and geometric constraints are independent of specific evolved organismic design.*

Blood vessels, the vascular system in leaves, and transportation routes to and within a city are not just like networks, they are networks. Nature settled on power-law networks to solve many problems. We're not far behind. The infrastructure that supports our cities, our regional and global transportation systems, and our power grids are networks that have similar properties to evolved network systems. So it should come as no surprise that graph models are useful for understanding many aspects of the natural world.

A graph is a great way to model contacts involved in the spread of disease. The graph starts with patient 0—the first person infected by a disease. From previous research, it may be known that this particular disease has a reproduction number of 2. That means that every infected person infects on average two more people. The incubation period may be on average 4 days before symptoms appear. Patient 0 was

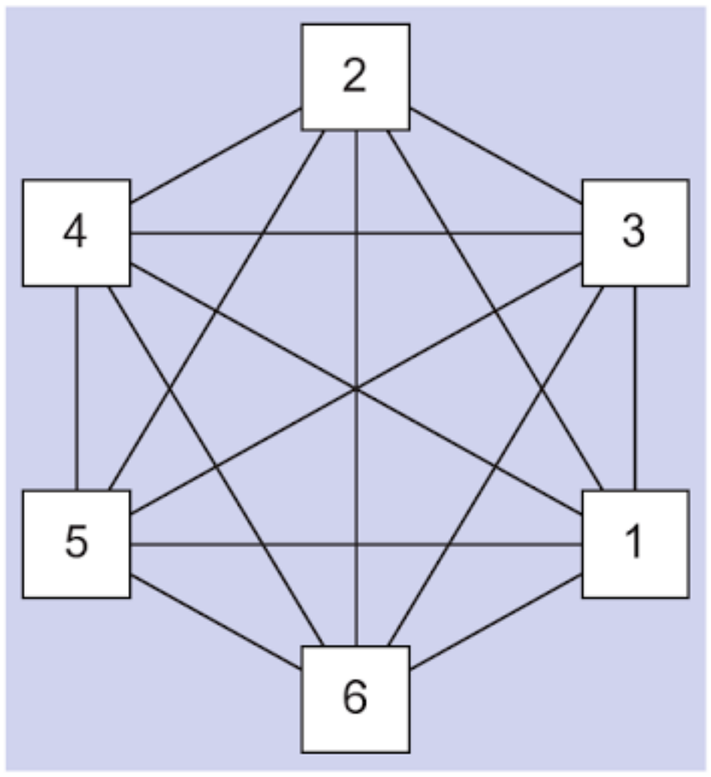

Figure I.2 G = (V,E) says that the graph G is a set of vertices V (labeled 1,2,3,4,5,6) and edges E.

infected 8 weeks ago, and at least 125 people have shown up in area hospitals with the disease over the last 7 weeks. Health care workers begin interviewing patients to determine who had contact with whom. If the contact preceded infection, and only one infected individual is identified, then there's a clear relationship. Pretty soon a tree-like graph emerges that confirms that the reproduction rate has been right around 2. The good news is, as time passes, the graph has more and more nodes with no new connections at all. Quarantine and early treatment are working. The disease seems to be burning itself out. The graph model provides tangible evidence of this. As you might guess from this example, epidemiologists love graphs.

In fact, many disciplines love graphs. Sociologists study communities and communication with graphs. Chemists can study graph representations of compounds before they ever attempt to produce them in the lab. Physicists can study phase transitions and myriad other aspects of matter and energy. Astronomers can study galactic clusters with graphs. Environmental scientists can represent food chains with graphs. Cell biologists can model metabolic processes in a cell and even get some sense for how such a process evolved over time. Economists can study purchasing or investing patterns among consumers. City planners can study traffic patterns to figure out where mass transit might relieve congested roadways. When that transit system is in place, the subway map daily commuters will use is itself a graph. In a connected world, being able to analyze connections is a powerful tool. A portion of this book is devoted to the use of graphs in various fields, and we use these reviews to illustrate various important concepts in graph theory, often building on concepts introduced in earlier chapters.

Tim Berners Lee looked to graph theory for inspiration as he developed a concept for a global network of information. His vision included a democratic model where anyone could encode and share any facts. These facts would be comprehensible by both man and machine. He sometimes even referred to this as the Global Giant Graph. His goal was to push the granularity of information representation down to the level of individual facts. These facts would be shared and

interconnected. And like the World Wide Web before it, there would be an ecosystem of standards and technologies upon which this system would be built.

We refer to this Web of interconnected knowledge by various names, including Linked Open Data, the linked data web, or simply the Semantic Web. At the core is a simple information model which defines the structure of a fundamental unit of knowledge called a triple. A triple is a graph segment made up of two nodes and a directed edge between them. These segments reside in a larger knowledge graph. This graph coexists with the World Wide Web. Nodes are represented by unique network identifiers that give a thing a unique name and provide a pointer to it. Identifiers make it possible to add more graph segments that further describe these things. This simple information representation model results in a complex web of knowledge. Sound familiar?

This book is intended to provide an overview of a number of graph-related topics. We have attempted whenever possible to write it so that each chapter can be consulted independently of any other chapter, but there is at times an unavoidable progression of background knowledge that is prerequisite for understanding later chapters.

It is a book of lists. Lists encapsulate and summarize in a way that most everyone finds helpful on some level. Of course when we do start off with a list, we follow up with a narrative that expands the list elements into the things you really need to know about a given topic.

It is a book of examples. To the extent possible, we don't just talk about a given technology, but try to show you examples and explain what it's good for. Standards documents tend to be broad and deep, as they should. We tend to feature vertical slices of a given technology which we hope will give you a good idea of some of the uses for it and an incentive to learn more.

It is a book for people who love books and reading. We try to bring the topics to life with anecdotes from current research papers, other more technical sources, and even historical texts and novels. These are intended to engage, enlighten, and occasionally even entertain.

It is a book that covers more topics at an introductory level, rather than a few topics at a deeper level. You may never need to delve deeply into some topics we cover, but we aim to make sure that if you ever hear about or encounter the topic again, it will be familiar, not foreign.

It is, on occasion, a book of code. Some chapters discuss program code, markup, and information representation languages. If you never plan to write software, you may not need to delve deeply into the handful of chapters that discuss programming and APIs, but we do our best to provide a nontechnical overview before we show you any code. If the code is more detail than you need, skip forward and share the code with a programming friend or colleague. If they are tasked with solving a problem using that particular technology and are new to it, they may thank you profusely, as we strive to introduce concepts from a beginner's perspective.

It is a book to help you get things done. These are big topics. There are more rabbit holes here than in the proverbial briar patch. We steer you clear of the rabbit holes by highlighting solid, widely adopted technologies, not standards and technologies still under development that may or may not come to fruition. We introduce you to things

that you can use now. We provide concrete examples throughout and we conclude the book with in-depth case studies of two real-world applications. We believe learning is good, doing is good, but learning while doing is better.

With this book, we will introduce you to some of the finer points of graph theory and the Semantic Web. We hope to provide you with a solid conceptual framework of graph theory, and a comprehensive overview of Semantic Web technologies, which we think makes this book unique in this field. For librarians, it will help you understand how patrons in a variety of fields might be modeling aspects of their field of research, and how you might apply graph theory to some information discovery and retrieval challenges in your library. For information technologists such as software developers, it will give you the background you need to understand how graph theory and the Semantic Web can be leveraged to represent knowledge. There are many open source tools, software libraries, and standards for graph data and for the Semantic Web. We will introduce some specific tools but also give you the knowledge you need to find and evaluate other similar tools.

Although this book is not specifically about complexity science, it is a guidebook to some of the technologies used to model and elucidate aspects of complex systems. There are many excellent books about complexity and complex systems, including "Complexity: a Guided Tour" by Melanie Mitchell, "Deep Simplicity" by John Gribbin, and "Simply Complexity" by Neil Johnson, just to name a few. Any or all of these resources may help you understand the field and its myriad applications.

Libraries are constantly striving to provide new and better ways to find information. We've been doing this for hundreds of years. Graph theory and Semantic Web technologies offer a way for libraries to reinvent themselves and their services, based on relationships. Finding things in libraries used to depend on physical access to those things. Then we introduced layers of abstraction: classification schemes, call number systems, card catalogs, online public access catalogs, and finding aides, to name a few. Yet relationships may be the most natural way for users to find things. Graphs and the Semantic Web are great at modeling and exploring relationships. Libraries can model the relationships among content, topics, creators, and consumers. Rather than mapping all our previous solutions onto the Semantic Web, perhaps it is time to develop a "relationship engine" that can leverage these new technologies to totally reinvent the library experience.

Humans are obviously not ants. But humans can model ant behavior using graphs. We can model the inner processes of an ant's physiology using graphs. We can model an ant colonies' relationship to its environment using graphs. We can model what we know about ants using graphs. Graphs are tools that help us understand complex systems. In a nondeterministic universe, we face myriad unknown and complex challenges, some will be random events, and some will be of our own making. It is contingent upon us to use every means at our disposal to augment our ability to comprehend complexity. Turning our back on complexity is no longer an option. Graph models of complex systems, and the practice of embedding knowledge into graphs, are powerful tools for comprehending complexity—and they enable us to use that understanding to our advantage.

Glossary

graph a set of things and the relationships between them, visually it is often rendered as a collection of dots and lines between them, mathematically a graph represented by the variable G, and is defined a set of vertices and edges {V,E}, thus G = {V,E} is a graph

node an object in a graph, in the mathematical definition of a graph, a node is referred to as a vertex, represented above by the variable V

edge a relationship in a graph, they connect nodes, same in mathematical definition of a graph, represented above by the variable E

property (graph) in a property graph a property is a characteristic of a node or edge, to which a value is assigned, for example, the property name has the value "James"

property (ontology) in a vocabulary a property is a characteristic of a Class, and in RDF it is usually the predicate in a triple. Property names start with a lower case letter and employ camel case, that is, there is no white space in a property name but every distinct word that forms part of the property name is capitalized

class refers to a distinct thing defined in a vocabulary. class names start with an upper case letter

metadata data about data, a collection of predefined terms intended to describe, label, point at or encapsulate other data

instance data this is data that is described by metadata or some element of a vocabulary, for example, an Oak could be considered instance data for a class tree, subclass species, property common name = "Oak"

vocabulary a group of terms that are defined as having some relationship with one another, which describe some thing or some knowledge domain

taxonomy a hierarchical knowledge representation defining a vocabulary

ontology (general) a structured model for representing things that exist in a given domain, their characteristics, and the relationships between them

ontology (Semantic Web) a model representing a vocabulary to describe aspects of and relationships among data

reasoning application of a logical set of decisions to a situation or to data

visualization a pictorial representation of some data, intended to convey details about that data and designed to be viewed on a computer screen or some other form of visually consumed projection

triple a syntactic representation which forms a statement of some information that consists of three distinct components separated in some fashion (e.g., by white space), arranged so that the second (predicate) and third (object) parts of the statement provide information about the first (subject)

URI a uniform, universal resource identifier

namespace a uniform resource identifier that uniquely identifies a vocabulary

vertex for our purposes, a vertex and a node are synonyms

network although network is often used interchangeably with the word graph, a network is a special type of graph that contains weighted (indicating strength of connection), directed edges (indicating direction of relationship)

Graphs in theory

1

Bridging the history

In 1736, partway through Konigsberg, Prussa, the Pregel River split the city into two parallel segments, inscribing an island between the north and south regions of the city. This island itself was also divided by a stream that connected the two forks of the Pregel River. What we know today is that churning somewhere in the river that simultaneously nourished and partitioned Konigsberg, somewhere streaming amid its divisive canals, Graph Theory was about to surface, and it would change everything.

That city, and the inherent problems bestowed by its water-carved enclaves, was the inspiration for a mathematical puzzle, the Seven Bridges of Konigsberg Problem. There were indeed seven bridges in the city, at least at that time. The Pregel River divided the city into north and south. Bridges connected the resulting islands. The western most island had two bridges to the north and two to the south, the eastern island had one bridge each to the north and to the south, and the two islands themselves were connected by a seventh bridge (Figure 1.1).

Mathematicians wondered, could one walk through the city such that he crosses every bridge once and only once?

Leonard Euler solved this puzzle, and in doing so, laid the foundations for graph theory. In considering the problem, he first stripped away the buildings, the roads, the trees—everything but the bridge crossings themselves, the connections between islands. He saw that the answer, whatever it was, could be represented simply by a sequence of these crossings. That is, the answer might be 3, 6, 4, 5, 7, 1, 2. Or 6, 3, 1, 7, 2, 4, 5. It all depended on how one numbered the bridges. But all that mattered was the order in which the bridges were crossed, not the particular route a person took to reach them. In essence, he reduced the problem to a graph.

Graph theory had not been invented at this point. But in his parlance, Euler saw the land masses as vertexes (also called nodes) and the bridges as edges. These are the building blocks of the simplest graphs—a set of nodes connected to each other by edges.

This simplified formulation of the problem made certain features stand out. For instance, Euler understand that for every land mass, one must enter it via a bridge and then leave it via a different bridge. The exception(s) are the starting and ending land masses. But there are at most two of these (one if the walker starts and ends at the same place); the other land masses are intermediate nodes, which must be entered and exited an equal number of times. Each entrance and exit relies on a unique bridge, so the number of bridges connected to one of these intermediate land masses must be even, or else the walk is doomed from the start (Figure 1.2).

Euler discovered that for a graph like this to have a solution, either all the nodes must have an even number of edges or exactly two of them could have an odd

A Librarian's Guide to Graphs, Data and the Semantic Web.

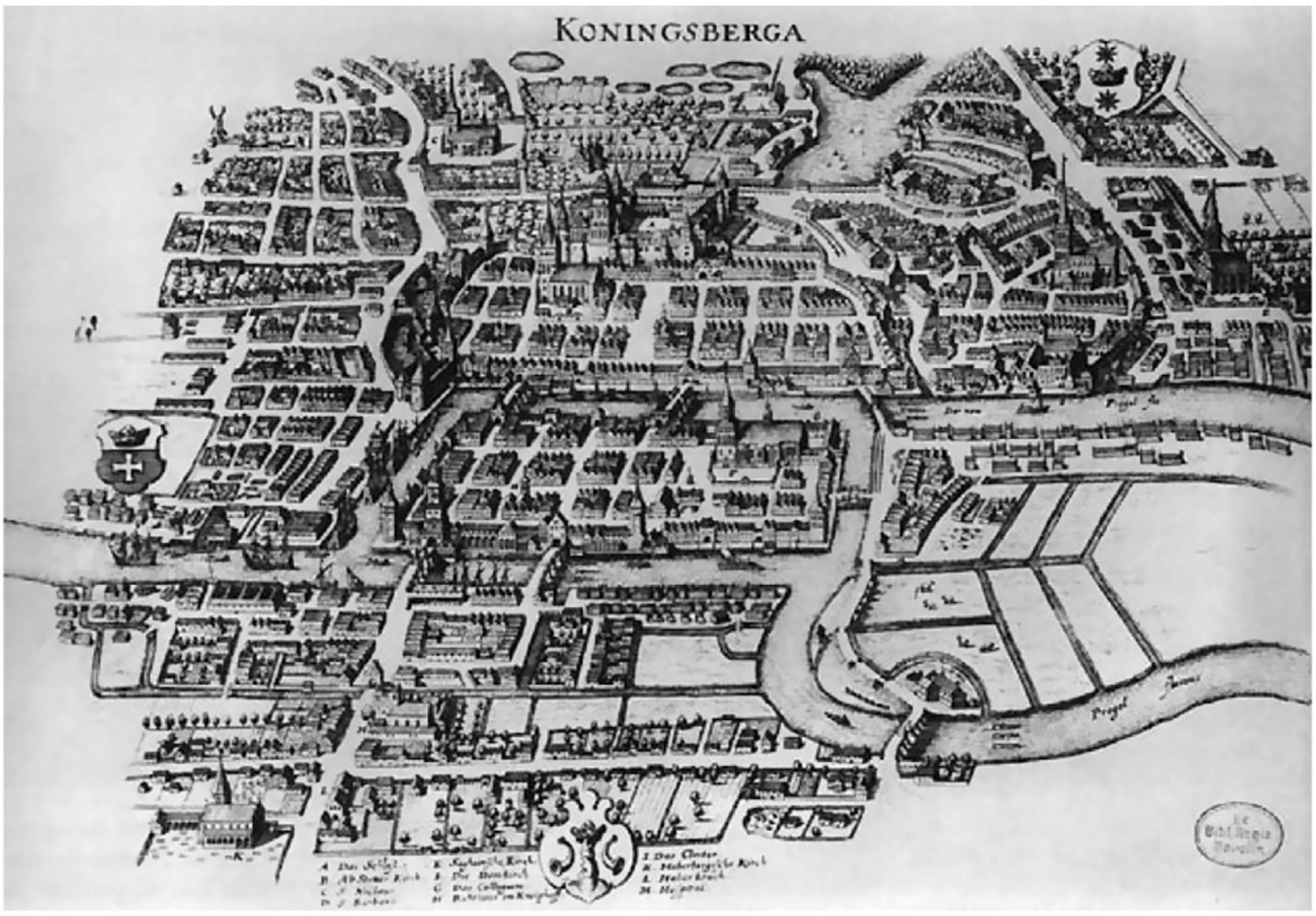

Figure 1.1 A drawing of Konigsberg as it appeared in 1651.
Source: Map by Merian-Erben. <https://commons.wikimedia.org/wiki/File:Image-Koenigsberg,_Map_by_Merian-Erben_1652.jpg>.

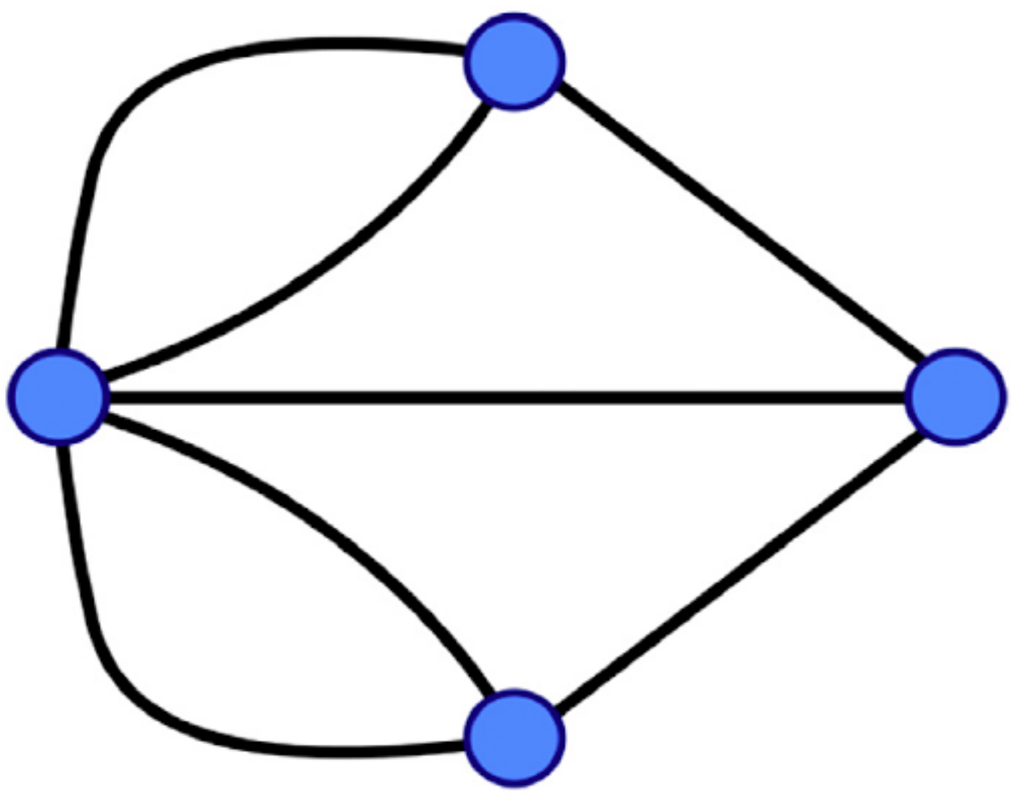

Figure 1.2 Konigsberg bridges as a graph.

number of edges (in which case the successful walker must start at one of these nodes and end at the other). But all the land regions of Konigsberg had an odd number of bridges. Thus, there was no solution. One could not walk a Eulerian path through Konigsberg (as the goal later became called), and certainly not a Eulerian circuit, which is a walk that starts and ends in the same place.

Often in math or science, whimsical problems that are investigated purely to satisfy the curiosity of those doing the investigating can beget solutions that lead to many important findings. This is true even when the solution is that there is no solution at all. (Though with the bombing of two of the bridges of Konigsberg, now called Kaliningrad, during WWII, a Eulerian path is now possible.) In reducing the problem, transforming bridges to edges and land regions to nodes, Euler generalized it. And his findings, and all those that came after, can apply to any problem which involves entities connected to other entities. Connections are everywhere—from the path of infectious diseases through a population to the circulation of money in an economy. These sets of connections, or networks, can be visualized as graphs, and they can be analyzed with graph theory.

Topology

Before we introduce more examples in the history of graph theory, we should note that Euler's solution to the Seven Bridges of Konigsberg problem is also seen as a founding work in the field of topology. A graph can be a generalization of a real world network, like the bridges spanning a city. But while that city has a definite north and south and has varying distances between its land regions, once it is translated into a graph, those distinctions no longer matter. A graph can be twisted around. Its nodes can be big or small or different sizes. Its edges can be long or short or even squiggly. This is a central tenet of topology, in which shapes can be transformed into one another as long as certain key features are maintained. For instance, no matter how you bend a convex polyhedron—like a cube or a pyramid—it is always true that $V-E+F=2$. This is the Euler characteristic, in which F represents the number of faces of the shape, and V and E represent the vertexes and edges we are already familiar with.

The world is teeming with graphs. When you have this mindset, you see them everywhere. Graphs exist in a city's bridges, in something as small as a pair of dice to something as large as the great Pyramids, and even something as vast and complicated as the internet. (And Google's search engine, which helps tame that complexity, does so in part by exploiting graph theoretics, as we will see in a later chapter.) This book, as part of a bibliographic network, can exist in a graph. You yourself live in a graph. After all, people (nodes) form relationships (edges) with other people (more nodes), so that the whole of human society comprises one large graph of social connections. One might ask, then, what type of graph are we?

Degrees of separation

Yes, there are different types or classes of graphs. Some graphs have many connections, while some have few. Sometimes these connections cluster around certain key nodes, while other times they are spread evenly across all the nodes. One way

to characterize a graph is to find the mean path length between its nodes. How many "degrees of separation" exist between any two nodes? In the 1929 story collection *Everything is Different*, Hungarian author Frigyes Karinthy posited the notion that everyone is connected by at most five individuals (hence the term, six degrees of separation). One of his characters posed a game—they should select a random individual, then see how many connections it took to reach him, via their personal networks. Stanley Milgram brought this game to life in his "small world experiments" starting in 1967. (These studies are not to be confused with the Milgram Experiment, which tested a person's willingness to obey authority, even when instructed to deliver seemingly painful shocks to another person, although this too could be graphed.)

Milgram recruited people living in the midwestern United States as starting nodes and had them connect to individuals living in Boston. Note that in a graph, distance has special connotation. The metric distance between two nodes as they are drawn on a sheet of paper is arbitrary. Nodes can be rearranged, shifted closer or farther apart, as long as the connections stay the same. Distance, or path length, refers to the minimum number of edges that must be traveled to go from one node to another within a graph. In Milgram's small world experiment, it is possible, perhaps even likely, that this path length distance is related to the geographical distance separating two nodes—i.e., Boston to Milwaukee. But in a graph of human social connections, the social circles of two nodes may contribute just as much to the path length distance between them. In choosing his start and end points, Milgram hoped to accommodate both geographic and social distance.

Suppose you were selected as one of his starting points. You would receive a packet in the mail. Inside you would find the name and info of a person in Boston. Perhaps you happen to know this target personally. If so, the instructions tell you to send the packet directly to him. More likely you don't know him, so instead you sign the roster and send the packet to someone you do know personally. You choose someone who you think is more likely than you to know the target personally. The experiment was basically a chain letter. And 64 out of 296 packages eventually reached the target, with an average path length between 5 and 6.

Of course, the procedure is not perfect. Any given node who receives the package might give up and never send it on. The longer the chain, the more likely this will happen. Thus, long chains were more likely to die out, thus underestimating the true average path length that connects people. On the other hand, people are only guessing at what the best path might be. They might not find the best chain, and this overestimates the true average path length. Either way, the experiment does tend to confirm out intuition that "it is a small world, after all."

Four color problem

In our next example, we return to maps. This is fitting, as some of the graphs we are most familiar with in our daily lives are subway or train maps. This puzzle, however,

concerns cartographic maps. The four color theorem states that any map of connected regions, such as the mainland United States or the countries of Africa, requires no more than four colors such that no two adjacent regions share the same color. Here, adjacency does not include corner-to-corner touching. So in that map of the United States, New Mexico and Colorado are adjacent, but New Mexico and Utah are not.

The theorem can be stated more simply and directly in the language of graph theory. Instead of a map, consider every region a vertex or node. Edges will then connect every pair of nodes which share a boundary segment (but not a boundary point, i.e., a corner). In graph theory parlance, this particular graph is planar. This means that it can be drawn in such a way, with points and lines, that none of the lines cross each other, and they only intersect at end points. We can confirm this by returning to the map configuration. Just draw a dot in the center of each region and then connect adjacent dots with lines. Now we can rephrase the constraint of the problem this way: the nodes must be drawn with at most four colors such that no connected nodes receive the same color. In short, "every planar graph is four colorable" (Figure 1.3).

The similar five-color theorem was proved in 1890 by Percy John Heawood in the midst of correcting an error in an earlier erroneous proof of the four color theorem. In the decades since, there had been many false proofs, or rather, false disproofs, of the theorem. If the theorem were untrue, then the disproof would be a simple matter of creating a map which required more than four colors. Many such potential maps were offered, and all had errors—with a change of the colors of a few regions, or sometimes many regions, they could be four colorable. As no counterexample had been demonstrated, the consensus was that the theorem was likely true; actually proving that was another matter.

Kenneth Appel and Wolfgang Haken proved it in 1976—the first major mathematical proof to use extensive computer assistance. The proof took years to gain acceptance in the mathematical community, for no mathematician could simply check it line by line. But we can describe the general method used, if not the details.

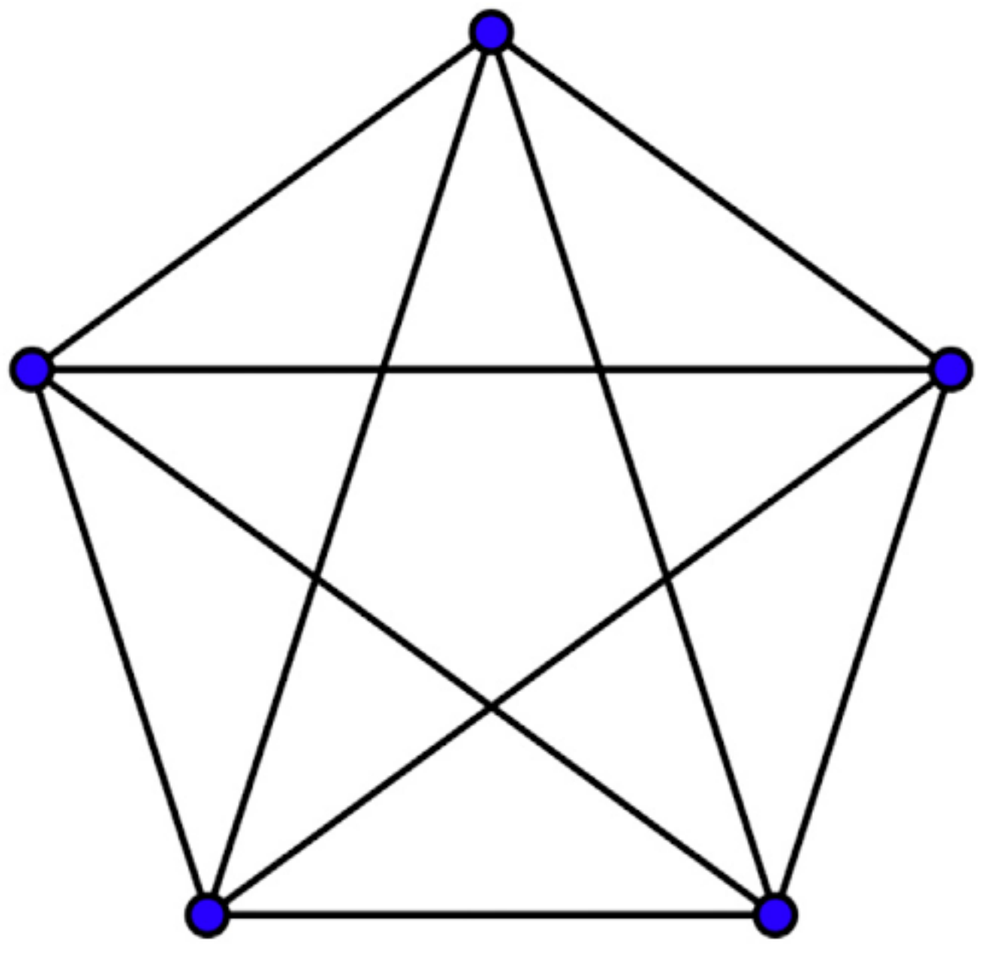

Figure 1.3 A non-planar, completely connected graph.

Imagine that the proof is false. Then there will be some region of a map that requires five colors. Reduce this five-color region to the smallest possible subgraph—i.e., take away any extraneous nodes or edges, leaving just enough so that the remainder still requires five colors. With this imagined minimum subgraph in mind, the proof proceeded this way. On the one hand, Appel and Haken devised a set of 1936 unique maps which are not part of this minimum five-color subgraph (i.e., they are all four colorable). On the other hand, they showed that any potential five-color subgraph contains within it one of those 1936 subgraphs. By the first point, the hypothetical five-color subgraph cannot contain any of those 1936 maps. By the second point, it must contain one of those 1936 graphs. Thus, it cannot exist, and the proof is true.

The De Bruijn-Erdos theorem (1951, preceding the four color proof) can extend the four color theorem from finite planar graphs to infinite planar graphs. It states that a graph can be colored by *n* colors if and only if every finite subgraph within it can be colored by *n* colors. Don't worry, we are done with serious math, at least for this chapter. We only bring up this theorem to introduce the mathematician Paul Erdos, a prolific researcher in the field of graph theory, and the subject of the final topic of this chapter. (He is also Hungarian like author Frigyes Karinthy. . .small world!).

Erdos has been honored with the Erdos number, a metric that describes the degree of separation between any researcher and Erdos himself. So someone who collaborated on a paper with Erdos would have an Erdos number of 1. Someone who had never collaborated with him, but had coauthored with one of his coauthors, would have an Erdos number of 2, and so on.

If this sounds familiar, it might be because you are thinking of the Bacon number, which is the degree of separation between a given actor and Kevin Bacon. There is even an Erdos-Bacon number which is the sum of a person's Erdos number and Bacon number, for those who have both published in academia and acted in Hollywood. For instance, the physicist Richard Feynman, who appeared in the film Anti-Clock (1979), has Bacon and Erdos numbers of 3 each, for an Erdos-Bacon number of 6. Natalie Portman also has an Erdos-Bacon number of 6. Graphs are still fun almost 300 years later. But they have also emerged as an important new tool for understanding the world around us.

Graphs and how to make them

2

Space junk and graph theory

Space is vast, and yet we still find ways to fill it. There are tens of thousands of pieces of junk orbiting the Earth. Many travel at incredible speeds. In fact, a discarded screw can cause as much damage as a bowling ball hurled at 300 miles per hour. And the problem is only getting worse, as these objects collide with one another and produce even more pieces of junk.

We don't yet have the ability to track all of this junk or to clean it up. It threatens the lives of astronauts, and even small pieces of debris can do tremendous damage to orbiting satellites. Many of our communication systems depend on those satellites. Over time, these same satellites inevitably fail and become liabilities as they drift from their original orbit or tumble out of control. It may seem like an easy problem to solve, just send up the robotic equivalent of a pool boy, with a net, to scoop up the junk and hurl it into the atmosphere where it will burn up. But its not that easy.

This dangerous game of space billiards seems as though it has very little to do with networks. Yet in a 2009 paper in *Acta Astraunatica* entitled "A new analysis of debris mitigation and removal using networks," researchers developed a theoretical network model for representing orbiting space debris and the interactions among these objects. In their graph model of space debris, the relationship between pieces of debris are interaction represents a collision or the potential for a collision with another object in the future.

The paper spent some time introducing the concept of a graph, and various graph analytic techniques to an audience that had likely not thought about space junk and networks as having any relation to one another. By analyzing this graph of space junk using some of these techniques, they identified a strategy that might help reduce the amount of space junk with minimal effort. Their graph model of debris interactions was vulnerable to what is known as a network attack. In this case, network refers to the Internet. If someone wants to affect the performance of the Internet or limit its access in some fashion, they can use a graph model of the network to identify highly connected points which, if damaged, would significantly affect the performance or availability of Internet services.

It turns out that the debris interaction network is also vulnerable to a network attack, which is obviously a very good thing. In this case, you identify a piece of space junk that is highly connected, that is, it is likely to collide with other debris. The authors suggest that if you can remove that piece of junk, you will have changed the network. And the effect would be greater than if you selected some piece of debris at random.

A Librarian's Guide to Graphs, Data and the Semantic Web.

Graph theory and graph modeling

Graph theory is the name for the discipline concerned with the study of graphs: constructing, exploring, visualizing, and understanding them. Some types of graphs, called networks, can represent the flow of resources, the steps in a process, the relationships among objects (such as space junk) by virtue of the fact that they show the direction of relationships. We intuitively understand graphs and networks because we encounter them in the real world. A family is a hierarchical network, with parents linked to children, siblings, other ancestors, and so on. Trees are a network of leaves or needles, connected to branches, which connect to the trunk, and they invert this form under ground to form roots. A spider's web is a network of filaments spanning the flight path of unsuspecting insects. In winter, whether we know it or not, as the common cold spreads through a school or business, the germs move from person to person in such a way that we can retroactively document the spread and predict their future path (and possibly, how to thwart it) via a directed graph. When we say something went viral, we are paying homage to this contagious behavior. Subway maps, airline hubs, and power grids are all examples of real world networks.

A graph can be used to model many systems, processes, and technologies. Therein lies the challenge, the interesting problem in terms of graph modeling: what part of the system or process to model, and how to represent it as a graph. This is referred to as graph modeling. It may or may not come as good news to you to know that there is almost never one right way to model data in a graph. Modeling usually happens at two levels, first the things and relationships that are immediately apparent in the domain data are mapped into a graph. Then you make another pass where you essentially test the graph model to see if there are important concepts that are lost or ill-represented by edges or as properties. Sometimes these may need to be "promoted" to nodes. Fortunately graph modeling is fairly intuitive and the modeling process can be inclusive, all you need to get started is a white board!

Scientists and researchers studying complex systems often use graph models because it is one of the best abstractions we have for modeling connectedness within a system. Sometimes the apparent network model so closely resembles the actual system that it even becomes tempting to think of the system as a network. As more and more scientists have modeled things using graphs, they have borrowed techniques from one another and developed new ways of exploring these networks. Then other researchers apply these new techniques to their graphs, and often they lead to new insights. Think of it like convergent evolution where animal forms tend to resurface in isolated and only remotely related populations. The tasmanian tiger looked like a dog, even though they are not related. Graph analytic techniques converge on significance despite the differences of what is being modeled. A completely connected graph can represent water molecules in an ice cube or a close-knit family or a group of coauthors who frequently work together.

Graphs are abstract conceptual models for representing some aspect of the world that you can detect, observe, or infer. It is a way of representing things and the relationships between those things. It is understandably easier to think of a graph the way it appears in pictures, as a bunch of circles with lines between them. Visualization tools will draw an image of a graph and try to minimize overlap using various strategies. With some tools, you can move nodes around and change the layout. It is important to keep in mind that a graph visualization is just one way to look at the graph data. No matter how a graph is drawn, the nodes and relationships between them do not change when the layout is changed or manipulated, any more than shuffling a deck of cards changes the contents of the deck.

Nodes, or vertices, and edges are the building blocks of all graphs. Weighted graph use some criteria to assign a numeric value representing the importance, for example, of a node or edge. A directed edge illustrates the direction that information or resources flow between two nodes in a graph. A more generalized graph is the property graph. A property graph associates additional data with each edge and node in the graph. You can think of these properties as a little database associated with each node or edge. Every node and edge has one or more key, value pairs that represent various characteristics of the objects and relationships. You can inspect the properties of a node or edge by first locating it in the graph via a path traversal, graph analysis such as degree metrics, or some type of search, such as a guided or breadth first search. Since every node and edge has a unique identifier, you can use this to get at its properties. If you know the key you are interested in, you can present it, or inspect the keys associated with that node or edge. In property graphs, path traversals, searches, you can make use of node and edge properties as you explore the graph, for example to select particular edges along a path or identify a group of nodes with common characteristics.

A graph representation often necessarily results in a reduction of information about relationships. Here is a trivial example. You have data for a social network that includes people, their friends, and their neighbors. However, you assume that many people don't know their neighbors very well, so you decide to create a graph that only contains friend relationships. This graph contains only a subset of the actors in this relationship. Many graph analytics, such as degree centrality which is counting the number of connections each object in a graph model has, implicitly assume that the objects and relationships in your model are each of a single type. Maybe you have the complete graph of friends and neighbors but you want to extract the friend graph for processing. You can create a subgraph, or a projection of the more complex graph containing friend relationships and the nodes connected to them, and analyze only that subgraph. Subgraphs can also be used to produce simpler visualizations of complex graphs. Large, complex graphs with many nodes and edges can quickly become what we sometimes call "inscrutable hairballs." Visually they are so complex and so tangled that any hope of intuitively learning anything from the visual representation is lost.

Analyzing graphs

There are many possible ways to explore graphs. Researchers sometimes look at graph analytics in different ways. For example, those who use graph models to model actual systems and processes tend to focus on these areas:

1. Network structure
2. Properties of the graph elements
3. How the network evolves over time
4. What kinds of interactions can occur on the network.

In general, graph analytic techniques are used to explore connections, distributions, graph-wide features, and clusters. Connections refer to relationships between objects in a graph (the edges). Some graph analytic techniques are based on counting, that is, identifying some element of a graph and determining quantities related to it. Degree centrality is based on how many edges are attached to a node. Path length is a count of the number of hops between two nodes. In directed graphs, the aspects of the graph that can be quantified can differentiate between in-degree and an out-degree values for a node. These values correspond to the number of incoming and outgoing edges attached to a node. For a graph in which weights have been assigned to edges, degree centrality can be a summation of the weights of all the edges, instead of a count of the edges.

A slightly more complex form of analysis called betweenness centrality is a measure which reveals which nodes are playing a primary role in connecting portions of the graph. In other words, were one to remove a node with high betweenness centrality, portions of the graph could become disconnected or at the very least, the average path length might increase. As we described at the beginning of this chapter, research into space debris as a network found that debris mitigation strategies would be most effective if they identified and eliminated highly connected nodes. Path analysis such as distance between nodes and shortest paths, or identification of particular kinds of paths such as Hamiltonian or Eularian paths, is edge centric means for exploring connections across the graph. Another node measure is cluster coefficient. The local cluster coefficient is a per-node value that is the number of actual connections per node divided by the maximum possible number of connections per node.

Distributions explore global aspects of a graph. Distributions identify statistically significant aspects of the graph, both respect to nodes and edges. This is not unlike the analysis that a researcher might perform on data resulting from a series of observations or experiments. Distribution analysis can incorporate centrality measures, graph density, strength of connections, path analysis, and identification of hubs. From this you can determine the average degree for the graph and identify nodes with the highest and lowest degree values. This type of analysis can also determine if the graph exhibits properties of a particular type of graph such as a small world or scale-free network. Another form of analysis that is the calculation of graph density. This is a comparison of node connectivity in a graph to an abstract graph model where all nodes are interconnected. Since distribution analysis is

concerned with global graph characteristics, it often involves comparing a graph with examples of random, small world and scale-free networks to see how closely it resembles one of these graph types.

Commonly occurring and widely studied graph types include:

- Lattices
- Trees
- Random graphs
- Small world graphs
- Regular graphs
- Scale-free graphs.

Cluster analysis is the process of identifying and analyzing clusters. Clusters are groups of nodes that share more connections with one another than with the graph as a whole. As with graphs, there are types of clusters that recur across graphs such as cliques. Cliques are collections of nodes that are highly or completely connected with one another, but sharing only a few connections with the graph as a whole. The cluster coefficient plays a role in identifying clusters since it is a per-node value that indicates how interconnected each node in a graph is with its neighbors. Node betweenness centrality is used to determine which vertices connect clusters to one another or to the graph.

Individual node centrality measures determine the relative importance of a node based on some factor, such as how many edges connect to it, or whether it serves as a bridge between groups of nodes (degree, betweenness centrality). Identification of clusters of nodes looks at the relative number of edges among and between groups of nodes. Graph-wide node-based metrics consider the overall distribution of node degrees, the role that some nodes play in the structure of the graph (degree distribution, hubs), and the characteristic found in some graphs where some aspect of nodes affects their likelihood to be connected in large groups that form partitions (homophily vs. heterogeneity).

Graph matching is a process of evaluating how similar two graphs are, such as whether they share the same types of nodes, whether the edge pattern is similar, and this measure can also take into consideration edge directionality and weights if appropriate. Then there are path measures, which fall into two broad categories. One looks at graph-wide characteristics, such as the graph diameter, whether or not there are cycles, the overall connectedness of the graph, etc. The other considers paths between nodes and measures distance, shortest path, etc. Many of these conceptual categories of graph analysis have been used for a long time. Most of the graph analytic concepts that we will discuss in this book fall into one of these broad categories. Advancements in graph analytics tend to involve refinements to these various approaches, although small world and random graphs, graph matching, and some other computational intense approaches are recent innovations that have emerged with the advancement of computers and software needed to make such analysis practical (Figure 2.1).

A graph model can tell you many things about the system or process it represents. In the last decade or so, much has been made of graph-wide metrics. Graph

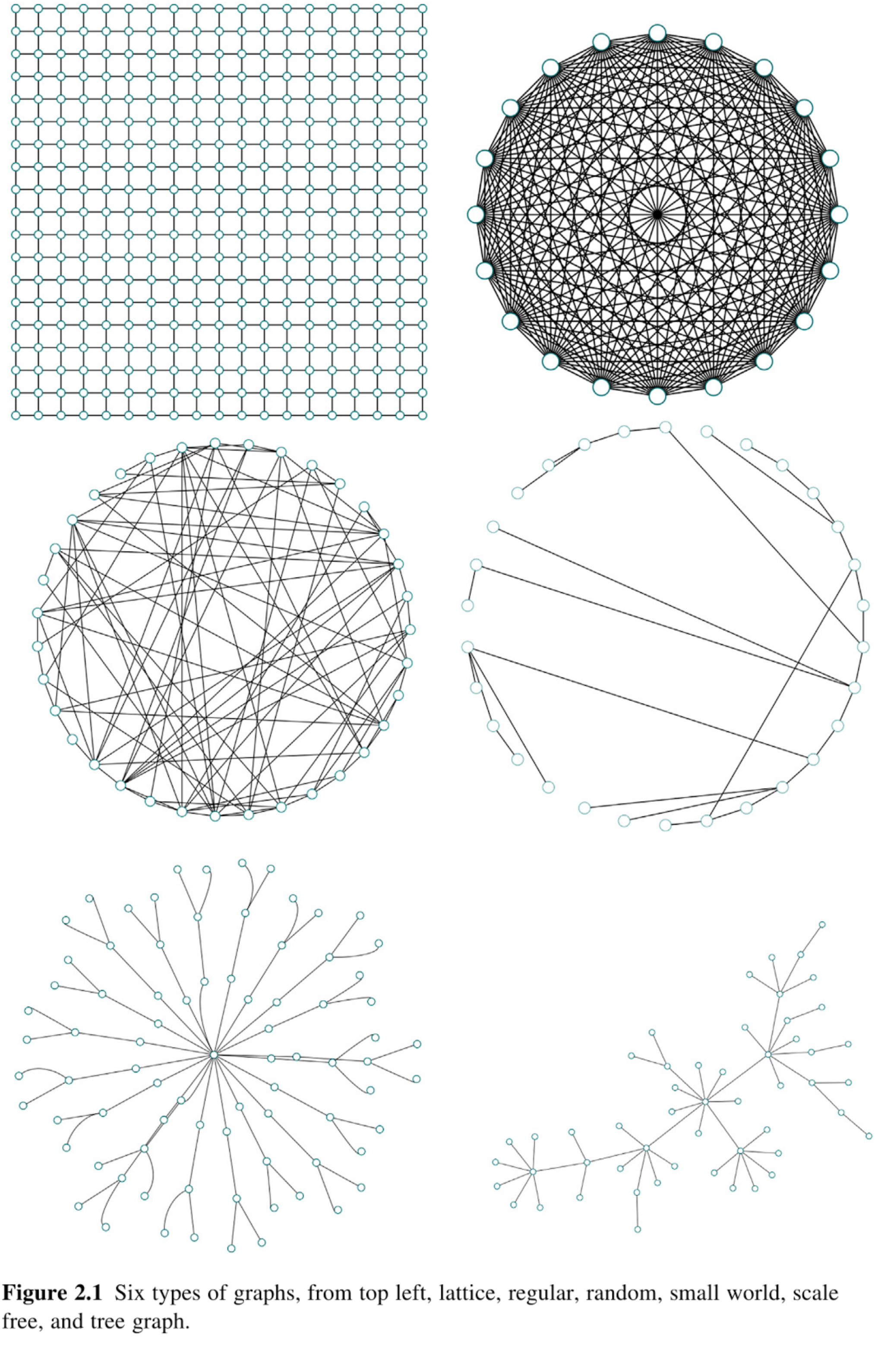

Figure 2.1 Six types of graphs, from top left, lattice, regular, random, small world, scale free, and tree graph.

theorists who study graphs in general tend to focus more on graph-wide metrics, distributions and determining how similar a given graph is to reference graph models such as random, small world, or scale-free network. Researchers in individual disciplines tend to be skeptical of this approach, preferring to focus on groups of nodes, clusters, and the flow of information or patterns of interaction in a localized portion of a graph. Some have even referred to graph-wide metrics and their application to every problem everywhere as a fad.

Both perspectives are valid. If the graph is very large, then certainly localized phenomena may be the only manageable aspect of the graph. If the network models a small and specific aspect of a system or process, then the granularity of the model may be such that graph-wide metrics are useful. Consider the example of a scale-free network. These tend to exhibit fractal-like characteristics, where they are self-similar at different scales. Zoom in and you see a similar pattern of graphs and connections as you do when you zoom out. So a portion of a scale-free network may still contain hubs and lots of nodes with low degree values. Graph-wide metrics are certainly not a fad, but their broad application tends to be. It is not wise to assume they are uniformly reliable predictors of significance for all graphs. That's why it is important to familiarize yourself with as many tools and techniques as possible. This is also why it is useful to understand graph modeling and the domain related to the system that the graph is modeling. This will help you determine how best to analyze your graphs and to interpret the results of that analysis.

There are many ways to represent graphs. In this book, we will explore textual representations such as adjacency matrices, numerous structured formats for documenting nodes and edges in a graph, software models for representing and working with graphs, and various ways of visualizing graph data. All of these things represent graphs, but a graph is a simpler abstraction. All of these techniques boil down to making sense of a system of things and the relationships among them.

The main goal of this book is to provide you with an overview of graph theory, the Semantic Web, and the relation between them. It will help you assemble your own toolkit to explore this exciting field!

This page intentionally left blank

Graphs and the Semantic Web

3

"...memory is transitory"

It is time to go to work, which in this particular future, means moving from the kitchen to your home office, where resides a traditional desk, but with a few upgrades.

Now your desk is retrofitted with things like an "improved microfilm, translucent screen, multiple projection displays, a keyboard, a smattering of specialized buttons," and, finally, even "a lever to simulate page turns." In this environment, one would perform a sort of mechanized cataloging by association—in other words, construct an index organically. And while this particular vision suffers many of the tropes of early twentieth century science fiction, it remains acutely prophetic in anticipating the importance of association in information retrieval systems.

In this future envisioned by Vannevar Bush, every person maintained their own personal library. Already a prescient notion, but not extraordinary, until you become aware of the rest of the vision, which is that each person's library would not be organized alphabetically or numerically or by any other conventional method. Instead, in a Vannevarian future, our personal libraries are organized by a machine (Vannevar called it "the memex") that constructs a unique path among all the items in our collections.

This idea was Vannevar Bush's answer to what he considered a towering shortfall of all current library and indexing approaches: that one of the main barriers to getting at information was "the artificiality of the systems of indexing." As a result, he suggested that our future information systems would be modeled after the associative nature of human thought, resulting in *trails through information that resemble human memory, but with greater permanence*. This is only one of the myriad insights that makes Vannevar Bush's 1943 essay "As We May Think" such a transcendent vision of the future. For it was in this essay, that he wistfully noted that "trails that are not frequently followed are prone to fade, items are not fully permanent, *memory is transitory*." Thus permanence emerges as an essential aspect of the value of information and one area where he thought we could—and eventually, would—improve over memory.

The accelerated speed and efficiency of the Vannevarian memex would make vast amounts of content available at the press of a few buttons. But perhaps his vision's most insightful element is his description of the creation and use of paths through information, which he referred to as *trails*: "Before him are the two items to be joined, projected onto adjacent viewing positions. The user taps a single key, and the items are permanently joined. Out of view, but also in the code space, is inserted a set of dots for photocell viewing; and on each item these dots by their positions designate the index number of the other item." The results, he suggested, would make

A Librarian's Guide to Graphs, Data and the Semantic Web.

"wholly new forms of encyclopedias...appear, ready made with a mesh of associative trails running through them." This mechanism for joining two pieces of content and thus forming "a mesh of associative trails" was not just uncanny, it was highly sophisticated and sounds remarkably like today's Semantic Web.

The RDF model

The Semantic Web defines a model for information called RDF. It is the means by which the Semantic Web facilitates "cataloging by association", as envisioned by Bush. At its core is the notion of a statement which describes an association. A statement has several characteristics. It identifies the thing it is about. It expresses a relationship to another thing. It has a truth value, that is to say, every statement is either true or false. Here are some examples of statements:

Birds have feathers.
The Earth orbits the sun.
Absolute 0 is 0 degrees Kelvin.

An RDF statement is intended to be consumed by software. In other words, it is written in such a way that a computer can unambiguously identify each of its components so that it can relate them to other statements. To avoid ambiguity, the components of statements consist primarily of unique identifiers. A URL is an example of a unique identifier. RDF uses Internationalized Resource Identifiers. IRI looks very much like a URI/URL except that Unicode characters are allowed throughout.

An RDF statement can be modeled in two ways, without changing its meaning or its truth value. You can represent an RDF statement as a triple, which looks like a sentence. A triple consists of a subject followed by a predicate followed by an object. You can also represent as a graph segment consisting of two nodes with a directed edge between them. In the graph version, the start node is the subject (also called "resource"), the target node is the object, and the edge is the predicate pointing from the subject to the object. Thus every RDF statement is a statement of truth regarding some relationship a subject has to an object.

In RDF statements, unique identifiers are used to identify the subject, predicates and sometimes for object values for RDF triples. Like URIs and URLs, an IRI uniquely identifies something. But unlike a URL, it is not required that this IRI resolves to anything. So, if you plug this IRI into a Web browser, you may or may not get anything back.

Here are a couple of examples of IRIs from DBPedia, which is a collection of RDF derived from Wikipedia. One identifies an author, and the second identifies a book:

```
http://dbpedia.org/resource/Cormac_McCarthy
http://dbpedia.org/resource/The_Road
```

The RDF model requires that you be able to relate any subject, predicate or object IRI to a vocabulary. A vocabulary, such as a taxonomy or ontology,

identifies what the IRI is and provides information about its relationship to other items in that vocabulary. You do this with a statement that indicates the IRI's type. The possible values fall into two categories: classes or properties. Classes and properties are defined in formal vocabulary descriptions such as taxonomies or ontologies (see chapter 5). Classes define things. Things have properties. Some classes are subclasses of other classes. That means they have the characteristics of their parent class, and some additional characteristics that set them apart from their parent. The same is true of properties: they can be subproperties of other properties. Both classes and properties have unique IRIs, which are defined in an external vocabulary. Here are two more IRIs from DBPedia. One identifies a class of thing (WrittenWork) and the second identifies a property (author):

```
http://dbpedia.org/ontology/WrittenWork
http://dbpedia.org/property/author
```

With the various identifiers we have seen so far, combined with the requirements we have identified for RDF statements, and the requirement that each statement has a truth value, we can construct an RDF triple that meets all these requirements:

```
http://dbpedia.org/resource/Cormac_McCarthy  http://dbpedia.org/prop-
erty/author http://dbpedia.org/resource/The_Road.
```

This statement says that Cormac McCarthy is the author of The Road, which is in fact true.

In this example every part of the triple statement is an IRI. But an object does not have to be an IRI. An object can be a raw data value such as a number, a date, a string, or some custom data type. For raw data values, it is important to specify the type of data, again, to reduce ambiguity. The Semantic Web uses the same data types defined in XML. Core types are xsd:string, xsd:boolean, xsd:decimal, and xsd:double, but there are many others for types of numbers, dates, binary data, and data encoded in other formats. So, we could have expressed the triple about Cormac McCarthy like this:

```
http://dbpedia.org/resource/Cormac_McCarthy  http://dbpedia.org/prop-
erty/author "The Road".
```

This is the only exception to RDF's IRI requirement. It is best to use an IRI even in the object portion if possible, because IRIs uniquely and unambiguously identify a thing. Since RDF lives on the World Wide Web, you can achieve this uniqueness in the same way that the Web ensures every page URL is unique, starting with the name of the computer that does (or could) host this RDF data. In fact, it is good practice to make RDF IRIs work. That is, if you type one in a Web browser, the browser would retrieve some information about the thing represented by the identifier. This is a requirement if you want to add your data to the Web of Linked Open Data, a vast collection of RDF data that is accessible on the Web (Figure 3.1).

Figure 3.1 A single RDF statement visualized as a graph.

Modeling triples

Even though triples are not a linguistic construct, there are instances when it is helpful to think of them as simple sentences. For example, if you are prototyping some information as triples with colleagues, you can use the triple structure as a model for that information. This is a good way to explore how to represent information in RDF without initially identifying a source vocabulary for predicates.

It is not uncommon to play fast and loose with RDF when first modeling your data as triples. Let us say you wanted to create a triple about a boy named Paul who is 9 years old. You do not yet know what ontologies or vocabularies might apply, you are just trying to outline things and relationships. So you just write on a whiteboard: aPerson/NewMexico/Paul hasAge 9. Or, maybe you start by drawing a pair circles, labeling one Paul, the other 9, and draw a line between them labeled "hasAge" These are both RDF models. You won't have a problem creating formal valid triples from them, even if you have to define your own vocabulary to get there.

This is a bottom-up approach to modeling semantic data in RDF. Eventually you have to select vocabularies, create IRIs, make links, and create triples. The resulting triples are called instance data. You can think of instance data as rows in a database table. The vocabularies you use in your instance data roughly correspond to the database schema, which defines the tables and relationships in a relational database. In the example above, the predicate "hasAge" could be a data property from a taxonomy or ontology about people. The statement that Paul hasAge 9 is instance data, a statement of fact about something identified as "Paul."

RDF graphs can be analyzed just like any other graphs, although applying graph analytic techniques to RDF can be a little tricky. One reason is because an RDF graph is a multipartite graph. This means there are different types of nodes and edges within the same graph. Summing up the edges that point to an IRI identifying something will tell you how many statements of some form have been made about that thing, but this rarely leads to insights about an RDF graph. Usually it is necessary to extract a portion of an RDF graph, thus creating some kind of subgraph, before something like degree is a useful metric. So graph analysis of RDF graphs tends to require more work and is not always a useful way to explore the information they contain.

RDF and deduction

You can evaluate and analyze the content of RDF graphs in other ways. A triple is an assertion of fact about a resource (the subject). It has a truth value. Most people can, when presented with a set of related facts can reason about the information and draw conclusions. We can deduce new facts from what we know to be true. The

RDF model is rigorous enough that it is possible to construct rules that a computer could use to reason about triples and deduce new facts. These rules are represented as a series of logical statements. You can express things like "if condition a and condition b are true about resource x, then we can conclude that something else is also true." If you have ever performed a boolean search, then you have at least a passing familiarity with formal logic. Consider the following statements about Bob:

"Bob has daughter"
"Bob has wife"
"Wife is mother."

A human would quickly reason through these statements to conclude that a daughter has a mother. We of course have the advantage of knowing what daughter, wife, and mother are, whereas a computer program does not. Because RDF uses unique identifiers for resources, common shared vocabularies, and because each triple is a statement of a fact, RDF is well suited for rules-based reasoning. You can construct rules that a computer could use to determine that there is an implicit relationship between a daughter and a mother:

if (subject has daughter) or (subject has son)
  and if (subject has wife) and (wife is mother)
    then daughter has mother.

These rules make it possible a infer a new fact if certain conditions are met. This fact can be added as a new edge to the graph, between daughter and mother, with an edge labeled "has." This process is called reasoning. Reasoning over RDF statements can result in the discovery of new facts or allow software to "make decisions" based on how a set of logical rules evaluate, given some collection of facts. In a sense, triples represent a sort of "memory" and rules are a very crude strategy for "thinking."

Graphs are usually used to model complex but well-defined systems. These kinds of graphs have a finite set of nodes and edges of one or a few types. In contrast, RDF is graph-based scaffolding for representing all human knowledge. It is a network representation of information, through which many paths are possible. It can grow and evolve over time as resources and relationships are added. In a sense, it is a vast collective memory.

With the Semantic Web, our machines not only gain an associative memory like ours, they can make use of it. Vannevar Bush, and others before him, anticipated that machines might eventually be able to reason over linked knowledge: "Whenever logical processes of thought are employed—that is, whenever thought for a time runs along an accepted groove—there is an opportunity for the machine. It is readily possible to construct a machine which will manipulate premises in accordance with formal logic, simply by the clever use of relay circuits. Put a set of premises into such a device and turn the crank, and it will readily pass out conclusion after conclusion, all in accordance with logical law, and with no more slips than would be expected of a keyboard adding machine."

This page intentionally left blank

RDF and its serializations

4

Abstract notions lead to shared concepts

In the 1880s, naturalist Charles Robertson began observing and recording the visits bees made to various flowering plants within a 10 mile vicinity of the town of Carlinville in northwestern Illinois. He finally published these observations in his 1923 book, *Flowers and Insects: Lists of Visitors of Four Hundred and Fifty-Three Flowers.* Robertson's work was so meticulous and thorough that it established a baseline of bee diversity for this small area. It's not hard to understand why this sort of research is valuable, given the precipitous decline in pollinating insects that have been widely reported in recent years.

The existence of exhaustive, reliable baseline observations of the biodiversity for any given area is rare. Such observations are especially useful in evaluating the impact that human activities may have on local biodiversity, as well as revealing deeper truths about universalities of all ecosystems. Researchers have periodically replicated Robertson's observations over time, most recently in 2009–2010 by researchers who subsequently published their findings in *Science* Magazine in a paper entitled "Plant-Pollinator Interactions over 120 Years: Loss of Species, Co-Occurrence, and Function." In this more recent paper, the investigators developed a simple graph model of the current and past observations, with insects and plants serving as nodes, and visitations representing edges between them.

The authors used simple visual cues to differentiate graph elements. Temporal distinctions were indicated using colors, with past edges colored red, and current edges colored blue. Edges appeared thicker when there are multiple visits recorded between a particular species of bee and a particular species of plant. This graph, and simple visualization made it readily apparent that there has been a loss of biodiversity in this area over time. It provided some evidence that other species were replacing lost bees, but it also illustrated that there is limited resilience in the bee–plant network. This research and the resulting graph model are a great example of insights that graph models can yield.

RDF graph

The plant-pollinator graph is very simple. There is only one relationship: visits, and two types of things: a species of bee and a species of plant. This is a situation where either a graph or a Semantic Web model would work equally well. In this case, the authors used a simple graph model. But, in a paper on a similar topic entitled "A Case-Study of Ontology-Driven Semantic Mediation of Flower-Visiting Data from Heterogeneous Data-Stores in Three South African Natural History Collections," the authors proposed to extend an existing ontology to represent similar data in

RDF. They proposed extensions to Darwin-SW, which is "an ontology using Darwin Core terms to make it possible to describe biodiversity resources in the Semantic Web" to describe the various interactions that insects have with plants. Their extension makes RDF statements like this possible:

```
http://dbpedia.org/page/Bumblebee Darwin-SW:participates_in FlowerVisitRole.
```

This is essentially the same relationship as expressed in the plant-pollinator graph, but it uses a semantic model. One of the more interesting problems associated with creating graph representations of a system of things and the relationships among them is the task of developing a comprehensive, yet flexible graph model. Graph modeling is a topic we will revisit many times, but the bottom line is, there is no wrong way to do it.

Sometimes there may be a particular approach to modeling the data that seems more intuitive, for example, if you want to represent a social network, people will be the nodes, and some aspect of their relationships will become edges. This can be modeled as either a simple object-relationship graph or an RDF graph. Perhaps you want to add weights to the graph edges according to how well two people know one another. Labeling graph models is very common. It provides a way to distinguish node types, individual nodes, and edges. Graphs where this practice is formalized are called property graphs. In a property graph, nodes and edges can have label (key) value pairs that document details for that individual graph element. This allows the model to be focused on things and relationships. At the same time, it ensures that you can find out what kind of things and what types of relationships an individual node or edge represents when you need that information. Thus, a property graph is very much an open-ended construct with few constraints. This makes them well suited for modeling various aspects of a great variety of complex systems with relative ease (Figure 4.1).

To make triples work with the World Wide Web, RDF graphs use IRIs, which both represent a discrete thing and provide a pointer to it. In the example above, http://dbpedia.org/page/Bumblebee is a DBPedia IRI for the "thing" bumblebee. This identifier points at other related data about bumblebees in other RDF graphs. It is part of the Web of linked data. Other portions of a triple often link out to other graphs. An RDF triple predicate, which is an edge between two nodes in an RDF graph, is also a node defined in an ontology. An ontology is a graph model for a vocabulary. Subjects and some objects are usually of a type (called a class) defined in an ontology. Ontologies define types of things, their relationships to other things, and the properties that are applicable to those things. The approach for defining ontologies borrows heavily from another branch of mathematics called first-order logic. First-order logic allows you to express statements about what is true (which is what a triple is) and to represent the logical steps that can be used to reason about these facts. The reason for this (pun intended) is that reasoning can lead to the deduction of new facts or to conclusions based on combinations of facts.

Ontologies build on the concept of taxonomies. A taxonomy is a sometimes hierarchical list of terms describing a domain or a type of thing. Consider a taxonomy for animals: at the top is the term "animal." Children of the term animal include "mammal," "reptile," "fish," and "bird." Mammals might then be subdivided into

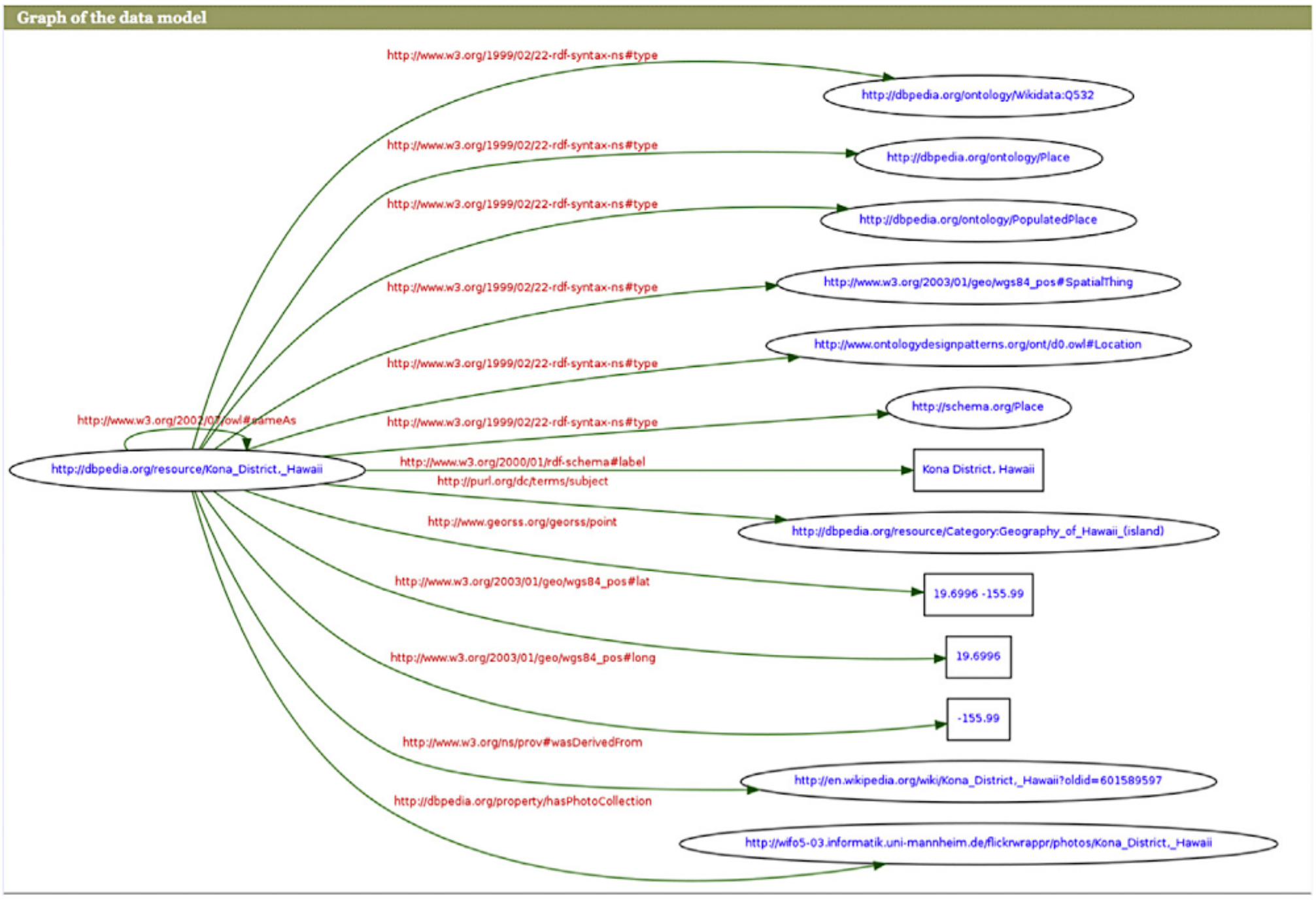

Figure 4.1 A graph visualization of RDF data with the subject at left, and predicate edges connecting it to objects on the right.

"carnivores," "omnivores," and "herbivores." An ontology takes this concept further by allowing for relationships among terms. You can express the fact that an animal cannot be both a reptile and a mammal, and that reptiles, bird, fish, and mammals can all be carnivores, omnivores, or herbivores. Once you define your ontology, you can publish it and others can use and refer back to it. If two different triplestores contain facts about animals, they can both reference the animal ontology and base their predicates on terms from that ontology by using the appropriate IRIs for the terms they wish to use. At that point the concept of animal and the characteristics of an animal are shared across those collections. In effect, they share a basic concept of what an animal is! Chapter 5 covers ontologies in greater detail. Many of the formats described in this chapter can also be used to represent ontologies.

RDF defines an abstract model representing information. The RDF model identifies the parts of a triple, specifies the role of ontologies, when IRIs are required, when literal values like numbers and strings are allowed. However, RDF doesn't tell you exactly how to write a triple. This is the role of serializations. A serialization describes a textual representation for information or data. Serializations like RDF/XML, N3, and Turtle define a particular syntax for RDF statements. As you will see, these various serializations all have their strengths and weakness. Just remember that the RDF model does not change, whether you represent your triples as RDF/XML, N3, or in some other format (Figure 4.2).

Another way to think about serializations is to compare RDF with time. We have numerous ways of representing time (e.g., clocks, calendars, etc.), and the

W3C RDF Validation Service

Home Documentation Feedback

Jump To: Source Triples Messages Graph Feedback Back to Validator Input

Validation Results

Your RDF document validated successfully.

Triples of the Data Model

Number	Subject	Predicate	Object
1	http://dbpedia.org/resource/Kona_District,_Hawaii	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://dbpedia.org/ontology/Wikidata:Q532
2	http://dbpedia.org/resource/Kona_District,_Hawaii	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://dbpedia.org/ontology/Place
3	http://dbpedia.org/resource/Kona_District,_Hawaii	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://dbpedia.org/ontology/PopulatedPlace
4	http://dbpedia.org/resource/Kona_District,_Hawaii	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing
5	http://dbpedia.org/resource/Kona_District,_Hawaii	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.ontologydesignpatterns.org/ont/d0.owl#Location
6	http://dbpedia.org/resource/Kona_District,_Hawaii	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://schema.org/Place
7	http://dbpedia.org/resource/Kona_District,_Hawaii	http://www.w3.org/2002/07/owl#sameAs	http://dbpedia.org/resource/Kona_District,_Hawaii
8	http://dbpedia.org/resource/Kona_District,_Hawaii	http://www.w3.org/2000/01/rdf-schema#label	"Kona District, Hawaii"@en
9	http://dbpedia.org/resource/Kona_District,_Hawaii	http://purl.org/dc/terms/subject	http://dbpedia.org/resource/Category:Geography_of_Hawaii_(island)
10	http://dbpedia.org/resource/Kona_District,_Hawaii	http://www.georss.org/georss/point	"19.6996 -155.99"
11	http://dbpedia.org/resource/Kona_District,_Hawaii	http://www.w3.org/2003/01/geo/wgs84_pos#lat	"19.6996"^^http://www.w3.org/2001/XMLSchema#float
12	http://dbpedia.org/resource/Kona_District,_Hawaii	http://www.w3.org/2003/01/geo/wgs84_pos#long	"-155.99"^^http://www.w3.org/2001/XMLSchema#float
13	http://dbpedia.org/resource/Kona_District,_Hawaii	http://www.w3.org/ns/prov#wasDerivedFrom	http://en.wikipedia.org/wiki/Kona_District,_Hawaii?oldid=601589597
14	http://dbpedia.org/resource/Kona_District,_Hawaii	http://dbpedia.org/property/hasPhotoCollection	http://wifo5-03.informatik.uni-mannheim.de/flickrwrappr/photos/Kona_District,_Hawaii

The original RDF/XML document

```
1: <?xml version="1.0" encoding="utf-8" ?>
2: <rdf:RDF
3:      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```

Figure 4.2 W3C's RDF Validator output.

same is true for RDF. Representing RDF using different serializations doe not change the original meaning of the statements. All RDF serializations conform to the rules of RDF for the formulation of triples. As a result, it is possible to create converters from one serialization to another. It is also possible to validate a set of statements, to determine if they do in fact conform to the syntactic rules governing that serialization format. Different serializations were developed with different purposes in mind. Some are better at representing RDF in a form that people can understand, others are well suited for embedding RDF in Web pages, and others work best for programs that exchange RDF data.

RDF serializations

RDF/XML co-opts existing tools and standards to express triples using XML. It is the standard interchange format for Semantic Web data and so it is a required format for triples and ontologies. RDF/XML is very difficult to read. Turtle, which stands for Terse RDF Triple Language, is concise, easy to read representation for triples. It is basically a variant of another syntax for representing triples, called Notation 3 or N3. Functional syntax is modeled after logic notation and is especially well suited for defining ontologies and rules. JSON-LD is the JSON (Javascript Object Notation) Linked Data format. Like RDF/XML, it leverages an existing standard (JSON) to facilitate machine exchange of semantic data. Unlike

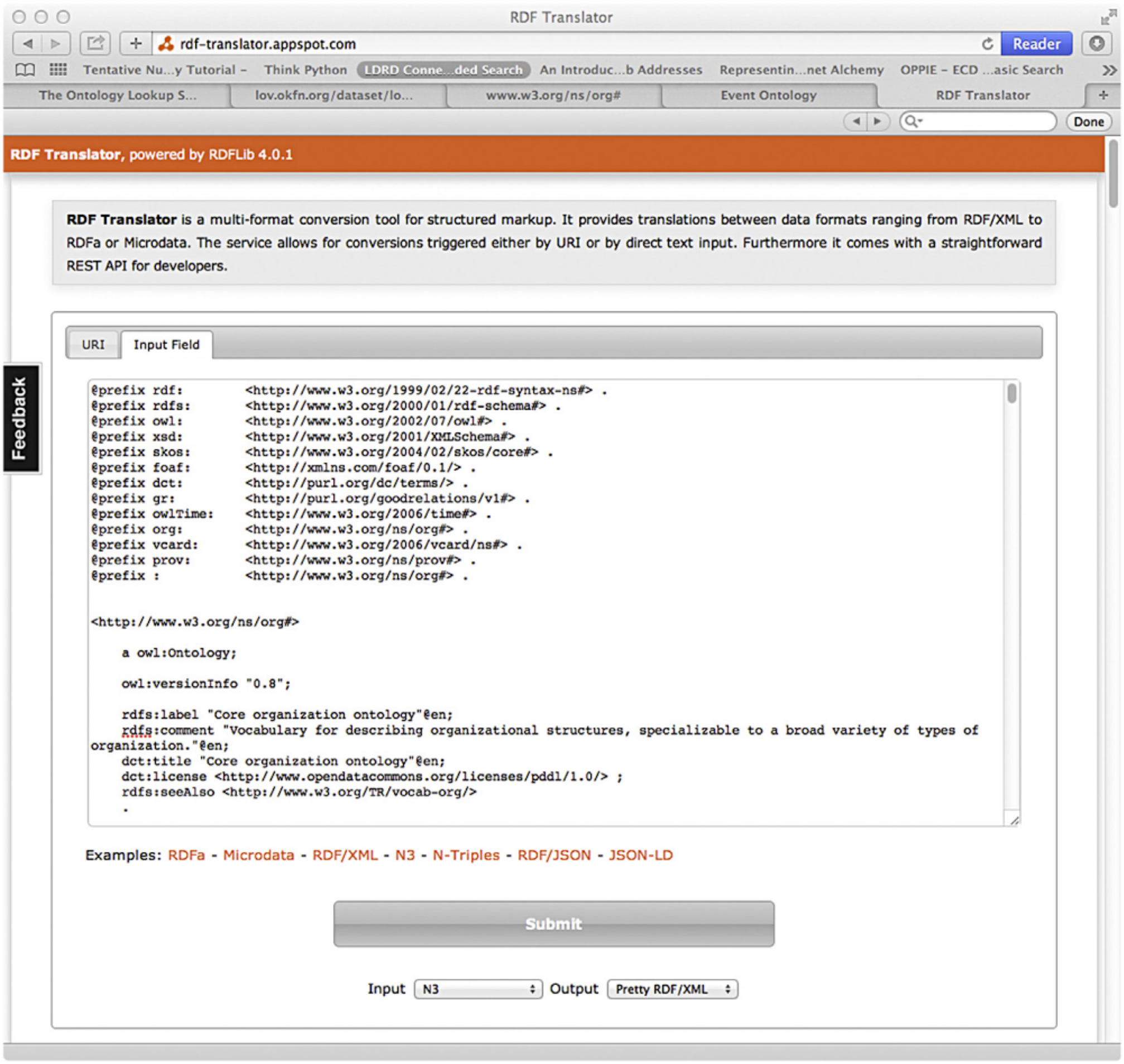

Figure 4.3 RDF Translator displays a portion of a Turtle/N3 document.

RDF/XML, it is actually pretty easy to understand. RDFa is a format for embedding triples into HTML Web pages (Figure 4.3).

Turtle is an RDF serialization that is well suited for human comprehension. It uses some simple syntactic rules also found in English. The resulting expressions of triples, where punctuation such as commas and semicolons are used to combine multiple statements, can almost seem at times like English language sentences. A straightforward representation of a triple in turtle consists of an IRI subject, an IRI predicate, and an IRI or literal object value followed by a period. An IRI is always contained within these characters < >. Turtle allows you to predefine a set of prefixes that correspond to the vocabularies and ontologies that subsequent triples reference. Each prefix definition starts with @prefix. The string that constitutes a prefix for a given namespace follows. This is followed by a colon and then the IRI for the namespace, embedded in < >. When you use a prefix in a triple, you specify the prefix, followed by a colon, followed by a specific element name from the vocabulary (class or property name). If you

want to make two statements about the same thing then the first statement concludes with a semicolon, and is followed by a second predicate and object, which ends with a period. Object values represented as literal strings are contained within quotes.

```
@prefix owl:  <http://www.w3.org/2002/07/owl#>.
@prefix ns1:  <http://wikidata.dbpedia.org/resource/>.
@prefix ns2:  <http://id.dbpedia.org/resource/>.
ns2:Distrik_Kona  owl:sameAs  <http://dbpedia.org/resource/Kona_District,_Hawaii>.
@prefix dbpprop:  <http://dbpedia.org/property/>.
@prefix dbpedia:  <http://dbpedia.org/resource/>.
<http://en.wikipedia.org/wiki/Kona_District,_Hawaii>  foaf:primaryTopic
  <http://dbpedia.org/resource/Kona_District,_Hawaii>.
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix geo:  <http://www.w3.org/2003/01/geo/wgs84_pos#>.
<http://dbpedia.org/resource/Kona_District,_Hawaii>  rdf:type  geo:SpatialThing.
@prefix d0:  <http://www.ontologydesignpatterns.org/ont/d0.owl#>.
<http://dbpedia.org/resource/Kona_District,_Hawaii>  rdf:type  d0:Location.
@prefix dcterms:  <http://purl.org/dc/terms/>.
<http://dbpedia.org/resource/Kona_District,_Hawaii>  dcterms:subject
  <http://dbpedia.org/resource/Category:Geography_of_Hawaii_(island)>.
@prefix grs:  <http://www.georss.org/georss/>.
<http://dbpedia.org/resource/Kona_District,_Hawaii>  grs:point  "19.6996 -155.99".
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>.
<http://dbpedia.org/resource/Kona_District,_Hawaii>
     geo:lat  "19.6996"^^xsd:float;
    geo:long  "-155.99"^^xsd:float;
     foaf:depiction
   <http://commons.wikimedia.org/wiki/Special:FilePath/HawaiiIslandDistrict-
NorthKona.svg>.
@prefix prov:  <http://www.w3.org/ns/prov#>.
<http://dbpedia.org/resource/Kona_District,_Hawaii>    prov:wasDerivedFrom
   <http://en.wikipedia.org/wiki/Kona_District,_Hawaii?oldid=601589597>;
   foaf:isPrimaryTopicOf  <http://en.wikipedia.org/wiki/Kona_District,_Hawaii>;
   dbpprop:hasPhotoCollection  <http://wifo5-03.informatik.uni-mannheim.de/flickrwrappr/
photos/Kona_District,_Hawaii>;
```

RDF/XML is a way to represent triples using an XML schema. XML is an especially useful format for providing triples in batch to software because processing it leverages existing software that can process XML. An XML document contains data wrapped with special tags enclosed between the < and > characters. When a tag contains some data, it has a start form and an end form <tag> </tag>. Sometimes it is useful to have an empty tag which looks like this <tag />. Sometimes tags include attributes, which express some additional information. They are contained within the start element tag and take the form attribute = "some value". Sometimes tags can be nested inside other tags. Tags are defined in XML schema or XML DTD documents, which are just two ways to define tag names and tag relationships, as well as the overall structure of an XML document that uses the defined tags. Documents containing data tagged based upon a particular tag set are

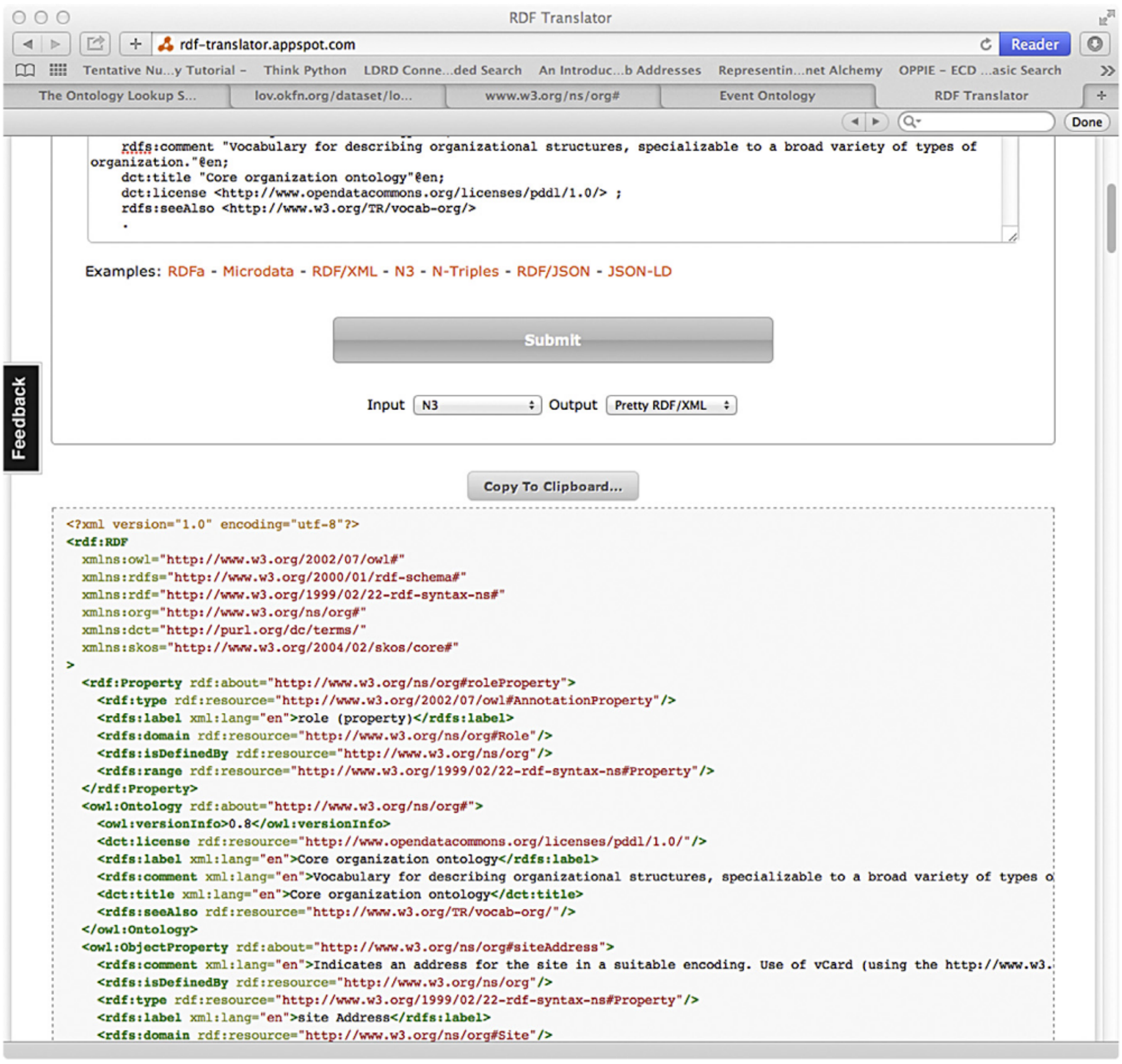

Figure 4.4 Turtle/N3 that has been converted to RDF/XML using the RDF Translator.

called instance documents or instance data. Triples expressed in RDF/XML conform to all of these requirements: there is a schema defining the triple tag and attribute set, and there are instance documents containing sets of triples tagged with RDF/XML (Figure 4.4).

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:dcterms="http://purl.org/dc/terms/"
    xmlns:grs="http://www.georss.org/georss/"
    xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#"
    xmlns:dbpedia-owl="http://dbpedia.org/ontology/"
```

```
xmlns:dbpprop = "http://dbpedia.org/property/"
xmlns:foaf = "http://xmlns.com/foaf/0.1/"
xmlns:prov = "http://www.w3.org/ns/prov#" >
< rdf:Description rdf:about = "http://dbpedia.org/resource/Kona_District,_Hawaii" >
< rdf:type rdf:resource = "http://dbpedia.org/ontology/Wikidata:Q532" / >
< rdf:type rdf:resource = "http://dbpedia.org/ontology/Place" / >
< rdf:type rdf:resource = "http://dbpedia.org/ontology/PpopulatedPlace" />
< rdf:type rdf:resource = "http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing" / >
< rdf:type rdf:resource = "http://www.ontologydesignpatterns.org/ont/d0.owl#Location" / >
< rdf:type rdf:resource = "http://schema.org/Place" / >
< owl:sameAs rdf:resource = "http://dbpedia.org/resource/Kona_District,_Hawaii" / >
< rdfs:label xml:lang = "en" > Kona District, Hawaii < /rdfs:label >
< dcterms:subject rdf:resource = "http://dbpedia.org/resource/Category:Geography_of_Hawaii_
(island)" / >
< grs:point > 19.6996 -155.99 < /grs:point >
< geo:lat rdf:datatype = "http://www.w3.org/2001/XMLSchema#float" > 19.6996 < /geo:lat >
< geo:long rdf:datatype = "http://www.w3.org/2001/XMLSchema#float" > -155.99 < /geo:long >
< prov:wasDerivedFrom rdf:resource = "http://en.wikipedia.org/wiki/Kona_District,_Hawaii?
oldid = 601589597" / >
< dbpprop:hasPhotoCollection rdf:resource = "http://wifo5-03.informatik.uni-mannheim.de/
flickrwrappr/photos/Kona_District,_Hawaii" / >
```

Functional syntax places the predicate before the subject and object values. It is most often used to describe ontologies but it works equally well for instance triples. If you are at all familiar with programming, you can think of it this way: the predicate becomes a function, and the subject and object are parameters passed to that function. As with other RDF serializations, you can define a default namespace and optional prefixes which serve as abbreviations for other vocabularies used by your triples.

```
Prefix(:=<http://www.example.com/ontology1#>) Ontology(
<http://www.example.com/ontology1>
    Import( <http://www.example.com/ontology2> )
    Annotation( rdfs:label "An example" )
      SubClassOf( :Child owl:Thing ) )
```

Notation 3, or N3, is similar to turtle. It differs primarily in how it handles blank nodes. Like N-triples, you have the option of defining prefixes that stand in for the namespaces for the various vocabularies and ontologies a particular collection of triples use. When a prefix is used subsequently, it is preceded by a colon. Blank nodes are denoted by an underscore character and the internal identifier assigned to the node. Turtle uses square brackets and omits the blank node label altogether. Otherwise the syntax for N3 is the same as Turtle.

JSON-LD uses JSON (Javascript Object Notation) to represent RDF data. Javascript is a programming language for the Web which runs in Web browsers. The LD portion of the name stands for Linked Data. Because of JSON's Web heritage, JSON-LD is well suited for Web applications written in Javascript. But JSON has become a more general purpose data interchange format. Mongo is a big data database that uses JSON natively. Many REST-based Web applications, which are

client−server applications that use Web protocols, send and receive JSON formatted data to each other. JSON defines a very flexible key/value pair format for representing various data structures. Additional syntactic rules allow for data to be grouped in various ways. JSON syntax for an object is to enclose the object components within curly brackets. Each key and value is contained within double quotes and the two are separated by a colon. Key/value pairs are separated by a comma. JSON-LD defines a JSON syntax for RDF data. JSON allows objects to be chained together, and this is how JSON-LD specifies things like multiple predicates and objects for a single subject. A JSON-LD document can define a local vocabulary using context, and it can reference external vocabularies as well. These occur at the beginning of a JSON-LD document

```
{ "@graph": [
   { "@id": "http://dbpedia.org/resource/Kona_District,_Hawaii",
     "@type" : [ "http://dbpedia.org/ontology/Place" ,
       "http://schema.org/Place" ,
       "http://dbpedia.org/ontology/PopulatedPlace" ,
       "http://www.ontologydesignpatterns.org/ont/d0.owl#Location" ,
       "http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing" ,
       "http://dbpedia.org/ontology/Wikidata:Q532" ] ,
       "http://id.dbpedia.org/resource/Distrik_Kona" ,
       { "@value" : "Kona District, Hawaii" , "@language" : "en" } ] ,
       "http://purl.org/dc/terms/subject" : [
"http://dbpedia.org/resource/Category:Geography_of_Hawaii_(island)" ] ,
       "http://www.georss.org/georss/point" : [ { "@value" : "19.6996 -155.99" } ],
"http://www.w3.org/2003/01/geo/wgs84_pos#lat":[{"@value":
       19.69960021972656,"@type":"http://www.w3.org/2001/XMLSchema#float"}],
"http://www.w3.org/2003/01/geo/wgs84_pos#long":[{"@value":-
       155.9900054931641,"@type":"http://www.w3.org/2001/XMLSchema#float"}],
"http://www.w3.org/ns/prov#wasDerivedFrom" : [
       "http://en.wikipedia.org/wiki/Kona_District,_Hawaii?oldid=601589597" ] ,
```

RDFa defines a handful of HTML attributes that can be used with existing HTML markup elements to encode RDF into various contexts such as Web or XML pages. Some aspects of RDFa are implicit. For example, it is understood that an RDFa attribute, when it represents a portion of an RDF statement, is generally referring to the child element or the HREF attribute value, if applicable, of a document. The vocab attribute allows you to define one or more ontology namespaces that will be used in the HTML markup of your Web page. The prefix attribute allows you to define a string prefix for each namespace, just as with other RDF serializations, to designate the ontology for a given property. The property attribute associates the contents of the element it is embedded in with an ontology property. If you want to indicate that something is of a particular class, there is the type of attribute.

Finally, the resource attribute allows you to identify the subject of an RDFa embedded statement explicitly. Full documentation for RDFa core and RDFa lite is available at the W3 website. A similar format called HTML + microdata, which also uses attributes to embed RDF in a Web page. In this case, the itemscope and itemid attributes identify the subject for one or more triples. The item prop and href

attributes identify predicate and their associated object values for the specified itemscope subject, until a new itemscope is specified.

```
<dl itemscope itemid="http://dbpedia.org/resource/Kona_District,_Hawaii">
<dt>Subject Item</dt><dd>n2:_Hawaii</dd>
<dt>rdf:type</dt><dd>
<a itemprop="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
    href="http://dbpedia.org/ontology/Place">dbpedia-owl:Place</a>
<a itemprop="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
    href="http://schema.org/Place">n9:Place</a>
<a itemprop="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
    href="http://dbpedia.org/ontology/PopulatedPlace">dbpedia-owl:PopulatedPlace</a>
<a itemprop="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
    href="http://www.ontologydesignpatterns.org/ont/d0.owl#Location">d0:Location</a>
<a itemprop="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
    href="http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing">geo:SpatialThing</a>
</dd>
<dt>dcterms:subject</dt><dd>
<a itemprop="http://purl.org/dc/terms/subject" href="http://dbpedia.org/resource/Category:
Geography_of_Hawaii_(island)">n28:</a>
</dd>
<dt>grs:point</dt><dd>
<span itemprop="http://www.georss.org/georss/point">
    19.6996 -155.99</span>
</dd>
<dt>geo:lat</dt><dd>
<span itemprop="http://www.w3.org/2003/01/geo/wgs84_pos#lat">
    19.6996</span>
</dd>
<dt>geo:long</dt><dd>
<span itemprop="http://www.w3.org/2003/01/geo/wgs84_pos#long">
    -155.99</span>
</dd>
<dt>dbpprop:hasPhotoCollection</dt><dd>
<a itemprop="http://dbpedia.org/property/hasPhotoCollection"
    href="http://wifo5-03.informatik.uni-mannheim.de/flickrwrappr/photos/Kona_District,
    _Hawaii">n5:_Hawaii</a>
</dd>
<dt>foaf:depiction</dt><dd>
<a itemprop="http://xmlns.com/foaf/0.1/depiction" href="http://commons.wikimedia.org/wiki/
Special:FilePath/HawaiiIslandDistrict-NorthKona.svg">n24:svg</a>
</dd>
<dt>prov:wasDerivedFrom</dt><dd>
<a itemprop="http://www.w3.org/ns/prov#wasDerivedFrom" href="http:// en.wikipedia.org/wiki/
Kona_District,_Hawaii?oldid=601589597">n11:601589597</a>
</dd>
```

Ontologies

5

Ontological autometamorphosis

The Semantic Web has its origins in the decades long struggle to enable computers (which until recently lacked means for perceiving or manipulating the world) to understand and interact with information as we do. A similar struggle for understanding occurs in the 1961 scifi novel *Solaris*, by Stanislaw Lem. In *Solaris* humankind had discovered alien life on the planet Solaris. Solaris was inhabited by a single organism that encompassed the entire planet as would an ocean. Despite a hundred years of effort and study (much of which was documented in imaginary books and journals which populated the library of the Solaris outpost), neither human nor alien had managed to communicate with the other. The inhabitant of Solaris was made up of a strange viscous liquid and had a tendency to mimic shapes of things that it perceived or encountered. In the novel, some scientists who studied Solaris suggested that the alien was engaging in a process they called ontological autometamorphosis; that is, in order to perhaps comprehend things, it took their form for a time. Fortunately for us, we can engage in a similar process as purely a mental exercise and document and share the results with others, by using our innate classification skills to create Semantic Web ontologies.

It is perhaps fitting that "ontology" is a vaguely alien sounding word that is a bit hard to define. With ties to both philosophy and linguistics, ontology is the embodiment of something that comes naturally for humans: categorizing and classifying things around us. We categorize to help us recognize things and to predict their behavior. We categorize so we can make guesses about new things that we encounter. Very young children sometimes use a simply classification strategy when learning new words, referred to as a taxonomic assumption. Stanford psychology professor Ellen Markman explains how this works:

> *[we] conducted a series of studies which compared how children would organize objects when an object was referred to with a novel label versus when it was not. When presented with two objects, such as a dog and cat, and a third object that was thematically related such as dog food, children would often select a dog and dog food as being the same kind of thing. If, however, the dog was called by an unfamiliar label such as* dax *and children told to find another dax, they now were more likely to select the cat. This illustrates the basic phenomenon: When children believe they are learning a new word, they focus on taxonomic, not thematic, relations.*

It is often challenging to unambiguously express the structure of a domain of knowledge and facts within it to another person, let alone an alien. Imagine that you had to describe the desk you are perhaps sitting in front of right now. How would you describe this particular desk to someone who has never seen it?

How would you describe a desk in general terms to someone who didn't know what a desk was? How would you describe the category of things we refer to as furniture to a lost tribe of Amazonian Indians who had never been contacted by the outside world? In each case, there are probably different characteristics you would emphasize first, in order to provide a context for the more specific descriptive features. In cases where the target audience's context is less similar to yours, you would build on these descriptions in order to provide all the necessary information. One way to do this is by developing an ontology.

Introduction to ontologies

In this chapter we have four main goals:

1. To convey the thought process for constructing an ontology
2. To give you a sense for what roles ontologies play in the Semantic Web
3. To introduce you to the ontology description languages RDFS and OWL
4. To help you be able to read and understand an ontology when you find one online

An ontology is, in a sense, a skeletal framework for knowledge. An ontology is a description of things, relationships, and their characteristics, usually in a well-bounded domain, for example, ecology or astronomy. It is in part a taxonomy, which is a graph structure that describes the hierarchical relationship of a group of things. Things that occur farther down in the taxonomy inherit characteristics from one or more parents that occur above them. In an ontology, the taxonomy is extended so that you express logical relationships among things, membership in groups, multiple inheritance relationships, the symmetry of relationships, exclusiveness, and various characteristics of a given thing.

Semantic Web ontologies are concerned with logical consistency and expressiveness to allow for machine reasoning. Semantic Web ontology languages employ first-order predicate logic. Logic is a mathematical field concerned with expressing simple statements of fact as axioms, and for expression rules as logical expressions, which, given some initial conditions, can be evaluated against the axioms using inferencing and deduction. Since ontologies are grounded in logic, they can facilitate a process that mimic human reasoning. This is how ontologies can be used in artificial intelligence applications.

In this chapter, we introduce the two languages used to create Semantic Web ontologies, RDFS and OWL. Like RDF, the description of ontology can be serialized in various formats, including N3, functional logic, and RDF/XML. Only RDF/XML is required, and only if you plan to publish and share your ontology with others. The RDF/XML document describing your ontology should be accessible via the IRI that corresponds to its unique namespace. This IRI is used to reference the ontology when its classes and properties are used in RDF triples. In triples, elements of an ontology typically occupy the predicate portion of a triple. In some

cases, they can also be subjects (for example, in the ontology definition file itself), or objects, if the statement is expressing that some item is of a type defined in an ontology, for example. In other words, the edges in RDF graphs are usually properties defined in some ontology. In fact, the use of ontologies is one of the defining characteristics of the Semantic Web.

While it is important to be able to read and comprehend an ontology, in practice, you will rarely need to create an entirely new ontology from scratch. There are Web and desktop ontology authoring tools that handle some of the complexity of generating an ontology and make it easier to focus on your model. Protege is one example. You can use Protege to create an ontology from scratch, extend an existing ontology, or simply to load an existing ontology to browse it. It enforces RDFS and OWL language requirements, and can even be used to generate instance triples using your ontology. There's also a Web version of Protege called WebProtege. If you have a basic knowledge of RDFS and OWL, you will be able to more effectively use any ontology authoring tool (Figure 5.1).

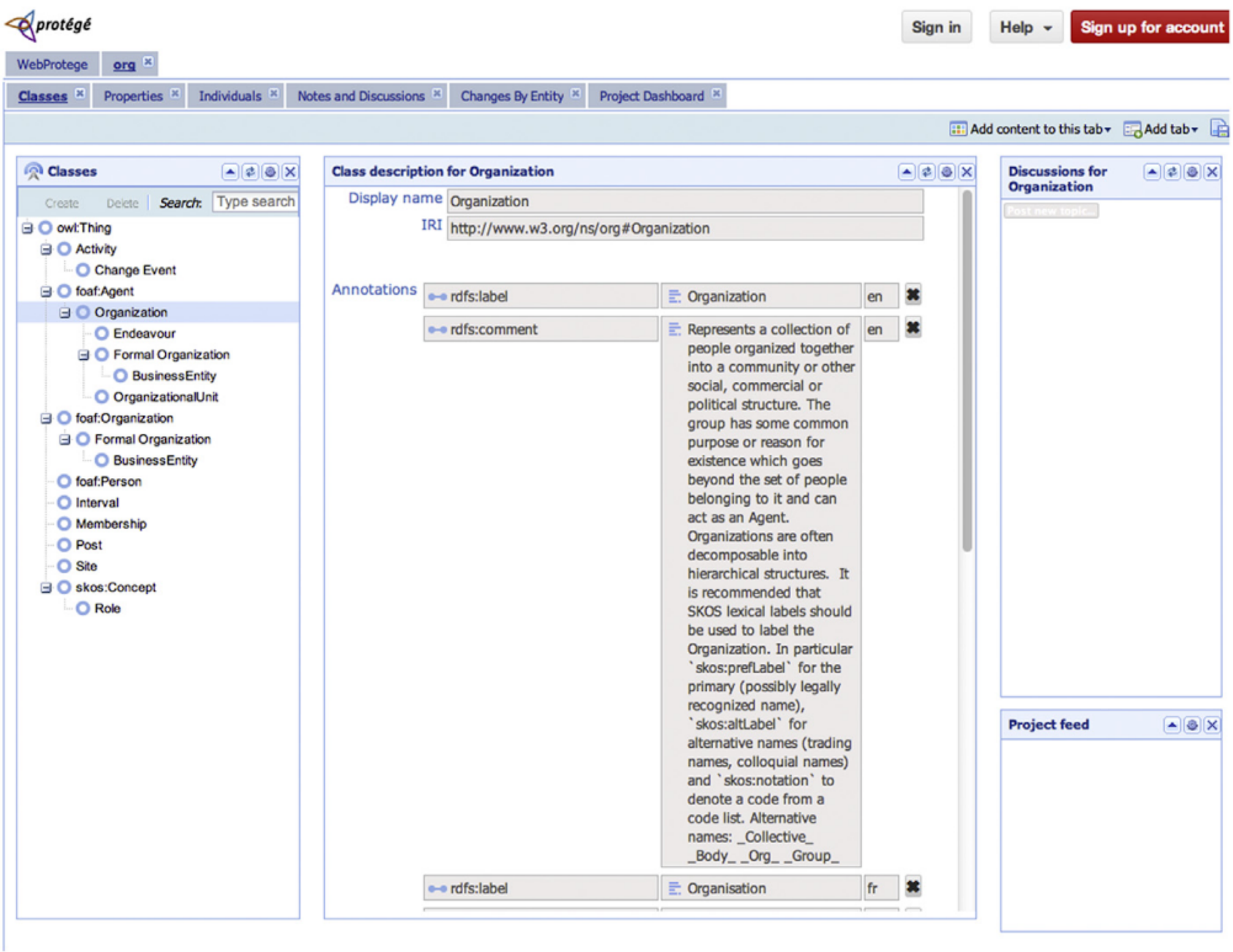

Figure 5.1 WebProtege ontology editor displaying an ontology's class hierarchy and some properties for a selected class.

Ontology development steps

1. Identify your rough taxonomy—what are the things and sub-things?
2. What things have non-relationships with one another?
3. What things inherit from multiple things?
4. What are the uniquely defining characteristics of an individual?
5. What are the measurable characteristics of a thing?
6. What things are described by referencing other things?

Some researchers in this field recommend that you build a separate object and knowledge ontology. They suggest that the object ontology might be reusable in other contexts if it is defined separately. In this approach, the object ontology defines the generic things that can exist, their placement in a hierarchy of classes, and their properties. The object ontology defines a language for the domain. The knowledge ontology in a sense extends the object ontology to express the relationships among its things in the context of a particular domain. This semantic modeling approach is embodied in a formal language called Gellish. Gellish is a representation-independent table of things, their identifiers, their labels in various languages, acronyms that apply to them, etc. If you define an object model for engines, the various possible parts of an engine could be defined in this model. It might include engine components such as a combustion chamber, cylinders, fuel injectors, a cooling system, and a super charger. A knowledge ontology could then describe the relationships among these components in a separate table for a knowledge domain about cars.

Building blocks of ontologies

RDFS (RDF Schema) is a language for describing basic types and relationships for Semantic Web vocabularies. It is essentially a taxonomy description language. A clue to this fact is the word schema, which is a term used for the definitions of XML vocabularies, which are hierarchical taxonomies of elements. To model the taxonomic aspects of an ontology, there are some primitives for specifying hierarchical relationship, including Class, subClassOf, Resource, and subPropertyOf. A class is a type of thing in an ontology—an animal, a vehicle, a name, etc. Mammal, marsupial, and kangaroo are all examples of classes in a domain defining animals. A subClass defines a hierarchical relationship—mammal is a subClass of animal for example. A property is a characteristic of a class, such as a measurement of some type, or a value from a control vocabulary of properties, or a numeric value or string literal such as 3.14159, "100 degrees C," or "blue." As you may recall, the Property class is defined in RDF, so RDFS simply extends this notion with the subPropertyOf property, to enable properties to inherit characteristics from one another in the same fashion that classes can do so.

```
<rdfs:Class rdf:resource="#Novel">
  <rdfs:subClassOf rdf:resource="#Book" />
</rdfs:Class>
```

If a class is defined as a subclass of another class, then it has the properties of that class. There's no need to redefine the properties of the parent class for the child. If a mammal has properties that express the fact that it has fur and its metabolism is "warm-blooded," and if a kangaroo is a subclass of a mammal, then by definition it has fur and is warm-blooded. You can add additional properties to a subclass which help to differentiate it to make it more specific. For a kangaroo, you might define a "means of locomotion" property, to which you assign the value "hopping." Properties can also inherit from one another. This can be used to narrow the range of potential values for the subproperty. Birds could be defined to have a color property that includes green and blue, but for mammals, you might elect to define a mammal color subproperty for which green and blue would be invalid values. These aspects of a property are defined with RDFS domain and range.

OWL, or the Web Ontology Language, was first published as a standard by the W3C in 2004. It is a language for modeling ontologies for the Semantic Web. In practice it is always used in conjunction with RDFS. For the remainder of this chapter, we will introduce the OWL language and use it in conjunction with RDFS to define some aspects of an ontology about libraries. Although this is a simple "toy" ontology, it should give you enough of a sense for what the Semantic Web ontology languages can do and the ability to read an ontology specification and understand most of what you would encounter. Once you complete this chapter, if you want to delve more deeply into ontology development, there are many good tutorials on the Web, including the well-known "Pizza Ontology Tutorial" (http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/) which is used as an exercise with an ontology editor called Protege. It can also be very helpful to download an ontology editor such as Protege and use it with an existing ontology that describes a subject area you are well acquainted with and use the software to explore the ontology (Figure 5.2).

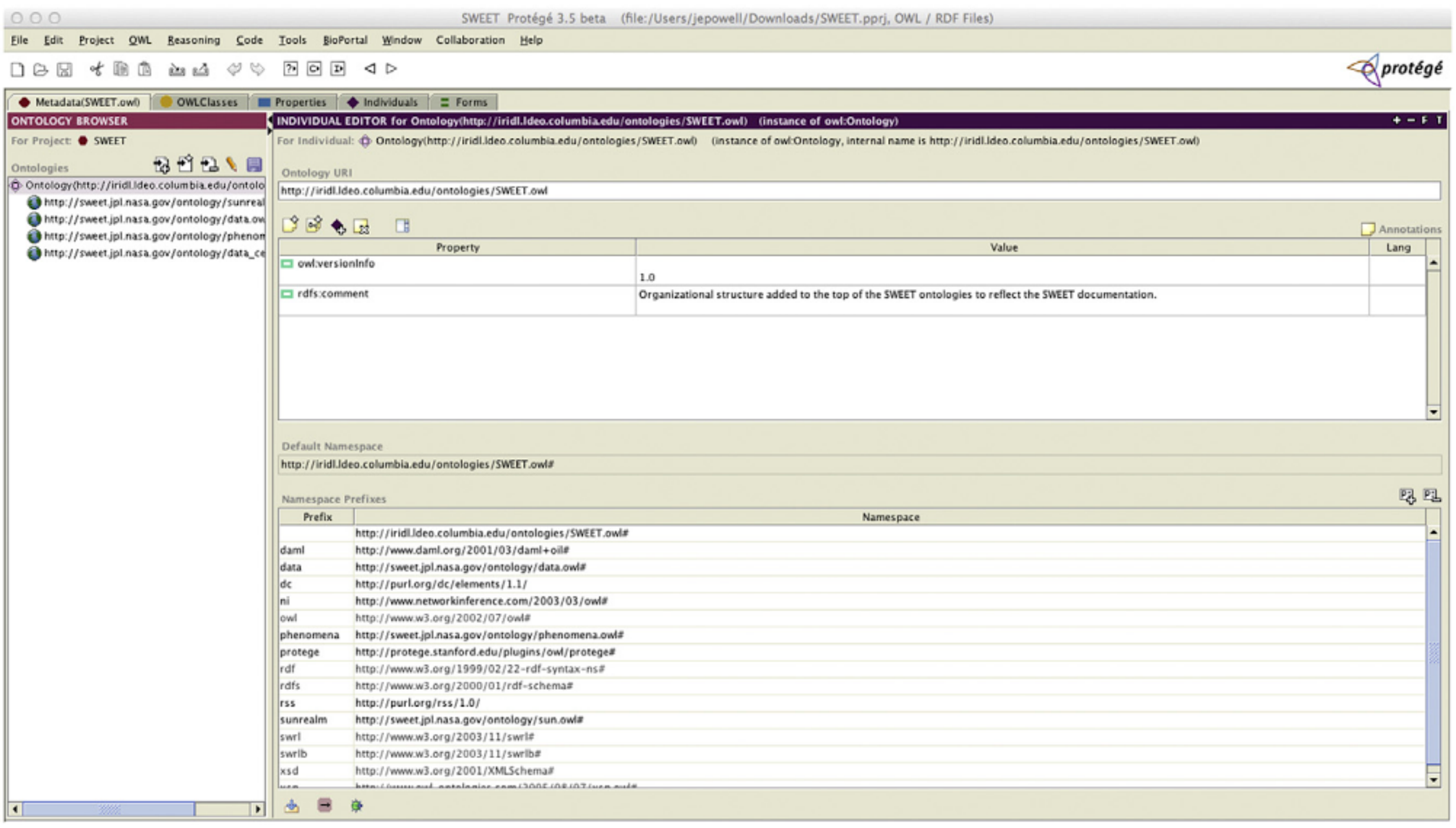

Figure 5.2 Protege displaying metadata about the SWEET ontology.

Ontology building tutorial

For our ontology, we will use RDFS and OWL to express various aspects and relationships among the following list of things:

- library
- librarian
- person
- book
- novel
- journal
- article
- chapter
- page
- director
- figure
- birthdate
- publication date
- genre
- patron
- call number
- publication

These terms represent the knowledge domain we want to model. The resulting ontology could be used to make statements about specific libraries, authors, books, etc. This is an interesting list from which to derive an ontology in part because there really isn't a strong taxonomic arrangement for the elements. Also, some of the items represent characteristics of other items in addition to, or instead of, things that exist. But in preparation for creating an ontology, let's go ahead and explore how these items might be reorganized into a taxonomy.

```
person
 + --librarian
 + --patron
library
publication
 + -- book
 + -- novel
 + -- journal
...
```

There aren't a lot of insights to be had in this taxonomy. For the most part, we still have a list of things that exist at the same top level, thus it is a shallow taxonomy. We do now know that a librarian and a patron are also a person, and that a journal and a book are a publication, and that a novel is a book. That's about it so far.

An ontology will tell us much more as you will soon see. Next we'll construct parts of an ontology to provide more information about these things and the relationships among them using RDFS and OWL. Since an ontology has to have a unique dereferenceable IRI, let's call ours: http://example.org/library#, which

we will refer to henceforth with the prefix lib. Here is how this is declared in RDF/XML:

```
<rdf:RDF
xmlns = "http://example.org/library#"
  xmlns:owl = "http://www.w3.org/2002/07/owl#"
  xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#"
  xmlns:dc = "http://purl.org/dc/elements/1.1/" >
  xmlns:xsd = "http://www.w3.org/2001/XMLSchema#" >
  <owl:Ontology rdf:about = "http://example.org/library"/ >
  <dc:title > Example Library Ontology </dc:title >
```

The first line defines our ontology namespace prefix for all our ontology elements that will follow. The remaining attributes for the rdf:RDF element reference other languages that our ontology model uses, including OWL, RDF, RDFS, and XSD. The last two lines declare the ontology namespace according to OWL 2 requirements and specify a title for the ontology.

An ontology defines classes for the things that exist in the domain you are modeling. Most of the things in our list and taxonomy above will be classes in our library ontology. OWL language elements for describing classes fall into six categories. The first kind of class description establishes a unique identifier for a thing. Here's an example that establishes librarian as a Thing:

```
<owl:Class rdf:ID = "Librarian" />
```

lib#Librarian can then serve as a predicate to indicate that something (well, someone) is a librarian. This statement is a shorthand way of asserting that a Librarian is an owl:Thing in your ontology. Owl supports two predefined classes, Thing and Nothing. Formally, every class in every ontology is a subclass of Thing. The other predefined class is Nothing, which is an empty set construct for instances where it is necessary to be explicit about the fact that you've logically reached the end of the line, so to speak. Or you may prefer to think of it as the logical equivalent of zero. The need for a Nothing class will become more apparent when we look at reasoning, in Chapter 7.

The next class description defines a class that is a collection of things. Since books and journals are both publications, we'll say there is a collection that can consist of books or journals, or both (Figure 5.3).

```
<owl:Class rdf:ID = "Publications" >
  <owl:someValuesFrom >
    <owl:Class >
      <owl:oneOf rdf:parseType = "Collection" >
        <owl:Thing rdf:about = "#Book" >
        <owl:Thing rdf:about = "#Journal" >
      </owl:oneOf >
    </owl:Class >
  </owl:someValuesFrom >
</owl:Class >
```

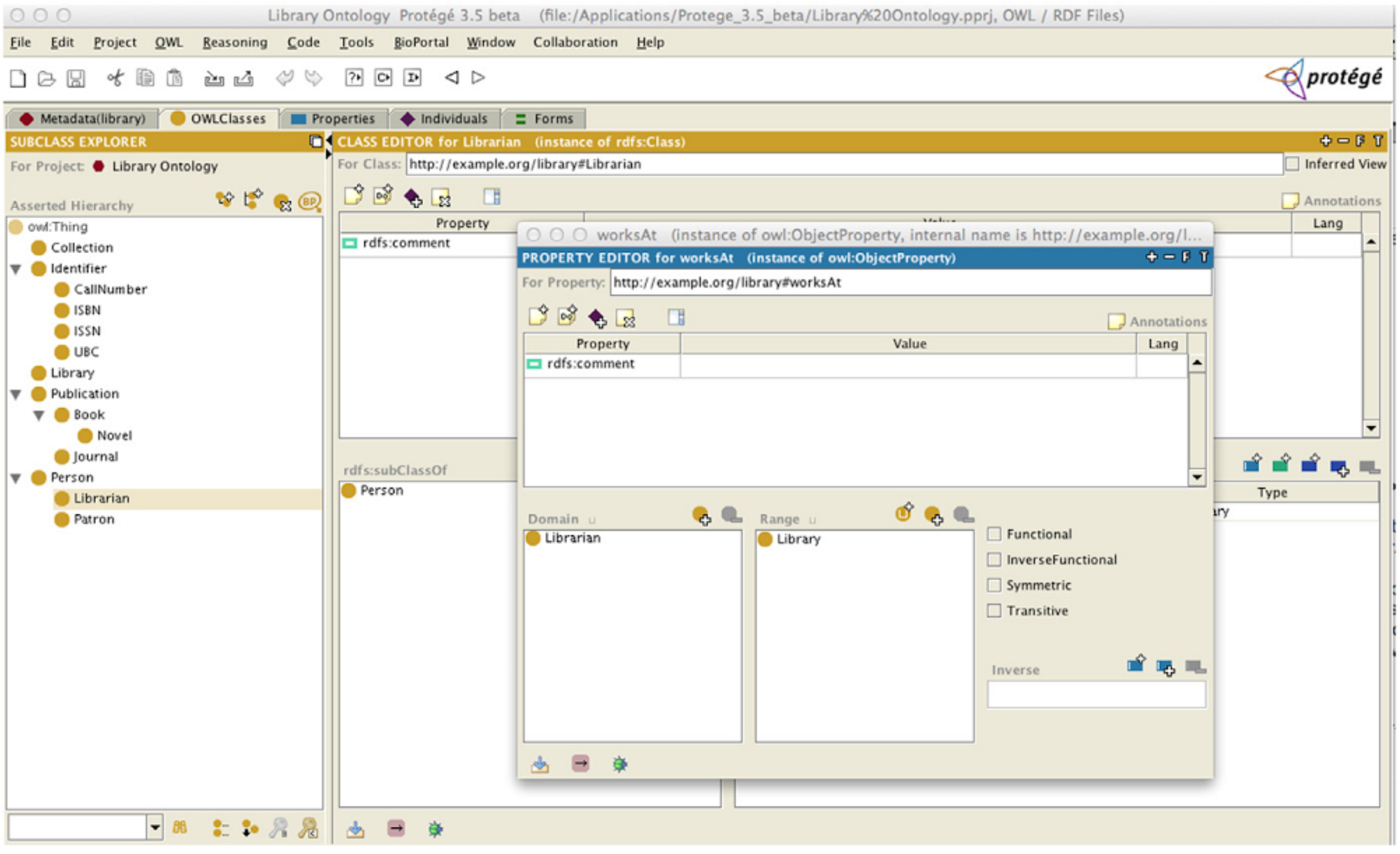

Figure 5.3 Composing a library ontology in Protege.

This states that a particular class Publication can be a Book or a Journal. This is called a class description. A class description is an anonymous class with no identifier. Class descriptions describe property restrictions, intersections, unions, and complement. Since we haven't yet discussed what a property is, we'll revisit the property restriction class description in a moment. Meanwhile, let's look at the "set" class descriptions' intersection, union, and complement. Since it's likely you've been exposed to the Boolean concepts of AND, OR, and NOT, we will use these to illustrate these class descriptions. Here's an example of a union (OR) set:

```
<owl:Class>
  <owl:unionOf rdf:parseType="Collection:>
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <owl:Thing rdf:about="#CallNumber"/>
        <owl:Thing rdf:about="#ISBN"/>
      </owl:oneOf>
    </owl:Class>
  <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <owl:Thing rdf:about="#ISSN"/>
        <owl:Thing rdf:about="#PMID"/>
      </owl:oneOf>
    </owl:Class>
  </owl:unionOf>
</owl:Class>
```

In this example, there are two collections. Whatever occurs in one or the other is added to the union, so the results are merged. The intersection (and) extension takes the same form as the example above except that it uses owl:intersectionOf instead of owl:unionOf. Only items that appear in both collections would be included in the intersection of the two collections. The complement extension looks a bit different:

```
<owl:Class>
  <owl:complimentOf>
    <owl:Class rdf:about="#UBC" />
  </owl:complimentOf>
</owlClass>
```

where this extension indicates that this class describes things that do not belong to a class called UBC (Figure 5.4).

Once you establish your classes and how they can be grouped, you will likely want to make some additional explicit statements regarding them. There is, as we've seen, some overlap in terms of class extensions and axioms. If you use an IRI in conjunction with class descriptions, then you quickly cross the line between extending a class and making statements about a class. But there are also three language elements for making axiomatic statements about classes in an ontology.

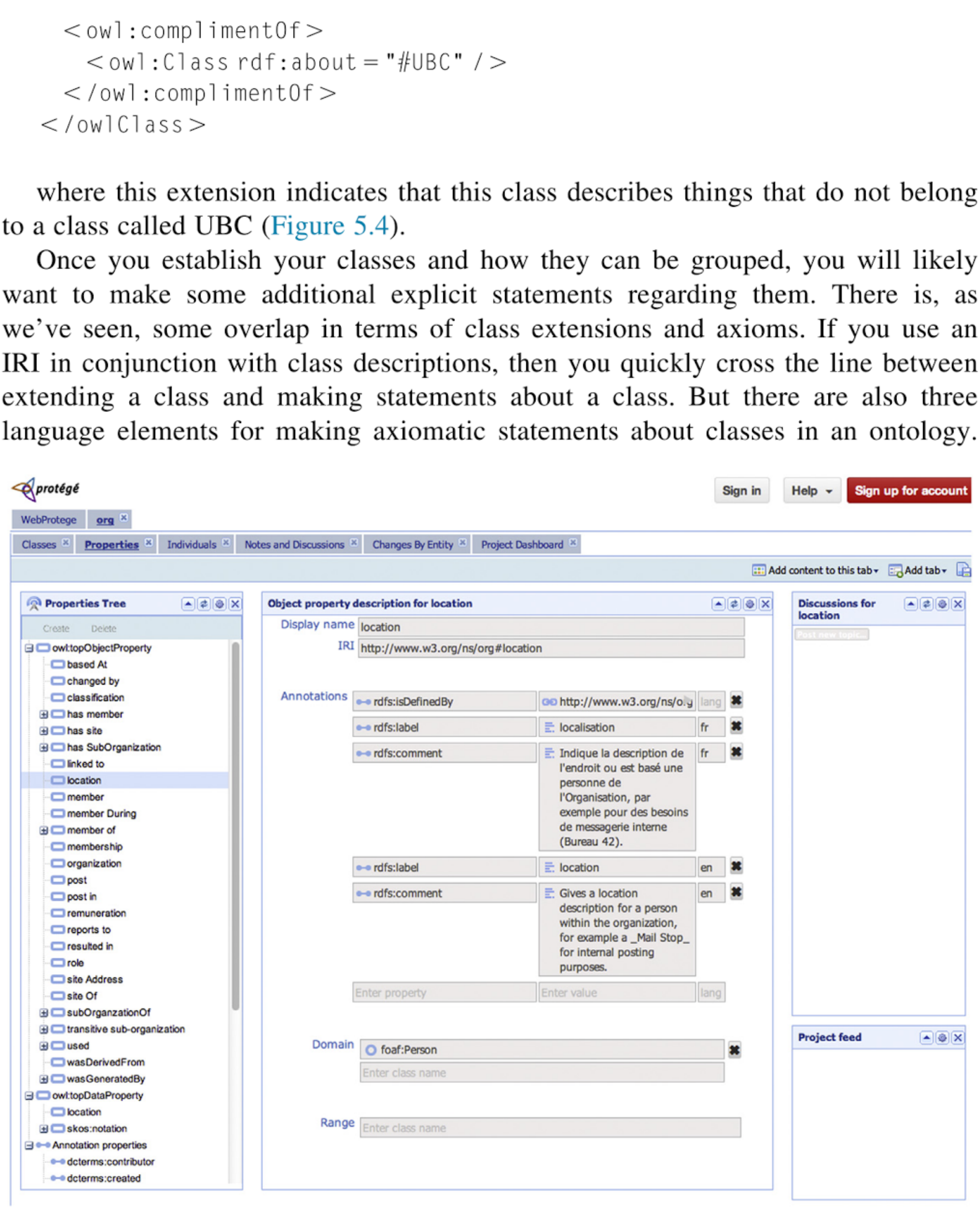

Figure 5.4 WebProtege property view.

They are rdfs:subClassOf which is used to define a hierarchical relationship between two classes:

```
<owl:Class rdf:ID="Novel">
  <rdfs:subClassOf rdf:resource="#Book" />
</owl:Class>
```

owl:equivalentClass which as its name implies says that two classes are equivalent or identical:

```
<owl:Class rdf:about="Journal">
  <owl:equivalentClass rdf:resource="#Magazine" />
</owl:Class>
```

and owl:disjointWith, which is used to indicate that one class is different than another (not equal). Here is an example:

```
<owl:Class rdf:about="CallNumber">
  <owl:disjointWith rdf:resource="ISSN" />
</owl:Class>
```

Most ontologies define both classes and properties. Properties represent characteristics of the things that have been described as classes in your ontology. There are two types of properties: object properties and data properties. You define object properties so that you can use classes to characterize other classes. You can define an object property by referring to the built-in class owl: ObjectProperty. Once you declare an object property, you can define its relation to other properties and how it can be used. Something you will often see associated with object properties in Semantic Web ontologies are rdfs:domain and rdfs:range. In practice, these establish the allowable class type(s) for the subject and object, respectively, for triples that use this property as a predicate (Figure 5.5).

```
<owl:ObjectProperty rdf:ID="hasPatron">
  <rdfs:domain rdf:resource="#Library" />
  <rdfs:range rdf:resource="#Patron" />
</owl:ObjectProperty>
```

The remaining RDFS and OWL language elements for object properties define relationships between them. For example, rdfs:subPropertyOf indicates that one property is a child of (and thus inherits the characteristics of) another property. OWL defines two elements for relating object properties to one another: equivalentProperty and inverseOf. Additionally, OWL defines a set of ObjectProperty subclasses that refine the notion of an object property in various ways: FunctionalProperty, InverseFunctionalProperty, TransitiveProperty, and

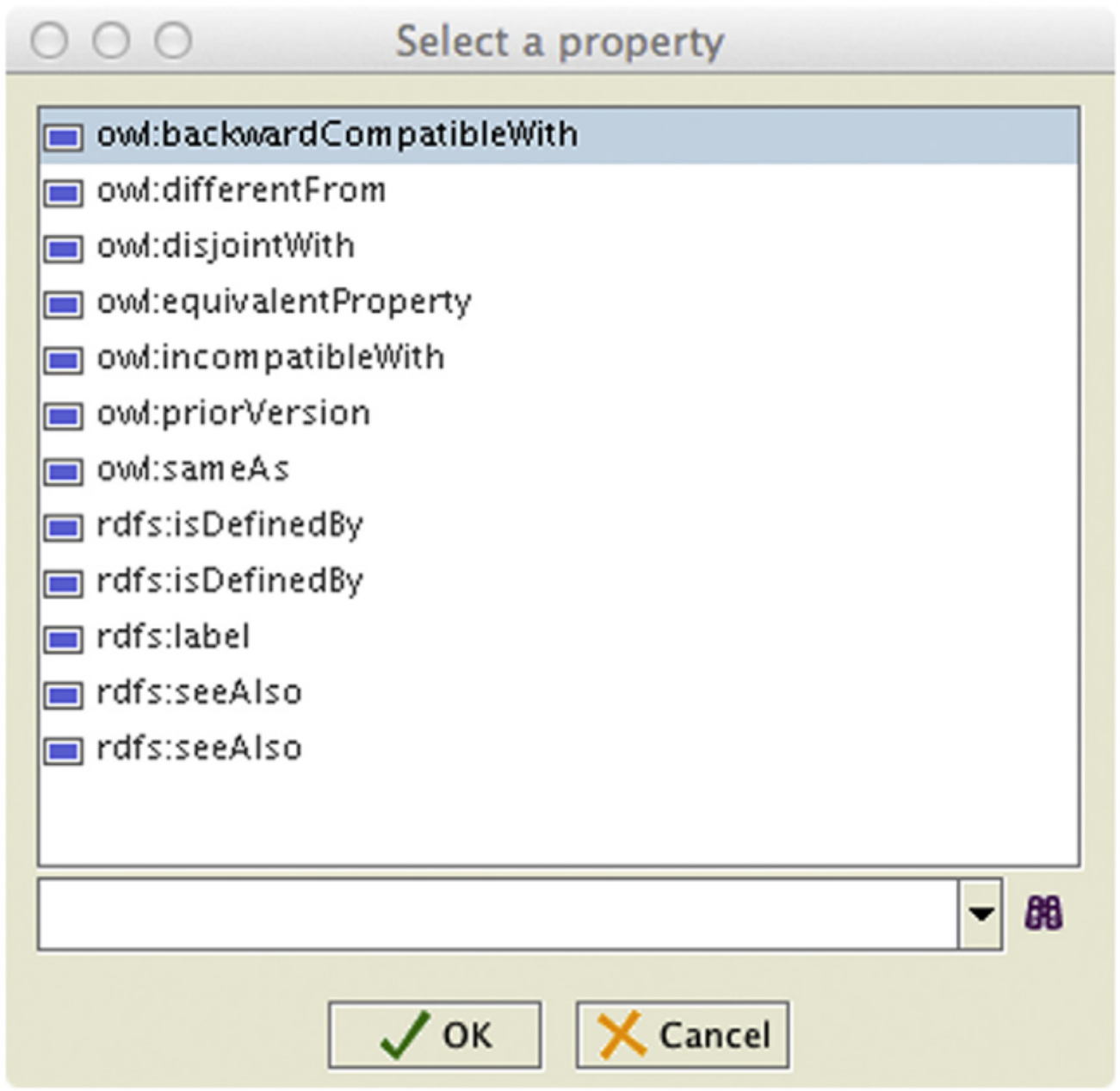

Figure 5.5 Protege data property view.

SymmetricProperty. Let's add a property that allows a patron to indicate that they "like" a certain publication:

```
<owl:ObjectProperty rdf:ID="like">
  <rdfs:domain rdf:resource="#Patron" />
  <rdfs:range rdf:resource="#Book" />
</owl:ObjectProperty>
```

Datatype properties primarily address individuals which are referred to as instances of some class. It is more typical in Semantic Web data sets to define individuals in instance data triples, but there are also various scenarios in which individuals are defined within the ontology itself. A data property is declared using owl:DatatypeProperty and is subsequently referred to as a modifier for some class or used to establish a characteristic of the instance of a class. Here is an example that declares a data property "birthDate" and subsequently uses it to establish that an individual Person named "Mary" has a birthdate of October 2, 1987:

```
<owl:DataProperty rdf:ID="birthDate" />
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="&xsd;dateTime"/>
</owl:DataProperty>
<Person rdf:about="Mary">
  <birthDate
</Person>
```

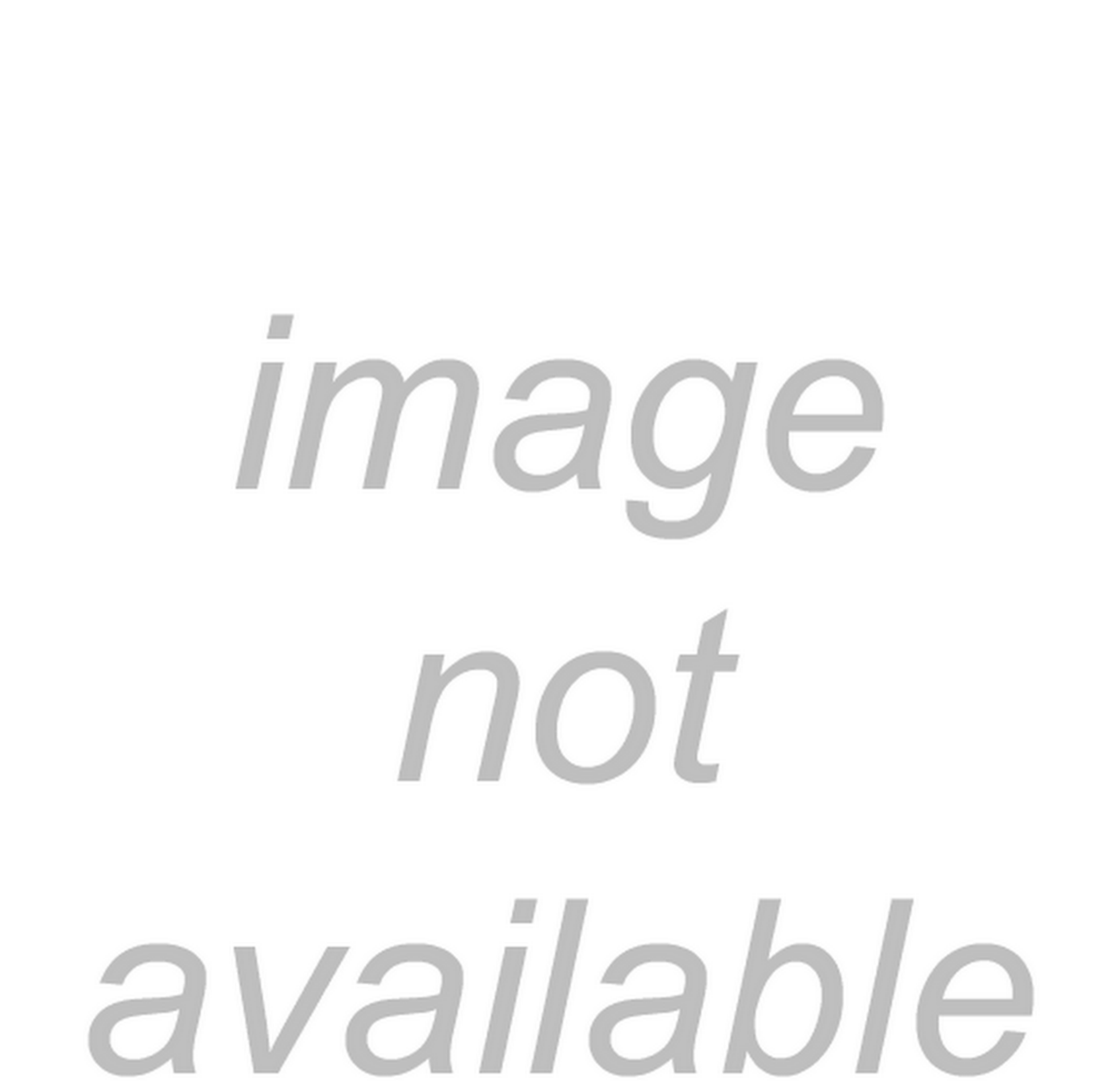
image
not
available

OWL 2 defines three profiles: OWL EL, OWL QL, and OWL RL. OWL EL is named for a descriptive logic model that is focused on defining what exists in a domain of knowledge (classes), rather than on the characteristic of a given class. It is easier (and thus faster) for software to process, so think of it as OWL Easy (not an official name!). OWL QL is geared toward supporting queries and reasoning, so you can think of it as OWL Query. The OWL RL profile is bounded so that it is well suited for rules languages, so you can think of OWL RL as OWL Rules. As you can tell, each profile is tuned to support a different type of machine processing. An important aspect of these profiles is to specify which components of OWL and RDFS (e.g., what class description types) can be used when defining an ontology. It sounds complicated, and it can be. In many Semantic Web applications, the predominant use case for the data might be searching or applying rules to determine actions based on the data, and for those uses, it is pretty clear which profile best fits the requirements.

We humans are adept at processing vast amounts of information about the world around us quickly and seemingly without much effort. With incredible speed, we apply a well-defined worldview, consider our assumptions, fill in the fuzzy parts, so we can in a split second conclude that this large clawed, fanged, growling, striped creature we've never seen before probably represents danger. We can start with some facts and draw new conclusions based on them, or figure out what set of circumstances would have to occur in order for something to be true.

In the Semantic Web world of deduction and logic, depending on how the ontologies you use are constructed, it may be essentially impossible for a computer to slog through all the data describing all the things this animal could be, and it could end up as lunch or a plaything for a tiger before (or if) it sorts it all out. If a reasoner can assume that anything that isn't explicitly stated is false, for example, then it bounds what is known to what is explicitly stated to be true. We generally don't think this way, but many judicial systems have a notion of innocent until proven guilty, and this lays a foundation for the machination of their court systems. This is known in logic as a closed world assumption. OWL uses an open world assumption, which means that whatever isn't explicitly stated is left as "undefined"—neither true nor false. This obviously implies further constraints on the ontology model.

Use cases for ontology models and resulting instance data can include searching, application of rules that examine the data to determine actions, and reasoning. Unambiguous modeling of facts and ensuring that data based on an ontology is machine processable were design considerations for OWL. As the global graph grows, and as various collections of triples make reference to the same thing in their instance data, and make use of the same ontologies, it becomes possible for software to connect the dots so to speak. But the vastness of the data set and the degree of expressiveness of the ontology, coupled with the assumptions that can be made in the absence of certainty, determines whether or not it is actually possible for this to occur. We will revisit modeling and reasoning in Chapter 7.

This page intentionally left blank

SPARQL

6

Triple patterns for search

Tim Berners Lee once said of the Semantic Web "Trying to use the Semantic Web without SPARQL is like trying to use a relational database without Structured Query Language (SQL)." Search is still the killer app for data. Every major advance in information technology has been driven by search. Card catalogs moved online so that users could sit in front of a terminal, and later a computer, and search by authors, titles, subjects, and even by keywords that could occur in any field. As soon as storage and computational speed allowed for it, solutions were devised for searching collections of full text documents.

Gerard Salton revolutionized document searching with the vector space model, which transformed documents into mathematical constructs called vectors. A term frequency vector maintains a slot corresponding to each word in a document. In this slot resides a number that indicates how many times the word occurred. With Salton's term frequency vectors, searching text documents became a problem that could be solved with basic geometry. Google revolutionized the World Wide Web when they debuted their fast, comprehensive, simple search for Web pages. Google leveraged the explicit graph within the Web by taking into account links among Web pages. Many big data algorithms in machine learning rely on search. A well-known clustering algorithm, k-means clustering, can operate on Salton's vectors to identify groups of related documents automatically. You can use links to meander around the Semantic Web without search, but finding just the right piece of information is like looking for a needle in the proverbial haystack. The Semantic Web needed its own search, so they baked one into the core suite of standards. That search is based on a search language called SPARQL. It seems that search, like quality, will never go out of style.

Search requires some kind of optimized representation of a collection of data. We often use the term database to refer to a system that stores data and makes it searchable. Semantic Web databases are sometimes referred to as an RDF graph database, a semantic repository, or simply a triplestore. A database for Semantic Web data is different because the underlying data is different. For example, there's no predefined schema for RDF. Content producers are free to mix and match various ontologies as they see fit. An RDF description of a person might start off with triples for their name, address, telephone number, and e-mail address. Sometime later, you might need to add some more information such as a twitter ID, a Facebook URL, a second phone number, or a birth date. With a relational database, you'd have to change the structure of the database, perhaps create new tables, new columns, etc. There are no such restrictions on a triplestore. If you need to use a new predicate, reference a new ontology, or load data from another source, you just do it.

A Librarian's Guide to Graphs, Data and the Semantic Web.

Since RDF is different, search is also different. RDF is a graph. We're looking for nodes and edges when we search RDF. If you are searching by edge, then you would want to know the predicate name and the ontology from which it comes. If you are searching by object values, then you'll want to know if the object is a literal value like a number or a person's name, or an URI for a class or instance of some class. So an RDF query is formulated at least in part as node−edge−node patterns and these patterns take the form of a triple. Some parts of the triple are explicitly identified in this pattern, and others are specified with placeholder variables, indicating that any value is acceptable.

Ontologies play a role in how the triplestore responds to certain queries. If the triplestore supports inferencing, then it can look at the data as if it had parents, so to speak. A resource has a type. The type is a class. That class might have a parent class. So the resource is like its parent, its grandparent, etc. This is an effect of the subClassOf or subPropertyOf statements in the source ontology. When you search, these relationships might affect the result you get back. There's no equivalent capability in a relational database!

The Semantic Web allows things described in triples to be linked to other things all over the Web. To facilitate this, search capabilities of a triplestore are often open to the public by default. This open-by-default model is very different than the approach taken by relational databases, which are, by default and design, closed systems not typically searchable by the world.

There are many open source and commercial triplestores. Some are better at scaling up to large amounts of data, some have better transaction support, visualization tools, software APIs, etc. The above description provides a general overview of their capabilities. In a later chapter, we'll take a closer look at triplestores. For the rest of this chapter, we'll focus on SPARQL, the query language for the Semantic Web (Figure 6.1).

SPARQL

SPARQL is a self-referential, recursive acronym for the Semantic Web's most widely used and supported query language. That's a long winded way of saying that the name SPARQL is a bit of a (nerdy) joke. It stands for SPARQL protocol and RDF query language. As the name implies, it defines both a search language and a way of communicating with a triplestore. The SPARQL protocol is built on top of the Web request protocol (HTTP). This means a SPARQL query can be transported over the World Wide Web. The SPARQL protocol requires that search results be returned as an XML document. We will look at details about the protocol later. But for starters, we'll focus on the query language itself. It is important to understand how to construct a SPARQL query since the query structure is the same whether you use the network protocol, a software library, or a triplestore's built-in interactive interface for performing SPARQL queries.

A SPARQL query is a search string that specifies what to search, an RDF pattern that results need to match, and details about constructing the results set. There are

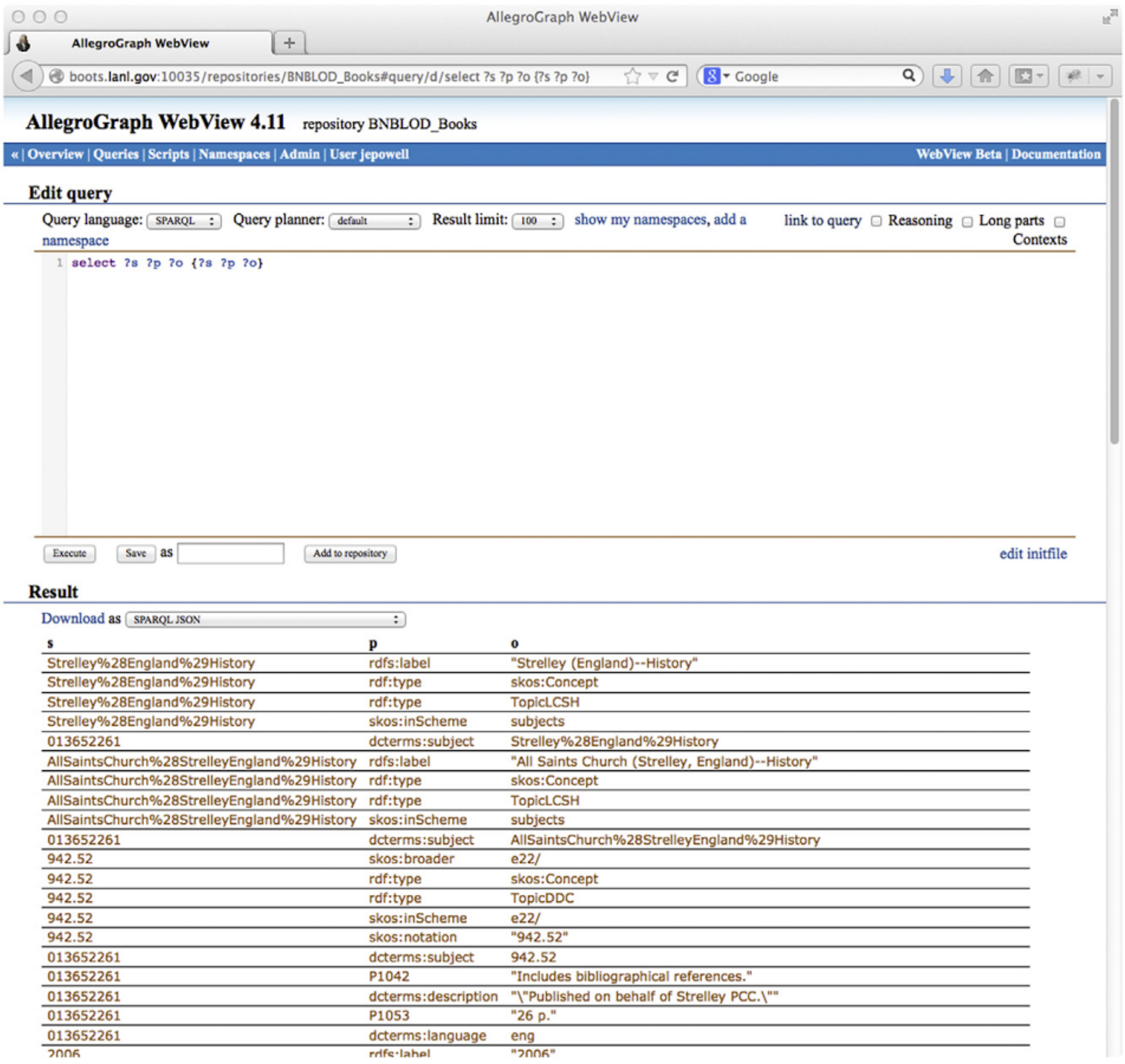

Figure 6.1 A simple SPARQL query in AllegroGraph.

keywords that indicate the aspect of the query that's being defined. The PREFIX keyword defines one or more namespace abbreviations that are subsequently used in the query. The SELECT keyword is one of four possible return clause keywords, indicating what data should be returned in the results set. The WHERE prefix defines the where clause that specifies one or more the graph patterns to be matched. These patterns are specified in RDF. Finally the result modifier clause uses various keywords to indicate how the collection of results is to be presented. Here is an example (Figure 6.2):

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?id ?name
WHERE { ?id foaf:name ?name. }
ORDER BY ?name.
```

In many ways, this ought to look familiar. The syntax is concise like Turtle and uses many of the same conventions. A prefix means the same thing here as in Turtle—substitute the IRI inside brackets for the prefix foaf. The content

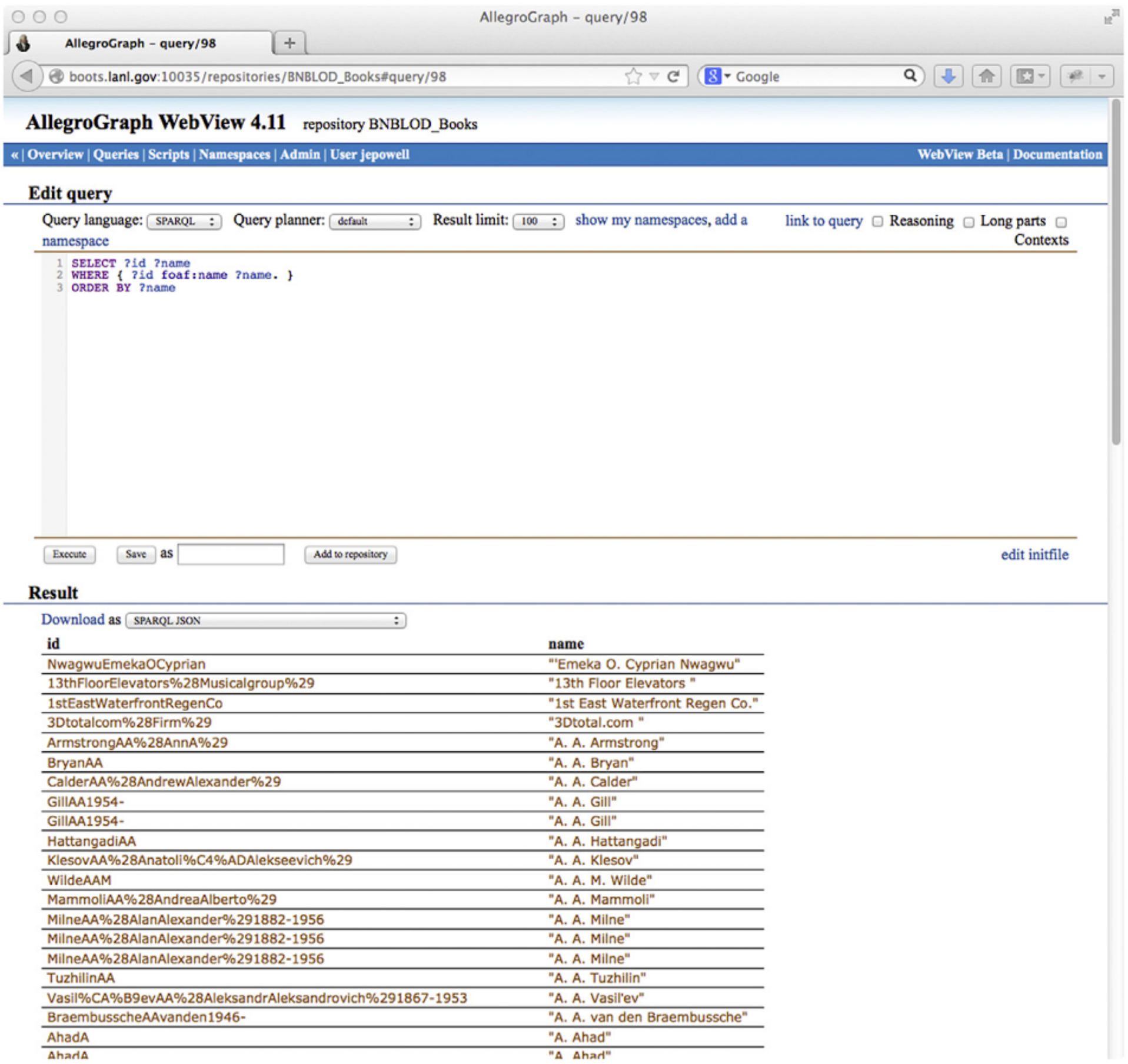

Figure 6.2 SPARQL query that limits results to triples that have foaf:name as a predicate.

between the curly brackets after WHERE constitutes an RDF graph pattern. A graph pattern can be as simple as a pair of nodes and an edge, in other words, a single triple, or include multiple triples, blank nodes, essentially a subgraph that results ought to match. Here is perhaps the simplest SPARQL query you will ever encounter:

```
select *
where { ?s ?p ?o. }
```

You are probably wondering about all these names and letters preceded by question marks. These are variables. They will assume any value from any triple in the repository that matches this pattern. In the first example, the only requirement this pattern specifies for a match is that the triple has foaf:name as the predicate. The subject and object can have any values. For all matching triples, the subject of each triple will be assigned to the ?id variable and the corresponding object to the ?name variable. Once all the results are retrieved, these pairs of values will be sorted by the contents of the name variable and then output as a table. In the second example, any triple matches, regardless of subject, predicate, or object value.

A very useful SPARQL query that can be performed on data sets you may not know much about is this:

```
select distinct ?pred
where { ?x ?pred ?y. }
```

This query will return the names of all of the distinct predicate values used in a given repository. Using this you can identify the ontologies and properties used in the instance data triples.

Here's an example that binds title and name values by using a multi-statement where clause (Figure 6.3):

```
select ?id ?title ?name
where {

  ?s dcterms:title ?title.
  ?s dcterms:creator ?name.

}
```

If you know anything at all about the relational database search language SQL, the role of SPARQL in the context of triplestores is identical. In some ways, SPARQL

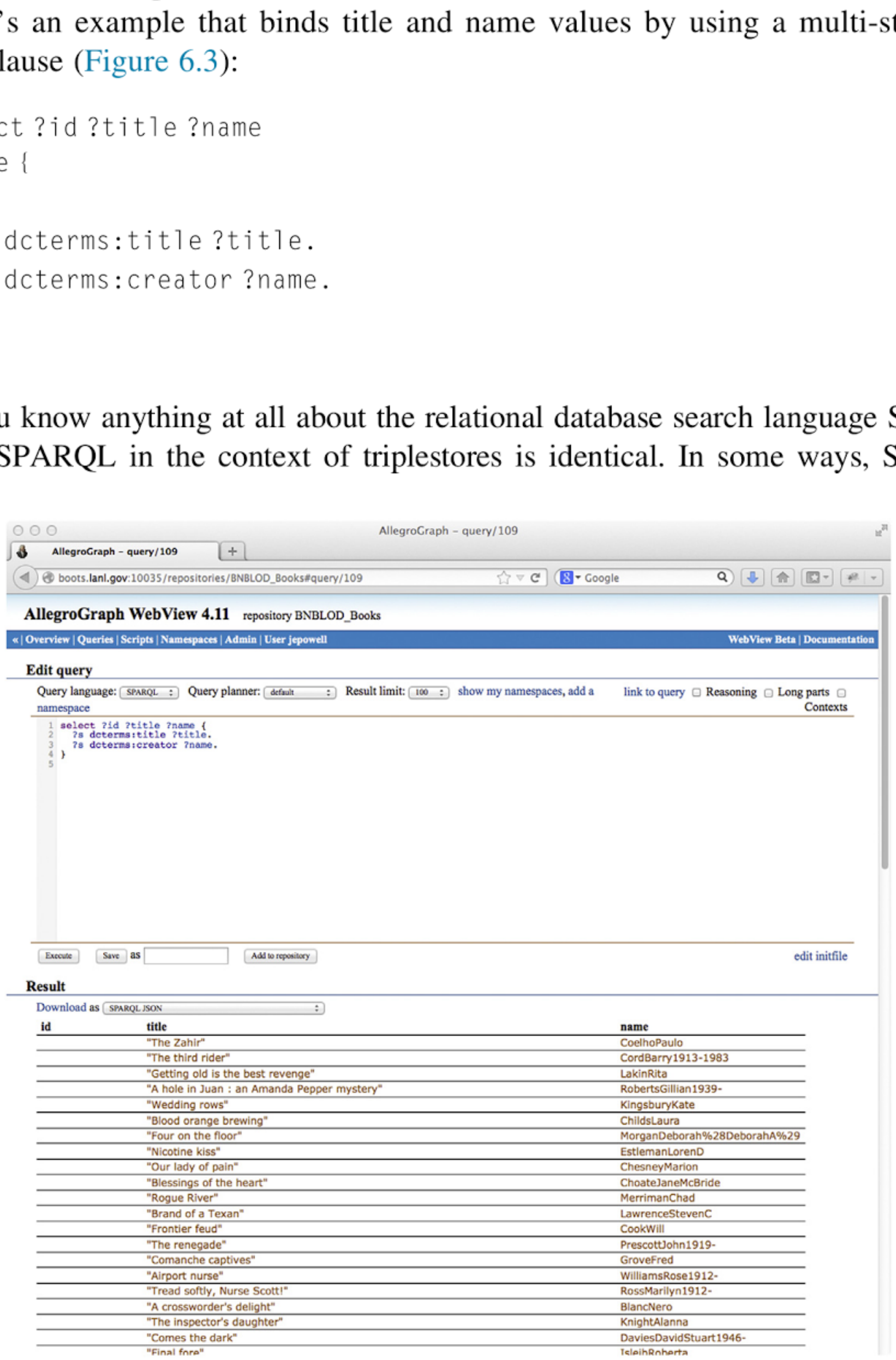

Figure 6.3 SPARQL query that retrieves dcterms title and creator property values for every matching triple.

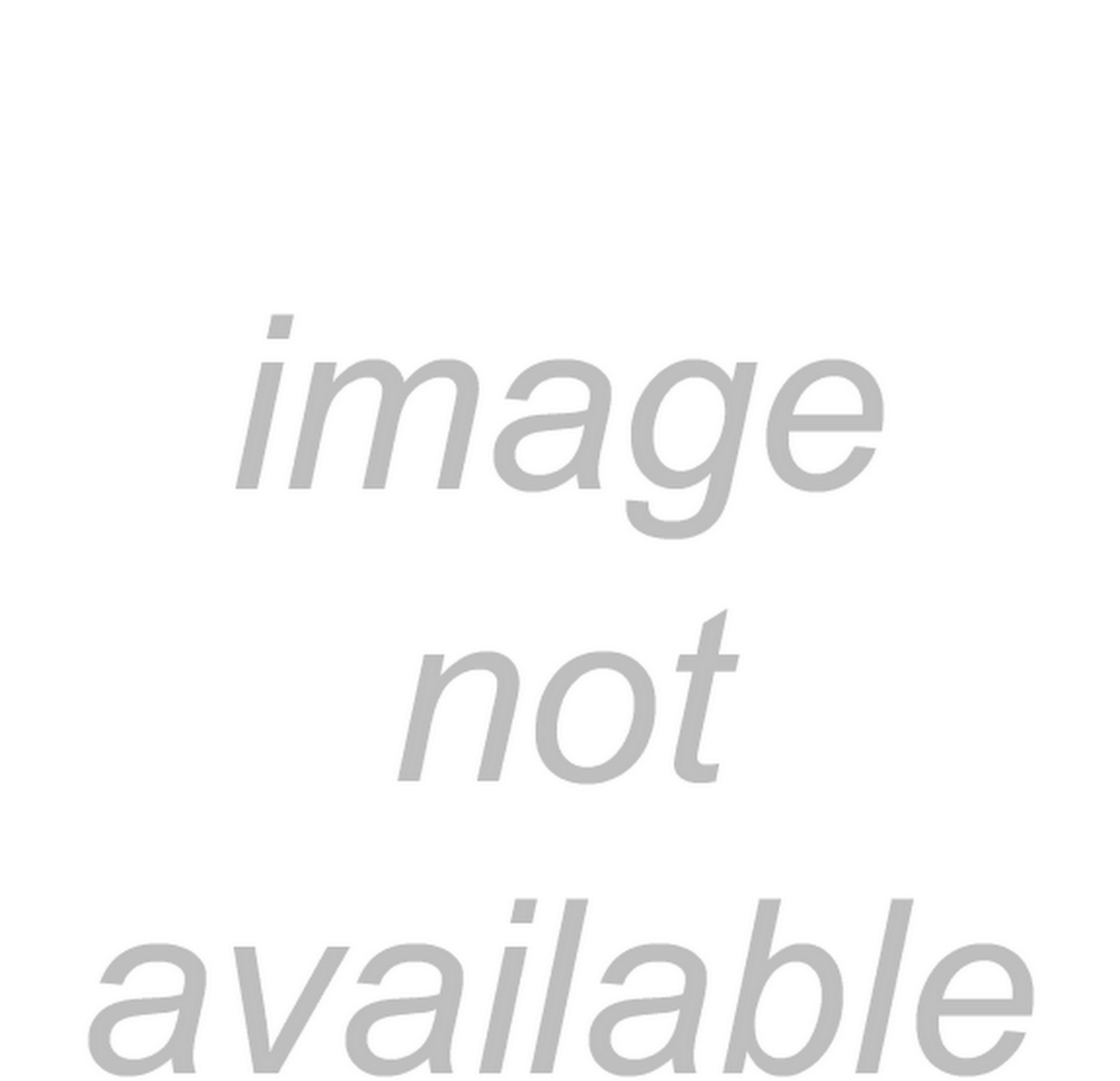
image
not
available