SECOND EDITION

Semantic Web for the Working Ontologist

Effective Modeling in RDFS and OWL

Dean Allemang
James Hendler

MK
MORGAN KAUFMANN

Acquiring Editor: Todd Green
Development Editor: Robyn Day
Project Manager: Sarah Binns
Designer: Kristen Davis

Morgan Kaufmann Publishers is an imprint of Elsevier.
225 Wyman Street, Waltham, MA 02451, USA

This book is printed on acid-free paper.

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.

Library of Congress Cataloging-in-Publication Data

Allemang, Dean.
Semantic Web for the working ontologist : effective modeling in RDFS and OWL / Dean Allemang, Jim Hendler. – 2nd ed.
p. cm.
Includes index.
ISBN 978-0-12-385965-5
1. Web site development. 2. Semantic Web. 3. Metadata. I. Hendler, James A. II. Title.
TK5105.888.A45 2012
025.042′7–dc22

2011010645

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

For information on all Morgan Kaufmann publications, visit our Web site at www.mkp.com or www.elsevierdirect.com

Printed in the United States of America

11 12 13 14 15 5 4 3 2 1

Contents

This page intentionally left blank

Preface to the second edition

Since the first edition of *Semantic Web for the Working Ontologist* came out in June 2008, we have been encouraged by the reception the book has received. Practitioners from a wide variety of industries—health care, energy, environmental science, life sciences, national intelligence, and publishing, to name a few—have told us that the first edition clarified for them the possibilities and capabilities of Semantic Web technology. This was the audience we had hoped to reach, and we are happy to see that we have.

Since that time, the technology standards of the Semantic Web have continued to develop. SPARQL, the query language for RDF, became a Recommendation from the World Wide Web Consortium and was so successful that version 2 is already nearly ready (it will probably be ratified by the time this book sees print). SKOS, which we described as an example of modeling "in the wild" in the first edition, has raced to the forefront of the Semantic Web with high-profile uses in a wide variety of industries, so we gave it a chapter of its own. Version 2 of the Web Ontology Language, OWL, also appeared during this time.

Probably the biggest development in the Semantic Web standards since the first edition is the rise of the query language SPARQL. Beyond being a query language, SPARQL is a powerful graph-matching language which pushes its utility beyond simple queries. In particular, SPARQL can be used to specify general inferencing in a concise and precise way. We have adopted it as the main expository language for describing inferencing in this book. It turns out to be a lot easier to describe RDF, RDFS, and OWL in terms of SPARQL.

The "in the wild" sections became problematic in the second edition, but for a good reason—we had too many good examples to choose from. We're very happy with the final choices, and are pleased with the resulting "in the wild" chapters (9 and 13). The Open Graph Protocol and Good Relations are probably responsible for more serious RDF data on the Web than any other efforts. While one may argue (and many have) that FOAF is getting a bit long in the tooth, recent developments in social networking have brought concerns about privacy and ownership of social data to the fore; it was exactly these concerns that motivated FOAF over a decade ago. We also include two scientific examples of models "in the wild"—QUDT (Quantities, Units, Dimensions, and Types) and The Open Biological and Biomedical Ontologies (OBO). QUDT is a great example of how SPARQL can be used to specify detailed computation over a large set of rules (rules for converting units and for performing dimensional analysis). The wealth of information in the OBO has made them perennial favorites in health care and the life sciences. In our presentation, we hope to make them accessible to an audience who doesn't have specialized experience with OBO publication conventions. While these chapters logically build on the material that precedes them, we have done our best to make them stand alone, so that impatient readers who haven't yet mastered all the fine points of the earlier chapters can still appreciate the "wild" examples.

We have added some organizational aids to the book since the first edition. The "Challenges" that appear throughout the book, as in the first edition, provide examples for how to use the Semantic Web technologies to solve common modeling problems. The "FAQ" section organizes the challenges by topic, or, more properly, by the task that they illustrate. We have added a numeric index of all the challenges to help the reader cross-reference them.

We hope that the second edition will strike a chord with our readers as the first edition has done.

On a sad note, many of the examples in Chapter 5 use "Elizabeth Taylor" as an example of a "living actress." During postproduction of this book, Dame Elizabeth Taylor succumbed to congestive heart failure and died. We were too far along in the production to make the change, so we have kept the examples as they are. May her soul rest in peace.

PREFACE TO THE FIRST EDITION

In 2003, when the World Wide Web Consortium was working toward the ratification of the Recommendations for the Semantic Web languages, RDF, RDFS, and OWL, we realized that there was a need for an industrial-level introductory course in these technologies. The standards were technically sound, but, as is typically the case with standards documents, they were written with technical completeness in mind rather than education. We realized that for this technology to take off, people other than mathematicians and logicians would have to learn the basics of semantic modeling.

Toward that end, we started a collaboration to create a series of trainings aimed not at university students or technologists but at Web developers who were practitioners in some other field. In short, we needed to get the Semantic Web out of the hands of the logicians and Web technologists, whose job had been to build a consistent and robust infrastructure, and into the hands of the practitioners who were to build the Semantic Web. The Web didn't grow to the size it is today through the efforts of only HTML designers, nor would the Semantic Web grow as a result of only logicians' efforts.

After a year or so of offering training to a variety of audiences, we delivered a training course at the National Agriculture Library of the U.S. Department of Agriculture. Present for this training were a wide variety of practitioners in many fields, including health care, finance, engineering, national intelligence, and enterprise architecture. The unique synergy of these varied practitioners resulted in a dynamic four-day investigation into the power and subtlety of semantic modeling. Although the practitioners in the room were innovative and intelligent, we found that even for these early adopters, some of the new ways of thinking required for modeling in a World Wide Web context were too subtle to master after just a one-week course. One participant had registered for the course multiple times, insisting that something else "clicked" each time she went through the exercises.

This is when we realized that although the course was doing a good job of disseminating the information and skills for the Semantic Web, another, more archival resource was needed. We had to create something that students could work with on their own and could consult when they had questions. This was the point at which the idea of a book on modeling in the Semantic Web was conceived. We realized that the readership needed to include a wide variety of people from a number of fields, not just programmers or Web application developers but all the people from different fields who were struggling to understand how to use the new Web languages.

It was tempting at first to design this book to be the definitive statement on the Semantic Web vision, or "everything you ever wanted to know about OWL," including comparisons to program modeling languages such as UML, knowledge modeling languages, theories of inferencing and logic, details of the Web infrastructure (URIs and URLs), and the exact current status of all the developing standards (including SPARQL, GRDDL, RDFa, and the new OWL 1.1 effort). We realized, however, that not only would such a book be a superhuman undertaking, but it would also fail to serve our primary purpose of putting the tools of the Semantic Web into the hands of a generation of intelligent practitioners who could build real applications. For this reason, we concentrated on a particular essential skill for constructing the Semantic Web: building useful and reusable models in the World Wide Web setting.

Many of these patterns entail several variants, each embodying a different philosophy or approach to modeling. For advanced cases such as these, we realized that we couldn't hope to provide a single, definitive answer to how these things should be modeled. So instead, our goal is to educate domain

practitioners so that they can read and understand design patterns of this sort and have the intellectual tools to make considered decisions about which ones to use and how to adapt them. We wanted to focus on those trying to use RDF, RDFS, and OWL to accomplish specific tasks and model their own data and domains, rather than write a generic book on ontology development. Thus, we have focused on the "working ontologist" who was trying to create a domain model on the Semantic Web.

The design patterns we use in this book tend to be much simpler. Often a pattern consists of only a single statement but one that is especially helpful when used in a particular context. The value of the pattern isn't so much in the complexity of its realization but in the awareness of the sort of situation in which it can be used.

This "make it useful" philosophy also motivated the choice of the examples we use to illustrate these patterns in this book. There are a number of competing criteria for good example domains in a book of this sort. The examples must be understandable to a wide variety of audiences, fairly compelling, yet complex enough to reflect real modeling situations. The actual examples we have encountered in our customer modeling situations satisfy the last condition but either are too specialized—for example, modeling complex molecular biological data; or, in some cases, they are too business-sensitive—for example, modeling particular investment policies—to publish for a general audience.

We also had to struggle with a tension between the coherence of the examples. We had to decide between using the same example throughout the book versus having stylistic variation and different examples, both so the prose didn't get too heavy with one topic, but also so the book didn't become one about how to model—for example, the life and works of William Shakespeare for the Semantic Web.

We addressed these competing constraints by introducing a fairly small number of example domains: William Shakespeare is used to illustrate some of the most basic capabilities of the Semantic Web. The tabular information about products and the manufacturing locations was inspired by the sample data provided with a popular database management package. Other examples come from domains we've worked with in the past or where there had been particular interest among our students. We hope the examples based on the roles of people in a workplace will be familiar to just about anyone who has worked in an office with more than one person, and that they highlight the capabilities of Semantic Web modeling when it comes to the different ways entities can be related to one another.

Some of the more involved examples are based on actual modeling challenges from fairly involved customer applications. For example, the ice cream example in Chapter 7 is based, believe it or not, on a workflow analysis example from a NASA application. The questionnaire is based on a number of customer examples for controlled data gathering, including sensitive intelligence gathering for a military application. In these cases, the domain has been changed to make the examples more entertaining and accessible to a general audience.

We have included a number of extended examples of Semantic Web modeling "in the wild," where we have found publicly available and accessible modeling projects for which there is no need to sanitize the models. These examples can include any number of anomalies or idiosyncrasies, which would be confusing as an introduction to modeling but as illustrations give a better picture about how these systems are being used on the World Wide Web. In accordance with the tenet that this book does not include everything we know about the Semantic Web, these examples are limited to the modeling issues that arise around the problem of distributing structured knowledge over the Web. Thus, the treatment focuses on how information is modeled for reuse and robustness in a distributed environment.

By combining these different example sources, we hope we have struck a happy balance among all the competing constraints and managed to include a fairly entertaining but comprehensive set of examples that can guide the reader through the various capabilities of the Semantic Web modeling languages.

This book provides many technical terms that we introduce in a somewhat informal way. Although there have been many volumes written that debate the formal meaning of words like *inference, representation,* and even *meaning,* we have chosen to stick to a relatively informal and operational use of the terms. We feel this is more appropriate to the needs of the ontology designer or application developer for whom this book was written. We apologize to those philosophers and formalists who may be offended by our casual use of such important concepts.

We often find that when people hear we are writing a new Semantic Web modeling book, their first question is, "Will it have examples?" For this book, the answer is an emphatic "Yes!" Even with a wide variety of examples, however, it is easy to keep thinking "inside the box" and to focus too heavily on the details of the examples themselves. We hope you will use the examples as they were intended: for illustration and education. But you should also consider how the examples could be changed, adapted, or retargeted to model something in your personal domain. In the Semantic Web, Anyone can say Anything about Any topic. Explore the freedom.

Second Printing: Since the first printing there have been advances in several of the technologies we discuss such as SPARQL, OWL 2, and SKOS that go beyond the state of affairs at the time of first printing. We have created a web site that covers developing technology standards and changing thinking about the best practices for the Semantic Web. You can find it at http://www.workingontologist.org/.

Acknowledgments

The second edition builds on the work of Semantic Web practitioners and researchers who have moved the field forward in the past two years—they are too numerous to thank individually. But we would like to extend special recognition to James "Chip" Masters, Martin Hepp, Ralph Hodgson, Austin Haugen, and Paul Tarjan, whose work on various ontologies allowed them to be mature enough to serve as examples "in the wild."

We also want to thank TopQuadrant, Inc. for making their software TopBraid Composer™ available for the preparation of the book. All examples were managed using this software, and the figures that show RDF data were laid out using its graphic capabilities. The book would have been much harder to manage without it.

Once again, Mike Uschold contributed heroic effort as a reviewer of several of the chapters. We also wish to thank John Madden, Scott Henninger, and Jeff Stein for their reviews of various parts of the second edition.

The faculty staff and students at the Tetherless World Constellation at RPI have also been a great help. The inside knowledge from members of the various W3C working groups they staff, the years of experience in Semantic Web among the staff, and the great work done by Peter Fox and Deborah McGuinness served as inspiration as well as encouragement in getting the second edition done.

We especially want to thank Todd Green and the staff at Elsevier for pushing us to do a second edition, and for their patience when we missed deadlines that meant more work for them in less time.

Most of all, we want to thank the readers who provided feedback on the first edition that helped us to shape the book as it is now. We write books for the readers, and their feedback is essential. Thank you for the work you put in on the web site—you have been heard, and your feedback is incorporated into the second edition.

This page intentionally left blank

About the authors

Dean Allemang is the chief scientist at TopQuadrant, Inc.—the first company in the United States devoted to consulting, training, and products for the Semantic Web. He codeveloped (with Professor Hendler) TopQuadrant's successful Semantic Web training series, which he has been delivering on a regular basis since 2003.

He was the recipient of a National Science Foundation Graduate Fellowship and the President's 300th Commencement Award at Ohio State University. He has studied and worked extensively throughout Europe as a Marshall Scholar at Trinity College, Cambridge, from 1982 through 1984 and was the winner of the Swiss Technology Prize twice (1992 and 1996).

He has served as an invited expert on numerous international review boards, including a review of the Digital Enterprise Research Institute—the world's largest Semantic Web research institute, and the Innovative Medicines Initiative, a collaboration between 10 pharmaceutical companies and the European Commission to set the roadmap for the pharmaceutical industry for the near future.

Jim Hendler is the Tetherless World Senior Constellation Chair at Rensselaer Polytechnic Institute where he has appointments in the Departments of Computer Science and Cognitive Science and the Assistant Dean for Information Technology and Web Science. He also serves as a trustee of the Web Science Trust in the United Kingdom. Dr. Hendler has authored over 200 technical papers in the areas of artificial intelligence, Semantic Web, agent-based computing, and Web science.

One of the early developers of the Semantic Web, he was the recipient of a 1995 Fulbright Foundation Fellowship, is a former member of the US Air Force Science Advisory Board, and is a Fellow of the IEEE, the American Association for Artificial Intelligence and the British Computer Society. Dr. Hendler is also the former chief scientist at the Information Systems Office of the US Defense Advanced Research Projects Agency (DARPA) and was awarded a US Air Force Exceptional Civilian Service Medal in 2002. He is the Editor-in-Chief emeritus of *IEEE Intelligent Systems* and is the first computer scientist to serve on the Board of Reviewing Editors for *Science* and in 2010, he was chosen as one of the 20 most innovative professors in America by *Playboy* magazine, Hendler currently serves as an "Internet Web Expert" for the US government, providing guidance to the Data.gov project.

This page intentionally left blank

CHAPTER

What is the Semantic Web?

1

CHAPTER OUTLINE

This book is about something we call the Semantic Web. From the name, you can probably guess that it is related somehow to the World Wide Web (WWW) and that it has something to do with semantics. Semantics, in turn, has to do with understanding the nature of meaning, but even the word *semantics* has a number of meanings. In what sense are we using the word *semantics?* And how can it be applied to the Web?

This book is for a working ontologist. That is, the aim of this book is not to motivate or pitch the Semantic Web but to provide the tools necessary for working with it. Or, perhaps more accurately, the World Wide Web Consortium (W3C) has provided these tools in the forms of standard Semantic Web languages, complete with abstract syntax, model-based semantics, reference implementations, test cases, and so forth. But these are like any tools—there are some basic tools that are all you need to build many useful things, and there are specialized craftsman's tools that can produce far more specializes outputs. Whichever tools are needed for a particular task, however, one still needs to understand how to use them. In the hands of someone with no knowledge, they can produce clumsy, ugly, barely functional output, but in the hands of a skilled craftsman, they can produce works of utility, beauty, and durability. It is our aim in this book to describe the craft of building Semantic Web systems. We go beyond only providing a coverage of the fundamental tools to also show how they can be used together to create semantic models, sometimes called *ontologies,* that are understandable, useful, durable, and perhaps even beautiful.

WHAT IS A WEB?

The idea of a web of information was once a technical idea accessible only to highly trained, elite information professionals: IT administrators, librarians, information architects, and the like. Since the widespread adoption of the World Wide Web, it is now common to expect just about anyone to be familiar with the idea of a web of information that is shared around the world. Contributions to this web come from every source, and every topic you can think of is covered.

Essential to the notion of the Web is the idea of an open community: Anyone can contribute their ideas to the whole, for anyone to see. It is this openness that has resulted in the astonishing comprehensiveness of topics covered by the Web. An information "web" is an organic entity that grows from the interests and energy of the communities that support it. As such, it is a hodgepodge of different analyses, presentations, and summaries of any topic that suits the fancy of anyone with the energy to publish a web page. Even as a hodgepodge, the Web is pretty useful. Anyone with the patience and savvy to dig through it can find support for just about any inquiry that interests them. But the Web often feels like it is "a mile wide but an inch deep." How can we build a more integrated, consistent, deep Web experience?

SMART WEB, DUMB WEB

Suppose you consult a web page, looking for a major national park, and you find a list of hotels that have branches in the vicinity of the park. In that list you see that Mongotel, one of the well-known hotel chains, has a branch there. Since you have a Mongotel rewards card, you decide to book your room there. So you click on the Mongotel web site and search for the hotel's location. To your surprise, you can't find a Mongotel branch at the national park. What is going on here? "That's so dumb," you tell your browsing friends. "If they list Mongotel on the national park web site, shouldn't they list the national park on Mongotel's web site?"

Suppose you are planning to attend a conference in a far-off city. The conference web site lists the venue where the sessions will take place. You go to the web site of your preferred hotel chain and find a few hotels in the same vicinity. "Which hotel in my chain is nearest to the conference?" you wonder. "And just how far off is it?" There is no shortage of web sites that can compute these distances once you give them the addresses of the venue and your own hotel. So you spend some time copying and pasting the addresses from one page to the next and noting the distances. You think to yourself, "Why should I be the one to copy this information from one page to another? Why do I have to be the one to copy and paste all this information into a single map?

Suppose you are investigating our solar system, and you find a comprehensive web site about objects in the solar system: Stars (well, there's just one of those), planets, moons, asteroids, and comets are all described there. Each object has its own web page, with photos and essential information (mass, albedo, distance from the sun, shape, size, what object it revolves around, period of rotation, period of revolution, etc.). At the head of the page is the object category: planet, moon, asteroid, comet. Another page includes interesting lists of objects: the moons of Jupiter, the named objects in the asteroid belt, the planets that revolve around the sun. This last page has the nine familiar planets, each linked to its own data page.

One day, you read in the newspaper that the International Astronomical Union (IAU) has decided that Pluto, which up until 2006 was considered a planet, should be considered a member of a new

category called a "dwarf planet"! You rush to the Pluto page and see that indeed, the update has been made: Pluto is listed as a dwarf planet! But when you go back to the "Solar Planets" page, you still see nine planets listed under the heading "Planet." Pluto is still there! "That's dumb." Then you say to yourself, "Why didn't someone update the web pages consistently?"

What do these examples have in common? Each of them has an apparent representation of data, whose presentation to the end user (the person operating the Web browser) seems "dumb." What do we mean by "dumb"? In this case, "dumb" means inconsistent, out of synchronized, and disconnected. What would it take to make the Web experience seem smarter? Do we need smarter applications or a smarter Web infrastructure?

Smart web applications

The Web is full of intelligent applications, with new innovations coming every day. Ideas that once seemed futuristic are now commonplace; search engines make matches that seem deep and intuitive; commerce sites make smart recommendations personalized in uncanny ways to your own purchasing patterns; mapping sites include detailed information about world geography, and they can plan routes and measure distances. The sky is the limit for the technologies a web site can draw on. Every information technology under the sun can be used in a web site, and many of them are. New sites with new capabilities come on the scene on a regular basis.

But what is the role of the Web infrastructure in making these applications "smart"? It is tempting to make the infrastructure of the Web smart enough to encompass all of these technologies and more. The smarter the infrastructure, the smarter the Web's performance, right? But it isn't practical, or even possible, for the Web infrastructure to provide specific support for all, or even any, of the technologies that we might want to use on the Web. Smart behavior in the Web comes from smart applications on the Web, not from the infrastructure.

So what role does the infrastructure play in making the Web smart? Is there a role at all? We have smart applications on the Web, so why are we even talking about enhancing the Web infrastructure to make a smarter Web if the smarts aren't in the infrastructure?

The reason we are improving the Web infrastructure is to allow smart applications to perform to their potential. Even the most insightful and intelligent application is only as smart as the data that is available to it. Inconsistent or contradictory input will still result in confusing, disconnected, "dumb" results, even from very smart applications. The challenge for the design of the Semantic Web is not to make a web infrastructure that is as smart as possible; it is to make an infrastructure that is most appropriate to the job of integrating information on the Web.

The Semantic Web doesn't make data smart because smart data isn't what the Semantic Web needs. The Semantic Web just needs to get the right data to the right place so the smart applications can do their work. So the question to ask is not "How can we make the Web infrastructure smarter?" but "What can the Web infrastructure provide to improve the consistency and availability of Web data?"

Connected data is smarter data

Even in the face of intelligent applications, disconnected data result in dumb behavior. But the Web data don't have to be smart; that's the job of the applications. So what can we realistically and productively expect from the data in our Web applications? In a nutshell, we want data that don't

surprise us with inconsistencies that make us want to say, "This doesn't make sense!" We don't need a smart Web infrastructure, but we need a Web infrastructure that lets us connect data to smart Web applications so that the whole Web experience is enhanced. The Web *seems* smarter because smart applications can get the data they need.

In the example of the hotels in the national park, we'd like there to be coordination between the two web pages so that an update to the location of hotels would be reflected in the list of hotels at any particular location. We'd like the two sources to stay synchronized; then we won't be surprised at confusing and inconsistent conclusions drawn from information taken from different pages of the same site.

In the mapping example, we'd like the data from the conference web site and the data from the hotels web site to be automatically understandable to the mapping web site. It shouldn't take interpretation by a human user to move information from one site to the other. The mapping web site already has the smarts it needs to find shortest routes (taking into account details like toll roads and one-way streets) and to estimate the time required to make the trip, but it can only do that if it knows the correct starting and endpoints.

We'd like the astronomy web site to update consistently. If we state that Pluto is no longer a planet, the list of planets should reflect that fact as well. This is the sort of behavior that gives a reader confidence that what they are reading reflects the state of knowledge reported in the web site, regardless of how they read it.

None of these things is beyond the reach of current information technology. In fact, it is not uncommon for programmers and system architects, when they first learn of the Semantic Web, to exclaim proudly, "I implemented something very like that for a project I did a few years back. We used…." Then they go on to explain how they used some conventional, established technology such as relational databases, XML stores, or object stores to make their data more connected and consistent. But what is it that these developers are building?

What is it about managing data this way that made it worth their while to create a whole subsystem on top of their base technology to deal with it? And where are these projects two or more years later? When those same developers are asked whether they would rather have built a flexible, distributed, connected data model support system themselves than have used a standard one that someone else optimized and supported, they unanimously chose the latter. Infrastructure is something that one would rather buy than build.

SEMANTIC DATA

In the Mongotel example, there is a list of hotels at the national park and another list of locations for hotels. The fact that these lists are intended to represent the presence of a hotel at a certain location is not explicit anywhere; this makes it difficult to maintain consistency between the two representations. In the example of the conference venue, the address appears only as text typeset on a page so that human beings can interpret it as an address. There is no explicit representation of the notion of an address or the parts that make up an address. In the case of the astronomy web page, there is no explicit representation of the status of an object as a planet. In all of these cases, the data describe the presentation of information rather than describe the entities in the world.

Could it be some other way? Can an application organize its data so that they provide an integrated description of objects in the world and their relationships rather than their presentation? The answer is

"yes," and indeed it is common good practice in web site design to work this way. There are a number of well-known approaches.

One common way to make Web applications more integrated is to back them up with a relational database and generate the web pages from queries run against that database. Updates to the site are made by updating the contents of the database. All web pages that require information about a particular data record will change when that record changes, without any further action required by the Web maintainer. The database holds information about the entities themselves, while the relationship between one page and another (presentation) is encoded in the different queries.

Consider the case of the national parks and hotel. If these pages were backed by the same database, the national park page could be built on the query "Find all hotels with location = national park," and the hotel page could be built on the query "Find all hotels from chain = Mongotel." If Mongotel has a location at the national park, it will appear on both pages; otherwise, it won't appear at all. Both pages will be consistent. The difficulty in the example given is that it is organizationally very unlikely that there could be a single database driving both of these pages, since one of them is published and maintained by the National Park Service and the other is managed by the Mongotel chain.

The astronomy case is very similar to the hotel case, in that the same information (about the classification of various astronomical bodies) is accessed from two different places, ensuring consistency of information even in the face of diverse presentation. It differs in that it is more likely that an astronomy club or university department might maintain a database with all the currently known information about the solar system.

In these cases, the Web applications can behave more robustly by adding an organizing query into the Web application to mediate between a single view of the data and the presentation. The data aren't any less dumb than before, but at least what's there is centralized, and the application or the web pages can be made to organize the data in a way that is more consistent for the user to view. It is the web page or application that behaves smarter, not the data. While this approach is useful for supporting data consistency, it doesn't help much with the conference mapping example.

Another approach to making Web applications a bit smarter is to write program code in a general-purpose language (e.g., C, Perl, Java, Lisp, Python, or XSLT) that keeps data from different places up to date. In the hotel example, such a program would update the National Park web page whenever a change is made to a corresponding hotel page. A similar solution would allow the planet example to be more consistent. Code for this purpose is often organized in a relational database application in the form of *stored procedures;* in XML applications, it can be affected using a transformational language like XSLT.

These solutions are more cumbersome to implement since they require special-purpose code to be written for each linkage of data, but they have the advantage over a centralized database that they do not require all the publishers of the data to agree on and share a single data source. Furthermore, such approaches could provide a solution to the conference mapping problem by transforming data from one source to another. Just as in the query/presentation solution, this solution does not make the data any smarter; it just puts an informed infrastructure around the data, whose job it is to keep the various data sources consistent.

The common trend in these solutions is to move away from having the presentation of the data (for human eyes) be the primary representation of the data; that is, they move from having a web site be a collection of pages to having a web site be a collection of data, from which the web page presentations are generated. The application focuses not on the presentation but on the subjects of the

presentation. It is in this sense that these applications are semantic applications; they explicitly represent the relationships that underlie the application and generate presentations as needed.

A distributed web of data

The Semantic Web takes this idea one step further, applying it to the Web as a whole. The current Web infrastructure supports a distributed network of web pages that can refer to one another with global links called Uniform Resource Locators (URLs). As we have seen, sophisticated web sites replace this structure locally with a database or XML backend that ensures consistency within that page.

The main idea of the Semantic Web is to support a distributed Web at the level of the data rather than at the level of the presentation. Instead of having one web page point to another, one data item can point to another, using global references called Uniform Resource Identifiers (URIs). The Web infrastructure provides a data model whereby information about a single entity can be distributed over the Web. This distribution allows the Mongotel example and the conference hotel example to work like the astronomy example, even though the information is distributed over web sites controlled by more than one organization. The single, coherent data model for the application is not held inside one application but rather is part of the Web infrastructure. When Mongotel publishes information about its hotels and their locations, it doesn't just publish a human-readable presentation of this information but instead a distributable, machine-readable description of the data. The data model that the Semantic Web infrastructure uses to represent this distributed web of data is called the Resource Description Framework (RDF) and is the topic of Chapter 3.

This single, distributed model of information is the contribution that the Semantic Web infrastructure brings to a smarter Web. Just as is the case with data-backed Web applications, the Semantic Web infrastructure allows the data to drive the presentation so that various web pages (presentations) can provide views into a consistent body of information. In this way, the Semantic Web helps data not be so dumb.

Features of a Semantic Web

The World Wide Web was the result of a radical new way of thinking about sharing information. These ideas seem familiar now, as the Web itself has become pervasive. But this radical new way of thinking has even more profound ramifications when it is applied to a web of data like the Semantic Web. These ramifications have driven many of the design decisions for the Semantic Web Standards and have a strong influence on the craft of producing quality Semantic Web applications.

Give me a voice ...

On the World Wide Web, publication is by and large in the hands of the content producer. People can build their own web page and say whatever they want on it. A wide range of opinions on any topic can be found; it is up to the reader to come to a conclusion about what to believe. The Web is the ultimate example of the warning *caveat emptor* ("Let the buyer beware"). This feature of the Web is so instrumental in its character that we give it a name: the *AAA Slogan:* "**A** nyone can say **A** nything about **A** ny topic."

In a web of documents, the AAA slogan means that anyone can write a page saying whatever they please, and publish it to the Web infrastructure. In the case of the Semantic Web, it means that our data

infrastructure has to allow any individual to express a piece of data about some entity in a way that can be combined with information from other sources. This requirement sets some of the foundation for the design of RDF.

It also means that the Web is like a data wilderness—full of valuable treasure, but overgrown and tangled. Even the valuable data that you can find can take any of a number of forms, adapted to its own part of the wilderness. In contrast to the situation in a large, corporate data center, where one database administrator rules with an iron hand over any addition or modification to the database, the Web has no gatekeeper. Anything and everything can grow there. A distributed web of data is an organic system, with contributions coming from all sources. While this can be maddening for someone trying to make sense of information on the Web, this freedom of expression on the Web is what allowed it to take off as a bottom-up, grassroots phenomenon.

... *So I may speak!*

In the early days of the document Web, it was common for skeptics, hearing for the first time about the possibilities of a worldwide distributed web full of hyperlinked pages on every topic, to ask, "But who is going to create all that content? Someone has to write those web pages!"

To the surprise of those skeptics, and even of many proponents of the Web, the answer to this question was that *everyone* would provide the content. Once the Web infrastructure was in place (so that Anyone could say Anything about Any topic), people came out of the woodwork to do just that. Soon every topic under the sun had a web page, either official or unofficial. It turns out that a lot of people had something to say, and they were willing to put some work into saying it. As this trend continued, it resulted in collaborative "crowdsourced" resources like Wikipedia and the Internet Movie DataBase (IMDB)—collaboratively edited information sources with broad utility.

The document Web grew because of a virtuous cycle that is called the *network effect*. In a network of contributors like the Web, the infrastructure made it *possible* for anyone to publish, but what made it *desirable* for them to do so? At one point in the Web, when Web browsers were a novelty, there was not much incentive to put a page on this new thing called "the Web"; after all, who was going to read it? Why do I want to communicate to them? Just as it isn't very useful to be the first kid on the block to have a fax machine (whom do you exchange faxes with?), it wasn't very interesting to be the first kid with a Web server.

But because a few people did have Web servers, and a few more got Web browsers, it became more attractive to have both web pages and Web browsers. Content providers found a larger audience for their work; content consumers found more content to browse. As this trend continued, it became more and more attractive, and more people joined in, on both sides. This is the basis of the network effect: The more people who are playing now, the more attractive it is for new people to start playing.

A good deal of the information that populates the Semantic Web started out on the document Web, sometimes in the form of tables, spreadsheets, or databases, and sometimes as organized group efforts like Wikipedia. Who is doing the work of converting this data to RDF for distributed access? In the earliest days of the Semantic Web there was little incentive to do so, and it was done primarily by vanguards who had an interest in Semantic Web technology itself. As more and more data is available in RDF form, it becomes more useful to write applications that utilize this distributed data. Already there are several large, public data sources available in RDF, including an RDF image of Wikipedia called dbpedia, and a surprisingly large number of government datasets. Small retailers publish

information about their offerings using a Semantic Web format called RDFa. Facebook allows content managers to provide structured data using RDFa and a format called the Open Graph Protocol. The presense of these sorts of data sources makes it more useful to produce data in linked form for the Semantic Web. The Semantic Web design allows it to benefit from the same network effect that drove the document Web.

What about the round-worlders?

The network effect has already proven to be an effective and empowering way to muster the effort needed to create a massive information network like the World Wide Web; in fact, it is the only method that has actually succeeded in creating such a structure. The AAA slogan enables the network effect that made the rapid growth of the Web possible. But what are some of the ramifications of such an open system? What does the AAA slogan imply for the content of an organically grown web?

For the network effect to take hold, we have to be prepared to cope with a wide range of variance in the information on the Web. Sometimes the differences will be minor details in an otherwise agreed-on area; at other times, differences may be essential disagreements that drive political and cultural discourse in our society. This phenomenon is apparent in the document web today; for just about any topic, it is possible to find web pages that express widely differing opinions about that topic. The ability to disagree, and at various levels, is an essential part of human discourse and a key aspect of the Web that makes it successful. Some people might want to put forth a very odd opinion on any topic; someone might even want to postulate that the world is round, while others insist that it is flat. The infrastructure of the Web must allow both of these (contradictory) opinions to have equal availability and access.

There are a number of ways in which two speakers on the Web may disagree. We will illustrate each of them with the example of the status of Pluto as a planet:

They may fundamentally disagree on some topic. While the IAU has changed its definition of *planet* in such a way that Pluto is no longer included, it is not necessarily the case that every astronomy club or even national body agrees with this categorization. Many astrologers, in particular, who have a vested interest in considering Pluto to be a planet, have decided to continue to consider Pluto as a planet. In such cases, different sources will simply disagree.

Someone might want to intentionally deceive. Someone who markets posters, models, or other works that depict nine planets has a good reason to delay reporting the result from the IAU and even to spreading uncertainty about the state of affairs.

Someone might simply be mistaken. Web sites are built and maintained by human beings, and thus they are subject to human error. Some web site might erroneously list Pluto as a planet or, indeed, might even erroneously fail to list one of the eight "nondwarf" planets as a planet.

Some information may be out of date. There are a number of displays around the world of scale models of the solar system, in which the status of the planets is literally carved in stone; these will continue to list Pluto as a planet until such time as there is funding to carve a new description for the ninth object. Web sites are not carved in stone, but it does take effort to update them; not everyone will rush to accomplish this.

While some of the reasons for disagreement might be, well, disagreeable (wouldn't it be nice if we could stop people from lying?), in practice there isn't any way to tell them apart. The infrastructure of the Web has to be able to cope with the fact that information on the Web will disagree from time to time and that this is not a temporary condition. It is in the very nature of the Web that there be variations and disagreement.

The Semantic Web is often mistaken for an effort to make everyone agree on a single ontology—but that just isn't the way the Web works. The Semantic Web isn't about getting everyone to agree, but rather about coping in a world where not everyone will agree, and achieving some degree of interoperability nevertheless. There will always be multiple ontologies, just as there will always be multiple web pages on any given topic. The Web is innovative because it allows all these multiple viewpoints to coexist.

To each their own

How can the Web infrastructure support this sort of variation of opinion? That is, how can two people say different things, about the same topic? There are two approaches to this issue. First, we have to talk a bit about how one can make any statement at all in a web context.

The IAU can make a statement in plain English about Pluto, such as "Pluto is a dwarf planet," but such a statement is fraught with all the ambiguities and contextual dependencies inherent in natural language. We think we know what "Pluto" refers to, but how about "dwarf planet"? Is there any possibility that someone might disagree on what a "dwarf planet" is? How can we even discuss such things?

The first requirement for making statements on a global web is to have a global way of identifying the entities we are talking about. We need to be able to refer to "the notion of Pluto as used by the IAU" and "the notion of Pluto as used by the American Federation of Astrologers" if we even want to be able to discuss whether the two organizations are referring to the same thing by these names.

In addition to Pluto, another object was also classified as a "dwarf planet." This object is sometimes known as UB313 and sometimes known by the name Xena. How can we say that the object known to the IAU as UB313 is the same object that its discoverer Michael Brown calls "Xena"?

One way to do this would be to have a global arbiter of names decide how to refer to the object. Then Brown and the IAU can both refer to that "official" name and say that they use a private "nickname" for it. Of course, the IAU itself is a good candidate for such a body, but the process to name the object has taken over two years. Coming up with good, agreed-on global names is not always easy business.

In the absence of such an agreement, different Web authors will select different URIs for the same real-world resource. Brown's Xena is IAU's UB313. When information from these different sources is brought together in the distributed network of data, the Web infrastructure has no way of knowing that these need to be treated as the same entity. The flip side of this is that we cannot assume that just because two URIs are distinct, they refer to distinct resources. This feature of the Semantic Web is called the *Nonunique Naming Assumption;* that is, we have to assume (until told otherwise) that some Web resource might be referred to using different names by different people. It's also crucial to note that there are times when unique names might be nice, but it may be impossible. Some other organization than the IAU, for example, might decide they are unwilling to accept the new nomenclature.

There's always one more

In a distributed network of information, as a rule we cannot assume at any time that we have seen all the information in the network, or even that we know everything that has been asserted about one single topic. This is evident in the history of Pluto and UB313. For many years, it was sufficient to say that a *planet* was defined as "any object of a particular size orbiting the sun." Given the information available during that time, it was easy to say that there were nine planets around the sun. But the new information about UB313 changed that; if a planet is defined to be any body that orbits the sun of a particular size, then UB313 had to be considered a planet, too. Careful speakers in the late twentieth century, of course, spoke of the "known" planets, since they were aware that another planet was not only possible but even suspected (the so-called "Planet X," which stood in for the unknown but suspected planet for many years).

The same situation holds for the Semantic Web. Not only might new information be discovered at any time (as is the case in solar system astronomy), but, because of the networked nature of the Web, at any one time a particular server that holds some unique information might be unavailable. For this reason, on the Semantic Web we can rarely conclude things like "there are nine planets," since we don't know what new information might come to light.

In general, this aspect of a Web has a subtle but profound impact on how we draw conclusions from the information we have. It forces us to consider the Web as an *Open World* and to treat it using the *Open World Assumption.* An Open World in this sense is one in which we must assume at any time that new information could come to light, and we may draw no conclusions that rely on assuming that the information available at any one point is all the information available.

For many applications, the Open World Assumption makes no difference; if we draw a map of all the Mongotel hotels in Boston, we get a map of all the ones we know of at the time. The fact that Mongotel might have more hotels in Boston (or might open a new one) does not invalidate the fact that it has the ones it already lists. In fact, for a great deal of Semantic Web applications, we can ignore the Open World Assumption and simply understand that a semantic application, like any other web page, is simply reporting on the information it was able to access at one time.

The openness of the Web only becomes an issue when we want to draw conclusions based on distributed data. If we want to place Boston in the list of cities that are not served by Mongotel (e.g., as part of a market study of new places to target Mongotels), then we cannot assume that just because we haven't found a Mongotel listing in Boston, no such hotel exists.

As we shall see in the following chapters, the Semantic Web includes features that correspond to all the ways of working with Open Worlds that we have seen in the real world. We can draw conclusions about missing Mongotels if we say that some list is a comprehensive list of all Mongotels. We can have an anonymous "Planet X" stand in for an unknown but anticipated entity. These techniques allow us to cope with the Open World Assumption in the Semantic Web, just as they do in the Open World of human knowledge.

When will the Semantic Web arrive? It already has. In selecting candidate examples for this second edition, we had to pick and choose from a wide range of Semantic Web deployments. We devote two chapters to in-depth studies of these deployments "in the wild." In Chapter 9, we see how the US government shares data about its operations in a flexible way and how Facebook uses the Semantic Web to link pages from all over the web into its network. Chapter 13 shows how the Semantic Web is used by thousands of e-commerce web pages to make information available to mass markets through

major search engines and how scientific communities share key information about engineering, chemistry, and biology. The Semantic Web is here today.

SUMMARY

The aspects of the Web we have outlined here—the AAA slogan, the network effect, nonunique naming, and the Open World Assumption—already hold for the document Web. As a result, the Web today is something of an unruly place, with a wide variety of different sources, organizations, and styles of information. Effective and creative use of search engines is something of a craft; efforts to make order from this include community efforts like social bookmarking and community encyclopedias to automated methods like statistical correlations and fuzzy similarity matches.

For the Semantic Web, which operates at the finer level of individual statements about data, the situation is even wilder. With a human in the loop, contradictions and inconsistencies in the document Web can be dealt with by the process of human observation and application of common sense. With a machine combining information, how do we bring any order to the chaos? How can one have any confidence in the information we merge from multiple sources? If the document Web is unruly, then surely the Semantic Web is a jungle—a rich mass of interconnected information, without any road map, index, or guidance.

How can such a mess become something useful? That is the challenge that faces the working ontologist. Their medium is the distributed web of data; their tools are the Semantic Web languages RDF, RDFS, SPARQL, SKOS, and OWL. Their craft is to make sensible, usable, and durable information resources from this medium. We call that craft *modeling,* and it is the centerpiece of this book.

The cover of this book shows a system of channels with water coursing through them. If we think of the water as the data on the Web, the channels are the model. If not for the model, the water would not flow in any systematic way; there would simply be a vast, undistinguished expanse of water. Without the water, the channels would have no dynamism; they have no moving parts in and of themselves. Put the two together, and we have a dynamic system. The water flows in an orderly fashion, defined by the structure of the channels. This is the role that a model plays in the Semantic Web.

Without the model, there is an undifferentiated mass of data; there is no way to tell which data can or should interact with other data. The model itself has no significance without data to describe it. Put the two together, however, and you have a dynamic web of information, where data flow from one point to another in a principled, systematic fashion. This is the vision of the Semantic Web—an organized worldwide system where information flows from one place to another in a smooth but orderly way.

Fundamental concepts

The following fundamental concepts were introduced in this chapter.

The AAA slogan—Anyone can say Anything about Any topic. One of the basic tenets of the Web in general and the Semantic Web in particular.

Open world/Closed world—A consequence of the AAA slogan is that there could always be something new that someone will say; this means that we must assume that there is always more information that could be known.

Nonunique naming—Since the speakers on the Web won't necessarily coordinate their naming efforts, the same entity could be known by more than one name.
The network effect—The property of a web that makes it grow organically. The value of joining in increases with the number of people who have joined, resulting in a virtuous cycle of participation.
The data wilderness—The condition of most data on the web. It contains valuable information, but there is no guarantee that it will be orderly or readily understandable.

CHAPTER

Semantic modeling

2

CHAPTER OUTLINE

What would you call a world in which any number of people can speak, when you never know who has something useful to say, and when someone new might come along at any time and make a valuable but unexpected contribution? What if just about everyone had the same goal of advancing the collaborative state of knowledge of the group, but there was little agreement (at first, anyway) about how to achieve it?

If your answer is "That sounds like the Semantic Web!" you are right (and you must have read Chapter 1). If your answer is "It sounds like any large group trying to understand a complex phenomenon," you are even more right. The jungle that is the Semantic Web is not a new thing; this sort of chaos has existed since people first tried to make sense of the world around them.

What intellectual tools have been successful in helping people sort through this sort of tangle? Any number of analytical tools has been developed over the years, but they all have one thing in common: They help people understand their world by forming an abstract description that hides certain details while illuminating others. These abstractions are called *models,* and they can take many forms.

How do models help people assemble their knowledge? Models assist in three essential ways:

1. *Models help people communicate.* A model describes the situation in a particular way that other people can understand.
2. *Models explain and make predictions.* A model relates primitive phenomena to one another and to more complex phenomena, providing explanations and predictions about the world.
3. *Models mediate among multiple viewpoints.* No two people agree completely on what they want to know about a phenomenon; models represent their commonalities while allowing them to explore their differences.

The Semantic Web standards have been created not only as a medium in which people can collaborate by sharing information but also as a medium in which people can collaborate on models. Models that they can use to organize the information that they share. Models that they can use to advance the common collection of knowledge.

How can a model help us find our way through the mess that is the Web? How do these three features help? The first feature, human communication, allows people to collaborate on their understanding. If someone else has faced the same challenge that you face today, perhaps you can learn from their experience and apply it to yours. There are a number of examples of this in the Web today, of newsgroups, mailing lists, and wikis where people can ask questions and get answers. In the case in which the information needs are fairly uniform, it is not uncommon for a community or a company to assemble a set of "Frequently Asked Questions," or FAQs, that gather the appropriate knowledge as answers to these questions. As the number of questions becomes unmanageable, it is not uncommon to group them by topic, by task, by affected subsystem, and so forth. This sort of activity, by which information is organized for the purpose of sharing, is the simplest and most common kind of modeling, with the sole aim of helping a group of people collaborate in their effort to sort through a complex set of knowledge.

The second feature, explanation and prediction, helps individuals make their own judgments based on information they receive. FAQs are useful when there is a single authority that can give clear answers to a question, as is the case for technical assistance for using some appliance or service. But in more interpretive situations, someone might want or need to draw a conclusion for themselves. In such a situation, a simple answer as given in a FAQ is not sufficient. Politics is a common example from everyday life. Politicians in debate do not tell people how to vote, but they try to convince them to vote in one way or another. Part of that convincing is done by explaining their position and allowing the individual to evaluate whether that explanation holds true to their own beliefs about the world. They also typically make predictions: If we follow this course of action, then a particular outcome will follow. Of course, a lot more goes into political persuasion than the argument, but explanation and prediction are key elements of a persuasive argument.

Finally, the third feature, mediation of multiple viewpoints, is essential to fostering understanding in a web environment. As the web of opinions and facts grows, many people will say things that disagree slightly or even outright contradict what others are saying. Anyone who wants to make their way through this will have to be able to sort out different opinions, representing what they have in common as well as the ways in which they differ. This is one of the most essential organizing principles of a large, heterogeneous knowledge set, and it is one of the major contributions that modeling makes to helping people organize what they know.

Astrologers and the IAU agree on the planethood of Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune. The IAU also agrees with astrologers that Pluto is a planet, but it disagrees by calling it a dwarf planet. Astrologers (or classical astronomers) do not accept the concept of dwarf planets, so they are not in agreement with the IAU, which categorizes UB313 and Ceres as such. A model for the Semantic Web must be able to organize this sort of variation, and much more, in a meaningful and manageable way.

MODELING FOR HUMAN COMMUNICATION

Models used for human communication have a great advantage over models that are intended for use by computers; they can take advantage of the human capacity to interpret signs to give them meaning. This means that communication models can be written in a wide variety of forms, including plain language or ad hoc images. A model can be explained by one person, amended by another, interpreted

by a third person, and so on. Models written in natural language have been used in all manner of intellectual life, including science, religion, government, and mathematics.

But this advantage is a double-edged sword; when we leave it to humans to interpret the meaning of a model, we open the door for all manner of abuse, both intentional and unintentional. Legislation provides a good example of this. A governing body like a parliament or a legislature enacts laws that are intended to mediate rights and responsibilities between various parties. Legislation typically sets up some sort of model of a situation, perhaps involving money (e.g., interest caps, taxes); access rights (who can view what information, how can information be legally protected); personal freedom (how freely can one travel across borders, when does the government have the right to restrict a person's movements); or even the structure of government itself (who can vote and how are those votes counted, how can government officials be removed from office). These models are painstakingly written in natural language and agreed on through an elaborate process (which is also typically modeled in natural language).

It is well known to anyone with even a passing interest in politics that good legislation is not an easy task and that crafting the words carefully for a law or statute is very important. The same flexibility of interpretation that makes natural language models so flexible also makes it difficult to control how the laws will be interpreted in the future. When someone else reads the text, they will have their own background and their own interests that will influence how they interpret any particular model. This phenomenon is so widespread that most government systems include a process (usually involving a court magistrate and possibly a committee of citizens) whereby disputes over the interpretation of a law or its applicability can be resolved.

When a model relies on particulars of the context of its reader for interpretation of its meaning, as is the case in legislation, we say that a model is *informal.* That is, the model lacks a formalism whereby the meaning of terms in the model can be uniquely defined.

In the document web today, there are informal models that help people communicate about the organization of the information. It is common for commerce web sites to organize their wares in catalogs with category names like "web-cams," "Oxford shirts," and "Granola." In such cases, the communication is primarily one way; the catalogue designer wants to communicate to the buyers the information that will help them find what they want to buy. The interpretation of these words is up to the buyers. The effectiveness of such a model is measured by the degree to which this is successful. If enough people interpret the categories in a way similar enough to the intent of the cataloguer, then they will find what they want to buy. There will be the occasional discrepancy like "Why wasn't that item listed as a *webcam?"* or "That's not granola, that's just plain cereal!" But as long as the interpretation is close enough, the model is successful.

A more collaborative style of document modeling comes in the form of community tagging. A number of web sites have been successful by allowing users to provide meaningful symbolic descriptions of their content in the form of *tags.* A tag in this sense is simply a single word or short phrase that describes some aspect of the content. Examples of tagging systems include Flickr for photos and del.icio.us for Web bookmarks. The idea of community tagging is that each individual who provides content will describe it using tags of their own choosing. If any two people use the same tag, this becomes a common organizing entity; anyone who is browsing for content can access information from both contributors under that tag. The tagging infrastructure shows which tags have been used by many people. Not only does this help browsers determine what tags to use in a search, but it also helps content providers to find commonly used tags that they might want to use to describe new content. Thus,

a tagging system will have a certain self-organizing character, whereby popular tags become more popular and unpopular tags remain unpopular—something like evolution by artificial selection of tags.

Tagging systems of this sort provide an informal organization to a large body of heterogeneous information. The organization is informal in the sense that the interpretation of the tags requires human processing in the context of the consumer. Just because a tag is popular doesn't mean that everyone is using it in the same way. In fact, the community selection process actually selects tags that are used in several different ways, whether they are compatible or not. As more and more people provide content, the popular tags saturate with a wide variety of content, making them less and less useful as discriminators for people browsing for content. This sort of problem is inherent in information modeling systems; since there isn't an objective description of the meaning of a symbol outside the context of the provider and consumer of the symbol, the communication power of that symbol degrades as it is used in more and more contexts.

Formality of a model isn't a black-and-white judgment; there can be degrees of formality. This is clear in legal systems, where it is common to have several layers of legislation, each one giving objective context for the next. A contract between two parties is usually governed by some regional law that provides standard definitions for terms in the contract. Regional laws are governed by national laws, which provide constraints and definitions for their terms. National laws have their own structure, in which a constitution or a body of case law provides a framework for new decisions and legislation. Even though all these models are expressed in natural language and fall back on human interpretation in the long run, they can be more formal than private agreements that rely almost entirely on the interpretation of the agreeing parties.

This layering of informal models sometimes results in a modeling style that is reminiscent of Talmudic scholarship. The content of the Talmud includes not only the original scripture but also interpretative comments on the scripture by authoritative sources (classical rabbis). Their comments have gained such respect that they are traditionally published along with the original scripture for comment by later rabbis, whose comments in turn have become part of the intellectual tradition. The original scripture, along with all the authoritative comments, is collectively called the Talmud, and it is the basis of a classical Jewish education to this day.

A similar effect happens with informal models. The original model is appropriate in some context, but as its use expands beyond that context, further models are required to provide common context to explicate the shared meaning. But if this further exposition is also informal, then there is the risk that its meaning will not be clear, so further modeling must be done to clarify that. This results in heavily layered models, in which the meaning of the terms is always subject to further interpretation. It is the inherent ambiguity of natural language at each level that makes the next layer of commentary necessary until the degree of ambiguity is "good enough" that no more levels are needed. When it is possible to choose words that are evocative and have considerable agreement, this process converges much more quickly.

Human communication, as a goal for modeling, allows it to play a role in the ongoing collection of human knowledge. The levels of communication can be quite sophisticated, including the collection of information used to interpret other information. In this sense, human communication is the fundamental requirement for building a Semantic Web. It allows people to contribute to a growing body of knowledge and then draw from it. But communication is not enough; to empower a web of human knowledge, the information in a model needs to be organized in such a way that it can be useful to a wide range of consumers.

EXPLANATION AND PREDICTION

Models are used to organize human thought in the form of explanations. When we understand how a phenomenon results from other basic principles, we gain a number of advantages. Not least is the feeling of confidence that we have actually understood it; people often claim to "have a grasp on" or "have their head around" an idea when they finally understand it. Explanation plays a major role in this sort of understanding. Explanation also assists in memory; it is easier to remember that putting a lid on a flaming pot can quench the flame if one knows the explanation that fire requires air to burn. Most important for the context of the Semantic Web, explanation makes it easier to reuse a model in whole or in part; an explanation relates a conclusion to more basic principles. Understanding how a pot lid quenches a fire can help one understand how a candle snuffer works. Explanation is the key to understanding when a model is applicable and when it is not.

Closely related to this aspect of a model is the idea of prediction. When a model provides an adequate explanation of a phenomenon, it can also be used to make predictions. This aspect of models is what makes their use central to the scientific method, where falsification of predictions made by models forms the basis of the methodology of inquiry.

Explanation and prediction typically require models with a good deal more formality than is usually required for human communication. An explanation relates a phenomenon to "first principles"; these principles, and the rules by which they are related, do not depend on interpretation by the consumer but instead are in some objective form that stands outside the communication. Such an objective form, and the rules that govern how it works, is called a *formalism.*

Formal models are the bread and butter of mathematical modeling, in which very specific rules for calculation and symbol manipulation govern the structure of a mathematical model and the valid ways in which one item can refer to another. Explanations come in the form of proofs, in which steps from premises (stated in some formalism) to conclusions are made according to strict rules of transformation for the formalism. Formal models are used in many human intellectual endeavors, wherever precision and objectivity are required.

Formalisms can also be used for predictions. Given a description of a situation in some formalism, the same rules that govern transformations in proofs can be used to make predictions. We can explain the trajectory of an object thrown out of a window with a formal model of force, gravity, speed, and mass, but given the initial conditions of the object thrown, we can also compute, and thus predict, its trajectory.

Formal prediction and explanation allow us to evaluate when a model is applicable. Furthermore, the formalism allows that evaluation to be independent of the listener. One can dispute the result that $2 + 2 = 4$ by questioning just what the terms "2," "4," "+," and "=" mean, but once people agree on what they mean, they cannot (reasonably) dispute that this formula is correct.

Formal modeling therefore has a very different social dynamic than informal modeling; because there is an objective reference to the model (the formalism), there is no need for the layers of interpretation that result in Talmudic modeling. Instead of layers and layers of interpretation, the buck stops at the formalism.

As we shall see, the Semantic Web standards include a small variety of modeling formalisms. Because they are formalisms, modeling in the Semantic Web need not become a process of layering interpretation on interpretation. Also, because they are formalisms, it is possible to couch explanations in the Semantic

Web in the form of proofs and to use that proof mechanism to make predictions. This aspect of Semantic Web models goes by the name *inference* and it will be discussed in detail in Chapter 5.

MEDIATING VARIABILITY

In any Web setting, variability is to be expected and even embraced. The dynamics of the network effect require the ability to represent a variety of opinions. A good model organizes those opinions so that the things that are common can be represented together, while the things that are distinct can be represented as well.

Let's take the case of Pluto as an example. From 1930 until 2006, it was considered to be a planet by astronomers and astrologers alike. After the redefinition of *planet* by the IAU in 2006, Pluto was no longer considered to be a planet but more specifically a *dwarf planet* by the IAU and by astronomers who accept the IAU as an authority. Astrologers, however, chose not to adopt the IAU convention, and they continued to consider Pluto a planet. Some amateur astronomers, mostly for nostalgic reasons, also continued to consider Pluto a planet. How can we accommodate all of these variations of opinion on the Web?

One way to accommodate them would be to make a decision as to which one is "preferred" and to control the Web so that only that position is supported. This is the solution that is most commonly used in corporate data centers, where a small group or even a single person acts as the database administrator and decides what data are allowed to live in the corporate database. This solution is not appropriate for the Web because it does not allow for the AAA slogan (see Chapter 1) that leads to the network effect.

Another way to accommodate these different viewpoints would be to simply allow each one to be represented separately, with no reference to one another at all. It would be the responsibility of the information consumer to understand how these things relate to one another and to make any connections as appropriate. This is the basis of an informal approach, and it indeed describes the state of the document web as it is today. A Web search for Pluto will turn up a wide array of articles, in which some call it a planet (e.g., astrological ones or astronomical ones that have not been updated), some call it a dwarf planet (IAU official web sites), and some that are still debating the issue. The only way a reader can come to understand what is common among these things—the notion of a planet, of the solar system, or even of Pluto itself—is through reader interpretation.

How can a model help sort this out? How can a model describe what is common about the astrological notion of a planet, the twentieth-century astronomical notion of a planet, and the post-2006 notion of a planet? The model must also allow for each of these differing viewpoints to be expressed.

Variation and classes

This problem is not a new one; it is a well-known problem in software engineering. When a software component is designed, it has to provide certain functionality, determined by information given to it at runtime. There is a trade-off in such a design; the component can be made to operate in a wide variety of circumstances, but it will require a complex input to describe just how it should behave at any one time. Or the system could be designed to work with very simple input but be useful in only a small number of very specific situations. The design of a software component inherently involves a model of the commonality and variability in the environment in which it is expected to be deployed. In response

to this challenge, software methodology has developed the art of object modeling (in the context of Object-Oriented Programming, or OOP) as a means of organizing commonality and variability in software components.

One of the primary organizing tools in OOP is the notion of a hierarchy of classes and subclasses. Classes high up in the hierarchy represent functionality that is common to a large number of components; classes farther down in a hierarchy represent more specific functionality. Commonality and variability in the functionality of a set of software components is represented in a class hierarchy.

The Semantic Web standards also use this idea of class hierarchy for representing commonality and variability. Since the Semantic Web, unlike OOP, is not focused on software representation, classes are not defined in terms of behaviors of functions. But the notion of classes and subclasses remains, and it plays much the same role. High-level classes represent commonality among a large variety of entities, whereas lower-level classes represent commonality among a small, specific set of things.

Let's take Pluto as an example. The 2006 IAU definition of *planet* is quite specific in requiring these three criteria for a celestial body to be considered a planet:

1. It is in orbit around the sun.
2. It has sufficient mass to be nearly round.
3. It has cleared the neighborhood around its orbit.

The IAU goes further to state that a dwarf planet is a body that satisfies conditions 1 and 2 (and not 3); a body that satisfies only condition 1 is a *small solar system body (SSSB).* These definitions make a number of things clear: The classes SSSB, dwarf planet, and planet are all mutually exclusive; no body is a member of any two classes. However, there is something that all of them have in common: They all are in orbit around the sun.

Twentieth-century astronomy and astrology are not quite as organized as this; they don't have such rigorous definitions of the word *planet.* So how can we relate these notions to the twenty-first-century notion of *planet?*

The first thing we need is a way to talk about the various uses of the word *planet:* the IAU use, the astrological use, and the twentieth-century astronomical use. This seems like a simple requirement, but until it is met, we can't even talk about the relationship among these terms. We will see details of the Semantic Web solution to this issue in Chapter 3, but for now, we will simply prefix each term with a short abbreviation of its source—for example, use `IAU:Planet` for the IAU use of the word, `horo:Planet` for the astrological use, and `astro:Planet` for the twentieth-century astronomical use.

The solution begins by noticing what it is that all three notions of *planet* have in common; in this case, it is that the body orbits the sun. Thus, we can define a class of the things that orbit the sun, which we may as well call *solar system body*, or *SSB* for short. All three notions are subclasses of this notion. This can be depicted graphically as in Figure 2.1.

We can go further in this modeling when we observe that there are only eight `IAU:Planets`, and each one is also a `horo:Planet` and an `astro:Planet`. Thus, we can say that `IAU:Planet` is a subclass of both `horo:Planet` and `astro:Planet`, as shown in Figure 2.2. We can continue in this way, describing the relationships among all the concepts we have mentioned so far: `IAU:DwarfPlanet` and `IAU:SSSB`. As we go down the tree, each class refers to a more restrictive set of entities. In this way, we can model the commonality among entities (at the high level) while respecting their variation (at a low level).

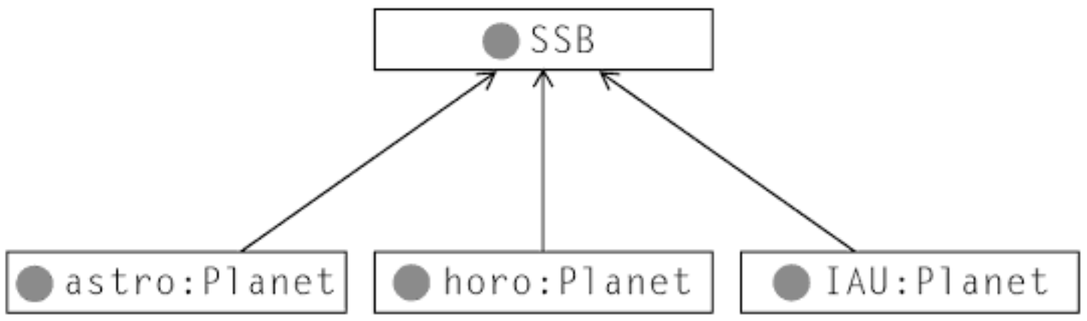


FIGURE 2.1

Subclass diagram for different notions of *planet*.

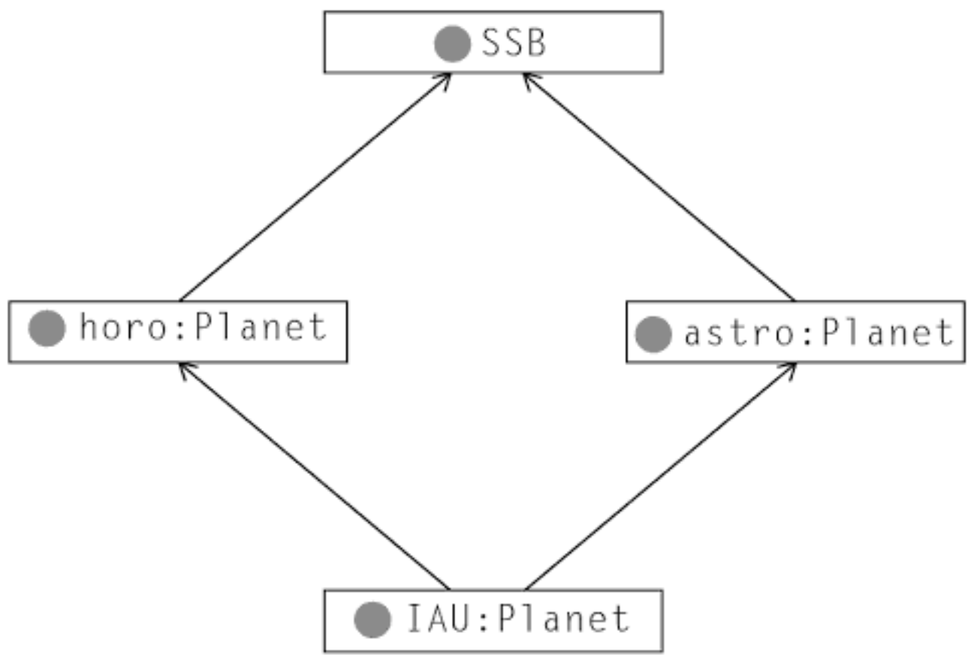


FIGURE 2.2

More detailed relationships between various notions of *planet*.

Variation and layers

Classes and subclasses are a fine way to organize variation when there is a simple, known relationship between the modeled entities and it is possible to determine a clear ordering of classes that describes these relationships. In a Web setting, however, this usually is not the case. Each contributor can have something new to say that may fit in with previous statements in a wide variety of ways. How can we accommodate variation of sources if we can't structure the entities they are describing into a class model?

The Semantic Web provides an elegant solution to this problem. The basic idea is that any model can be built up from contributions from multiple sources. One way of thinking about this is to consider a model to be described in layers. Each layer comes from a different source. The entire model is the combination of all the layers, viewed as a single, unified whole.

Let's have a look at how this could work in the case of Pluto. Figure 2.3 illustrates how different communities could assert varying information about Pluto. In part (a) of the figure, we see some information about Pluto that is common among astrologers—namely, that Pluto signifies rebirth and regeneration and that the preferred symbol for referring to Pluto is the glyph indicated. Part (b) shows some information that is of concern to astronomers, including the composition of the body Pluto and their preferred symbol. How can this variation be accommodated in a web of information? The simplest way is to simply merge the two models into a single one that includes all the information from each model, as shown in part (c).

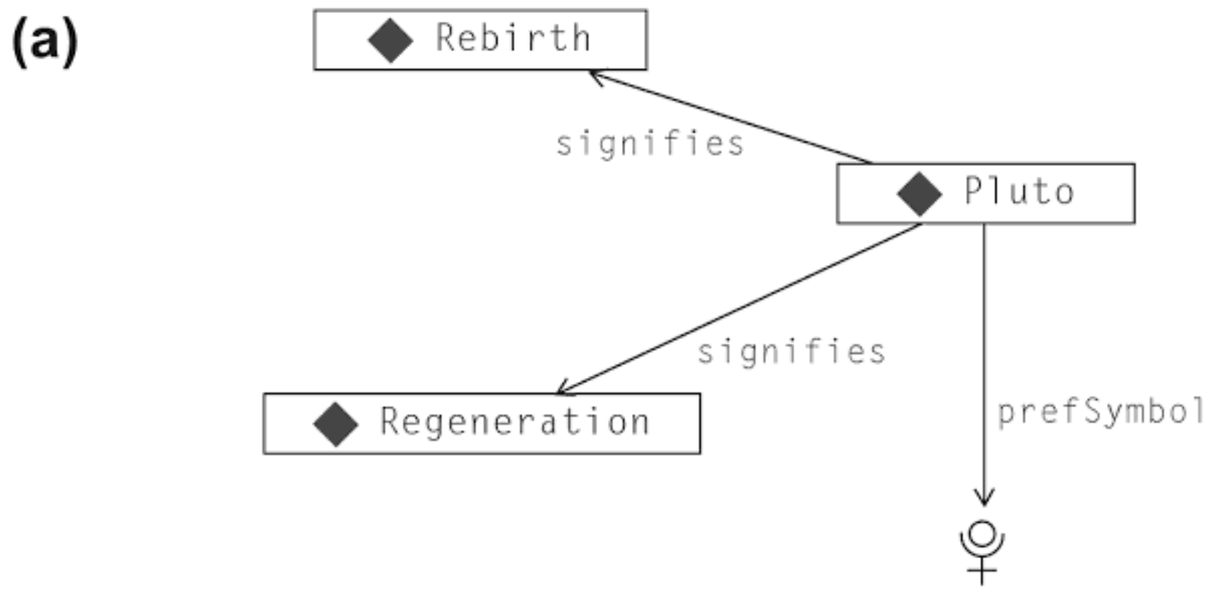


FIGURE 2.3

Layers of modeled information about Pluto.

Merging models in this way is a conceptually simple thing to do, but how does it cope with variability? In the first place, it copes in the simplest way possible: It allows the astrologers and the astronomers to both have their say about Pluto (remember the AAA slogan!). For any party that is interested in both of these things (perhaps someone looking for a spiritual significance for elements?), the information can be viewed as a single, unified whole.

But merging models in this way has a drawback as well. In Figure 2.3(c), there are two distinct glyphs, each claiming to be the "preferred" symbol for Pluto. This brings up issues of consistency of viewpoints. On the face of it, this appears to be an inconsistency because, from its name, we might expect that there can be exactly one preferred symbol (`prefSymbol`) for any body. But how can a machine know that? For a machine, the name `prefSymbol` can't be treated any differently from any other label—for instance, `madeOf` or `signifies`. In such a context, how can we even tell that this is an inconsistency? After all, we don't think it is an inconsistency that Pluto can be composed of more than one chemical compound or that it can signify more than one spiritual theme. Do we have to describe this in a natural language commentary on the model?

Detailed answers to questions like these are exactly the reason why we need to publish models on the Semantic Web. When two (or more!) viewpoints come together in a web of knowledge, there will typically be overlap, disagreement, and confusion before there is synergy, cooperation, and collaboration. If the infrastructure of the Web is to help us to find our way through the wild stage of information sharing, an informal notion of how things fit together, or should fit together, will not suffice. It is easy enough to say that we have an intuition that states there is something special about `prefSymbol` that makes it different from `madeOf` or `signifies`. If we can inform our infrastructure about this distinction in a sufficiently formal way, then it can, for instance, detect discrepancies of this sort and, in some cases, even resolve them.

This is the essence of modeling in the Semantic Web: providing an infrastructure where not only can anyone say anything about any topic but the infrastructure can help a community work through the resulting chaos. A model can provide a framework (like classes and subclasses) for representing and organizing commonality and variability of viewpoints when they are known. But in advance of such an organization, a model can provide a framework for describing what sorts of things we can say about something. We might not agree on the symbol for Pluto, but we can agree that it should have just one preferred symbol.

EXPRESSIVITY IN MODELING

There is a trade-off when we model, and although anyone can say anything about any topic, not everyone will want to say certain things. There are those who are interested in saying details about individual entities, like the preferred symbol for Pluto or the themes in life that it signifies. Others (like that IAU) are interested in talking about categories, what belongs in a category, and how you can tell the difference. Still others (like lexicographers, information architects, and librarians) want to talk about the rules for specifying information, such as whether there can be more than one preferred label for any entity. All of these people have contributions to make to the web of knowledge, but the kinds of contributions they make are very different, and they need different tools. This difference is one of *level of expressivity.*

The idea of different levels of expressivity is as well known in the history of collaborative human knowledge as modeling itself. Take as an example the development of models of a water molecule, as

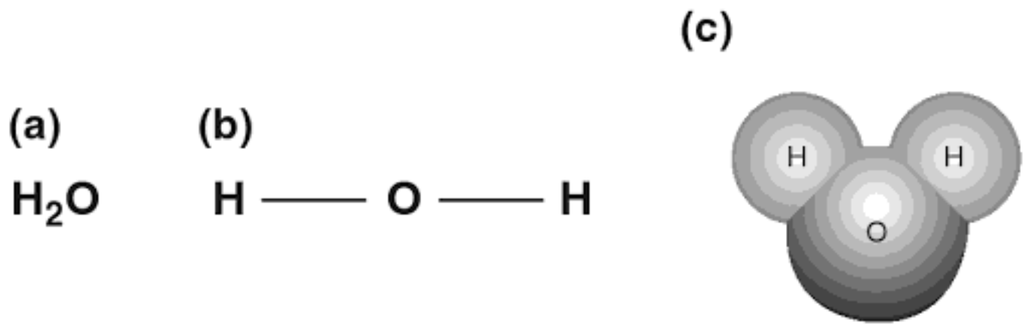


FIGURE 2.4

Different expressivity of models of a water molecule.

shown in Figure 2.4. In part (a), we see a model of the water molecule in terms of the elements that make up the molecule and how many of each is present—namely, two hydrogen atoms and one oxygen atom. This model expresses important information about the molecule, and it can be used to answer a number of basic questions about water, such as calculating the mass of the molecule (given the masses of its component atoms) and what components would have to be present to be able to construct water from constituent parts.

In Figure 2.4(b), we see a model with more expressivity. Not only does this model identify the components of water and their proportions, but it also shows how they are connected in the chemical structure of the molecule. The oxygen molecule is connected to each of the hydrogen molecules, which are not (directly) connected to one another at all. This model is somewhat more expressive than the model in part (a); it can answer further questions about the molecule. From (b), it is clear that when the water molecule breaks down into smaller molecules, it can break into single hydrogen atoms (H) or into oxygen-hydrogen ions (OH) but not into double-hydrogen atoms (H_2) without some recombination of components after the initial decomposition.

Finally, the model shown in Figure 2.4(c) is more expressive still in that it shows not only the chemical structure of the molecule but also the physical structure. The fact that the oxygen atom is somewhat larger than the hydrogen atoms is shown in this model. Even the angle between the two hydrogen atoms as bound to the oxygen atom is shown. This information is useful for working out the geometry of combinations of water molecules, as is the case, for instance, in the crystalline structure of ice.

Just because one model is more expressive than another does not make it superior; different expressive modeling frameworks are different tools for different purposes. The chemical formula for water is simpler to determine than the more expressive, but more complex, models, and it is useful for resolving a wide variety of questions about chemistry. In fact, most chemistry textbooks go for quite a while working only from the chemical formulas without having to resort to more structural models until the course covers advanced topics.

The Semantic Web provides a number of modeling languages that differ in their level of expressivity; that is, they constitute different tools that allow different people to express different sorts of information. In the rest of this book, we will cover these modeling languages in detail. The Semantic Web standards are organized so that each language level builds on the one before so the languages themselves are layered. The following are the languages of the Semantic Web from least expressive to most expressive.

RDF—The Resource Description Framework. This is the basic framework that the rest of the Semantic Web is based on. RDF provides a mechanism for allowing anyone to make a basic

statement about anything and layering these statements into a single model. Figure 2.3 shows the basic capability of merging models in RDF. RDF has been a recommendation from the W3C since 1999.

RDFS—The RDF Schema language. RDFS is a language with the expressivity to describe the basic notions of commonality and variability familiar from object languages and other class systems—namely classes, subclasses, and properties. Figures 2.1 and 2.2 illustrated the capabilities of RDFS. RDFS has been a W3C recommendation since 2004.

RDFS-Plus. RDFS-Plus is a subset of OWL that is more expressive than RDFS but without the complexity of OWL. There is no standard in progress for RDFS-Plus, but there is a growing awareness that something between RDFS and OWL could be industrially relevant. We have selected a particular subset of OWL functionality to present the capabilities of OWL incrementally. RDFS-Plus includes enough expressivity to describe how certain properties can be used and how they relate to one another. RDFS-Plus is expressive enough to show the utility of certain constructs beyond RDFS, but it lacks the complexity that makes OWL daunting to many beginning modelers. The issue of uniqueness of the preferred symbol is an example of the expressivity of RDFS-Plus.

OWL. OWL brings the expressivity of logic to the Semantic Web. It allows modelers to express detailed constraints between classes, entities, and properties. OWL was adopted as a recommendation by the W3C in 2004, with a second version adopted in 2009.

SUMMARY

The Semantic Web, just like the document web that preceded it, is based on some radical notions of information sharing. These ideas—the AAA slogan, the open world assumption, and nonunique naming—provide for an environment in which information sharing can thrive and a network effect of knowledge synergy is possible. But this style of information gathering creates a chaotic landscape rife with confusion, disagreement, and conflict. How can the infrastructure of the Web support the development from this chaotic state to one characterized by information sharing, cooperation, and collaboration?

The answer to this question lies in modeling. Modeling is the process of organizing information for community use. Modeling supports this in three ways: It provides a framework for human communication, it provides a means for explaining conclusions, and it provides a structure for managing varying viewpoints. In the context of the Semantic Web, modeling is an ongoing process. At any point in time, some knowledge will be well structured and understood, and these structures can be represented in the Semantic Web modeling language. At the same time, other knowledge will still be in the chaotic, discordant stage, where everyone is expressing himself differently. And typically, as different people provide their own opinions about any topic under the sun, the Web will simultaneously contain organized and unorganized knowledge about the very same topic. The modeling activity is the activity of distilling communal knowledge out of a chaotic mess of information. This was nicely illustrated in the Pluto example.

The next several chapters of the book introduce each of the modeling languages of the Semantic Web and illustrate how they approach the challenges of modeling in a Semantic Web context. For each

modeling language—RDF, RDFS, and OWL—we will describe the technical details of how the language works, with specific examples "in the wild" of the standard in use.

Fundamental concepts

The following fundamental concepts were introduced in this chapter.

Modeling—Making sense of unorganized information.
Formality/Informality—The degree to which the meaning of a modeling language is given independent of the particular speaker or audience.
Commonality* and *Variability—When describing a set of things, some of them will have somem things in common (commonality), and some will have important differences (variability). Managing commonality and variability is a fundamental aspect of modeling in general, and of Semantic Web models in particular.
Expressivity—The ability of a modeling language to describe certain aspects of the world. More expressive modeling language can express a wider variety of statements about the model. Modeling languages of the Semantic *Web—RDF, RDFS,* and OWL—differ in their levels of expressivity.

This page intentionally left blank

CHAPTER

3 RDF—The basis of the Semantic Web

CHAPTER OUTLINE

RDF, RDFS, and OWL are the basic representation languages of the Semantic Web, with RDF serving as the foundation. RDF addresses one fundamental issue in the Semantic Web: managing distributed data. All other Semantic Web standards build on this foundation of distributed data. RDF relies heavily on the infrastructure of the Web, using many of its familiar and proven features, while extending them to provide a foundation for a distributed network of data.

The Web that we are accustomed to is made up of documents that are linked to one another. Any connection between a document and the thing(s) in the world it describes is made only by the person who reads the document. There could be a link from a document about Shakespeare to a document about Stratford-upon-Avon, but there is no notion of an entity that is Shakespeare or linking it to the thing that is Stratford.

In the Semantic Web we refer to the things in the world as *resources;* a *resource* can be anything that someone might want to talk about. Shakespeare, Stratford, "the value of X," and "all the cows in Texas" are all examples of things someone might talk about and that can be resources in the Semantic Web. This is admittedly a pretty odd use of the word *resource,* but alternatives like *entity* or *thing,* which might be more accurate, have their own issues. In any case, *resource* is the word used in the

Semantic Web standards. In fact, the name of the base technology in the Semantic Web (RDF) uses this word in an essential way. *RDF* stands for *R*esource *D*escription *F*ramework.

In a web of information, anyone can contribute to our knowledge about a resource. It was this aspect of the current Web that allowed it to grow at such an unprecedented rate. To implement the Semantic Web, we need a model of data that allows information to be distributed over the Web.

DISTRIBUTING DATA ACROSS THE WEB

Data are most often represented in tabular form, in which each row represents some item we are describing, and each column represents some property of those items. The cells in the table are the particular values for those properties. Table 3.1 shows a sample of some data about works completed around the time of Shakespeare.

Let's consider a few different strategies for how these data could be distributed over the Web. In all of these strategies, some part of the data will be represented on one computer, while other parts will be represented on another. Figure 3.1 shows one strategy for distributing information over many machines. Each networked machine is responsible for maintaining the information about one or more complete rows from the table. Any query about an entity can be answered by the machine that stores its corresponding row. One machine is responsible for information about "Sonnet 78" and *Edward II,* whereas another is responsible for information about *As You Like It.*

This distribution solution provides considerable flexibility, since the machines can share the load of representing information about several individuals. But because it is a distributed representation of data, it requires some coordination between the servers. In particular, each server must share information about the columns. Does the second column on one server correspond to the same information as the second column on another server? This is not an insurmountable problem, and, in fact, it is a fundamental problem of data distribution. There must be some agreed-on coordination between the servers. In this example, the servers must be able to, in a global way, indicate which property each column corresponds to.

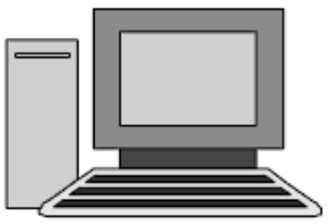
Table 3.1 Tabular Data about Elizabethan Literature and Music

ID	Title	Author	Medium	Year
1	*As You Like It*	Shakespeare	Play	1599
2	*Hamlet*	Shakespeare	Play	1604
3	*Othello*	Shakespeare	Play	1603
4	"Sonnet 78"	Shakespeare	Poem	1609
5	*Astrophil and Stella*	Sir Phillip Sidney	Poem	1590
6	*Edward II*	Christopher Marlowe	Play	1592
7	*Hero and Leander*	Christopher Marlowe	Poem	1593
8	*Greensleeves*	Henry VIII Rex	Song	1525

1	*As You Like It*	Shakespeare	Play	1599

Needs common schema—which column is which?

4	Sonnet 78	Shakespeare	Poem	1609
6	Edward II	Christopher Marlowe	Play	1592



7	Hero and Leander	Christopher Marlowe	Poem	1593
3	Othello	Shakespeare	Play	1603

FIGURE 3.1

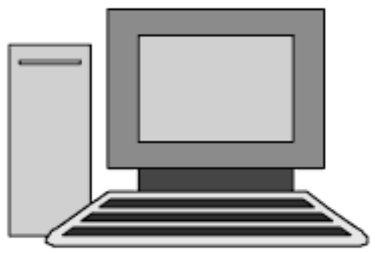
Distributing data across the Web, row by row.

Figure 3.2 shows another strategy, in which each server is responsible for one or more complete columns from the original table. In this example, one server is responsible for the publication dates and medium, and another server is responsible for titles. This solution is flexible in a different way from the solution of Figure 3.1. The solution in Figure 3.2 allows each machine to be responsible for one kind of information. If we are not interested in the dates of publication, we needn't consider information from that server. If we want to specify something new about the entities (say, how many pages the manuscript is), we can add a new server with that information without disrupting the others.

This solution is similar to the solution in Figure 3.1 in that it requires some coordination between the servers. In this case, the coordination has to do with the identities of the entities to be described. How do I know that row 3 on one server refers to the same entity as row 3 on another server? This solution requires a global identifier for the entities being described.

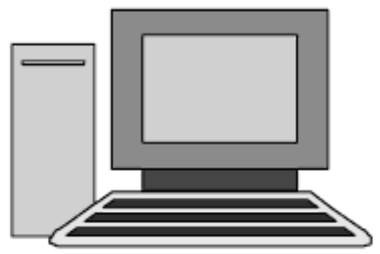
The strategy outlined in Figure 3.3 is a combination of the previous two strategies, in which information is neither distributed row by row nor column by column but instead is distributed cell by cell. Each machine is responsible for some number of cells in the table. This system combines the flexibility of both of the previous strategies. Two servers can share the description of a single entity (in the figure, the year and title of *Hamlet* are stored separately), and they can share the use of a particular property (in Figure 3.3, the Mediums of rows 6 and 7 are represented on different servers).

This flexibility is required if we want our data distribution system to really support the AAA slogan that "Anyone can say Anything about Any topic." If we take the AAA slogan seriously, any server needs to be able to make a statement about any entity (as is the case in Figure 3.2), but also any server needs to be able to specify any property of an entity (as is the case in Figure 3.1). The solution in Figure 3.3 has both of these benefits.



Year	Medium
1599	Play
1604	Play
1603	Play
1609	Poem
1590	Poem
1592	Play
1593	Poem
1525	Song

Needs to reference entities—which thing are we talking about?



Author
Shakespeare
Shakespeare
Shakespeare
Shakespeare
Sir Phillip Sidney
Christopher Marlowe
Christopher Marlowe
Henry VIII Rex

Title
As You Like It
Hamlet
Othello
"Sonnet 78"
Astrophil and Stella
Edward II
Hero and Leander
Greensleeves

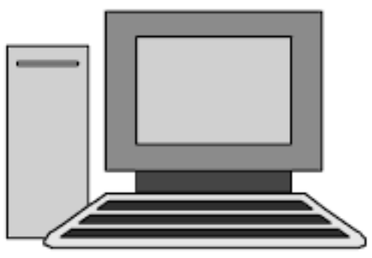
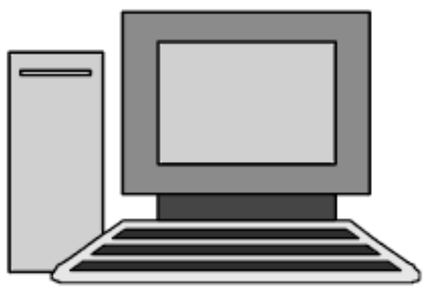


FIGURE 3.2

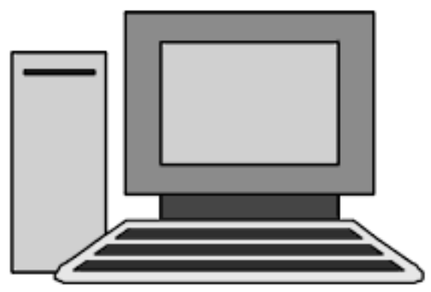
Distributing data across the Web, column by column.



	Medium
Row 7	Poem

	Title
Row 2	*Hamlet*

Needs to reference both schema and entities



	Author
Row 4	Shakespeare

	Year
Row 2	1604

	Medium
Row 6	Play

FIGURE 3.3

Distributing data across the Web, cell by cell.

Table 3.2 Sample Triples

Subject	Predicate	Object
Row 7	Medium	Poem
Row 2	Title	Hamlet
Row 2	Year	1604
Row 4	Author	Shakespeare
Row 6	Medium	Play

But this solution also combines the costs of the other two strategies. Not only do we now need a global reference for the column headings, but we also need a global reference for the rows. In fact, each cell has to be represented with three values: a global reference for the row, a global reference for the column, and the value in the cell itself. This third strategy is the strategy taken by RDF. We will see how RDF resolves the issue of global reference later in this chapter, but for now, we will focus on how a table cell is represented and managed in RDF.

Since a cell is represented with three values, the basic building block for RDF is called the *triple*. The identifier for the row is called the *subject* of the triple (following the notion from elementary grammar, since the subject is the thing that a statement is about). The identifier for the column is called the *predicate* of the triple (since columns specify properties of the entities in the rows). The value in the cell is called the *object* of the triple. Table 3.2 shows the triples in Figure 3.3 as subject, predicate, and object.

Triples become more interesting when more than one triple refers to the same entity, such as in Table 3.3. When more than one triple refers to the same thing, sometimes it is convenient to view the triples as a *directed graph* in which each triple is an edge from its subject to its object, with the predicate as the label on the edge, as shown in Figure 3.4. The graph visualization in Figure 3.4 expresses the same information presented in Table 3.3, but everything we know about Shakespeare (either as subject or object) is displayed at a single node.

Table 3.3 Sample Triples

Subject	Predicate	Object
Shakespeare	wrote	King Lear
Shakespeare	wrote	Macbeth
Anne Hathaway	married	Shakespeare
Shakespeare	livedIn	Stratford
Stratford	isIn	England
Macbeth	setIn	Scotland
England	partOf	UK
Scotland	partOf	UK

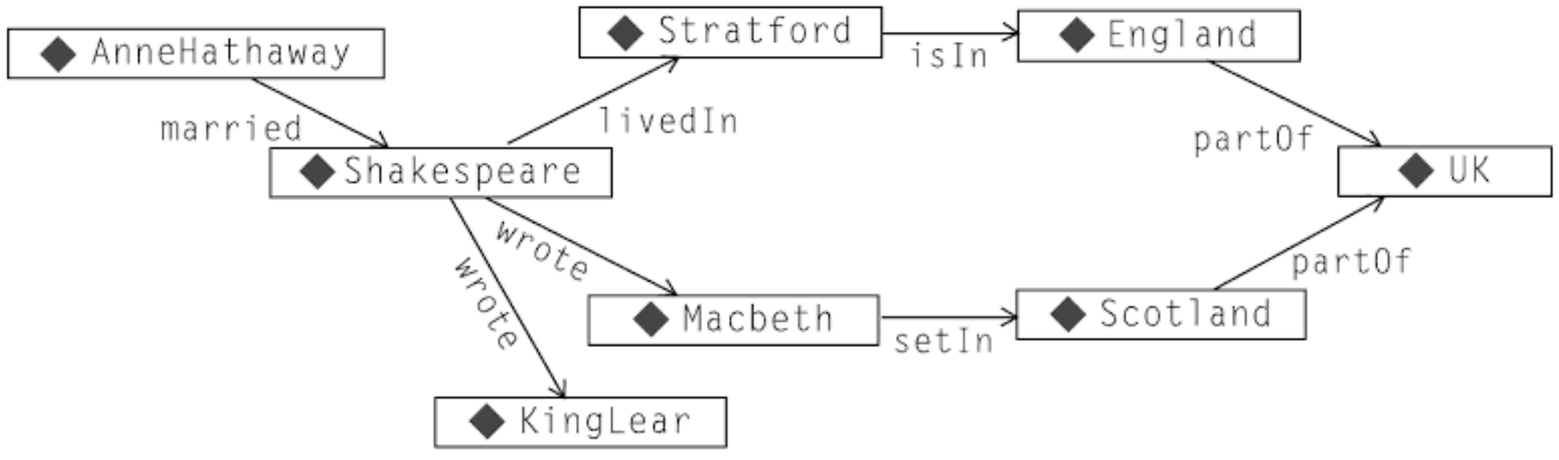


FIGURE 3.4

Graph display of triples from Table 3.3. Eight triples appear as eight labeled edges.

MERGING DATA FROM MULTIPLE SOURCES

We started off describing RDF as a way to distribute data over several sources. But when we want to use that data, we will need to merge those sources back together again. One value of the triples representation is the ease with which this kind of merger can be accomplished. Since information is represented simply as triples, merged information from two graphs is as simple as forming the graph of all of the triples from each individual graph, taken together. Let's see how this is accomplished in RDF.

Suppose that we had another source of information that was relevant to our example from Table 3.3—that is, a list of plays that Shakespeare wrote or a list of parts of the United Kingdom. These would be represented as triples as in Tables 3.4 and 3.5. Each of these can also be shown as a graph, just as in the original table, as shown in Figure 3.5.

What happens when we merge together the information from these three sources? We simply get the graph of all the triples that show up in Figures 3.4 and 3.5. Merging graphs like those in Figures 3.4 and 3.5 to create a combined graph like the one shown in Figure 3.6 is a straightforward process—but only when it is known which nodes in each of the source graphs match.

Table 3.4 Triples about Shakespeare's Plays

Subject	Predicate	Object
Shakespeare	Wrote	*As You Like It*
Shakespeare	Wrote	*Henry V*
Shakespeare	Wrote	*Love's Labour's Lost*
Shakespeare	Wrote	*Measure for Measure*
Shakespeare	Wrote	*Twelfth Night*
Shakespeare	Wrote	*The Winter's Tale*
Shakespeare	Wrote	*Hamlet*
Shakespeare	Wrote	*Othello*
		etc.

Table 3.5 Triples about the Parts of the United Kingdom

Subject	Predicate	Object
Scotland	part Of	The UK
England	part Of	The UK
Wales	part Of	The UK
Northern Ireland	part Of	The UK
Channel Islands	part Of	The UK
Isle of Man	part Of	The UK

NAMESPACES, URIS, AND IDENTITY

The essence of the merge comes down to answering the question "When is a node in one graph *the same node* as a node in another graph?" In RDF, this issue is resolved through the use of Uniform Resource Identifiers (*URIs*).

In the figures so far, we have labeled the nodes and edges in the graphs with simple names like *Shakespeare* or *Wales.* On the Semantic Web, this is not sufficient information to determine whether two nodes are really the same. Why not? Isn't there just one thing in the universe that everyone agrees refers to as *Shakespeare?* When referring to agreement on the Web, never say, "everyone." Somewhere, someone will refer not to the historical Shakespeare but to the title character of the feature film *Shakespeare in Love,* which bears very little resemblance to the historical figure. And "Shakespeare" is one of the more stable concepts to appear on the Web; consider the range of referents for a name like "Washington" or "Bordeaux." To merge graphs in a Semantic Web setting, we have to be more specific: In what sense do we mean the word *Shakespeare?*

RDF borrows its solution to this problem from foundational Web technology—in particular, the URI. The syntax and format of a URI are familiar even to casual users of the Web today because of the special, but typical, case of the URL—for example, *http://www.WorkingOntologist.org/Examples/Chapter3/Shakespeare#Shakespeare.* But the significance of the URI as a global identifier for a Web resource is often not appreciated. A URI provides a global identification for a resource that is common across the Web. If two agents on the Web want to refer to the same resource, recommended practice on the Web is for them to agree to a common URI for that resource. This is not a stipulation that is particular to the Semantic Web but to the Web in general; global naming leads to global network effects.

URIs and URLs look exactly the same, and, in fact, a URL is just a special case of the URI. Why does the Web have both of these ideas? Simplifying somewhat, the URI is an identifier with global (i.e., "World Wide" in the "World Wide Web" sense) scope. Any two Web applications in the world can refer to the same thing by referencing the same URI. But the syntax of the URI makes it possible to "dereference" it—that is, to use all the information in the URI (which specifies things like server name, protocol, port number, file name, etc.) to locate a file (or a location in a file) on the Web.[1] This

[1]We are primarily discussing files here, but a URI can refer to other resources. The Wikipedia article on URIs includes more than 50 different resource types that can be referenced by URIs—see *http://en.wikipedia.org/wiki/URI_scheme*.

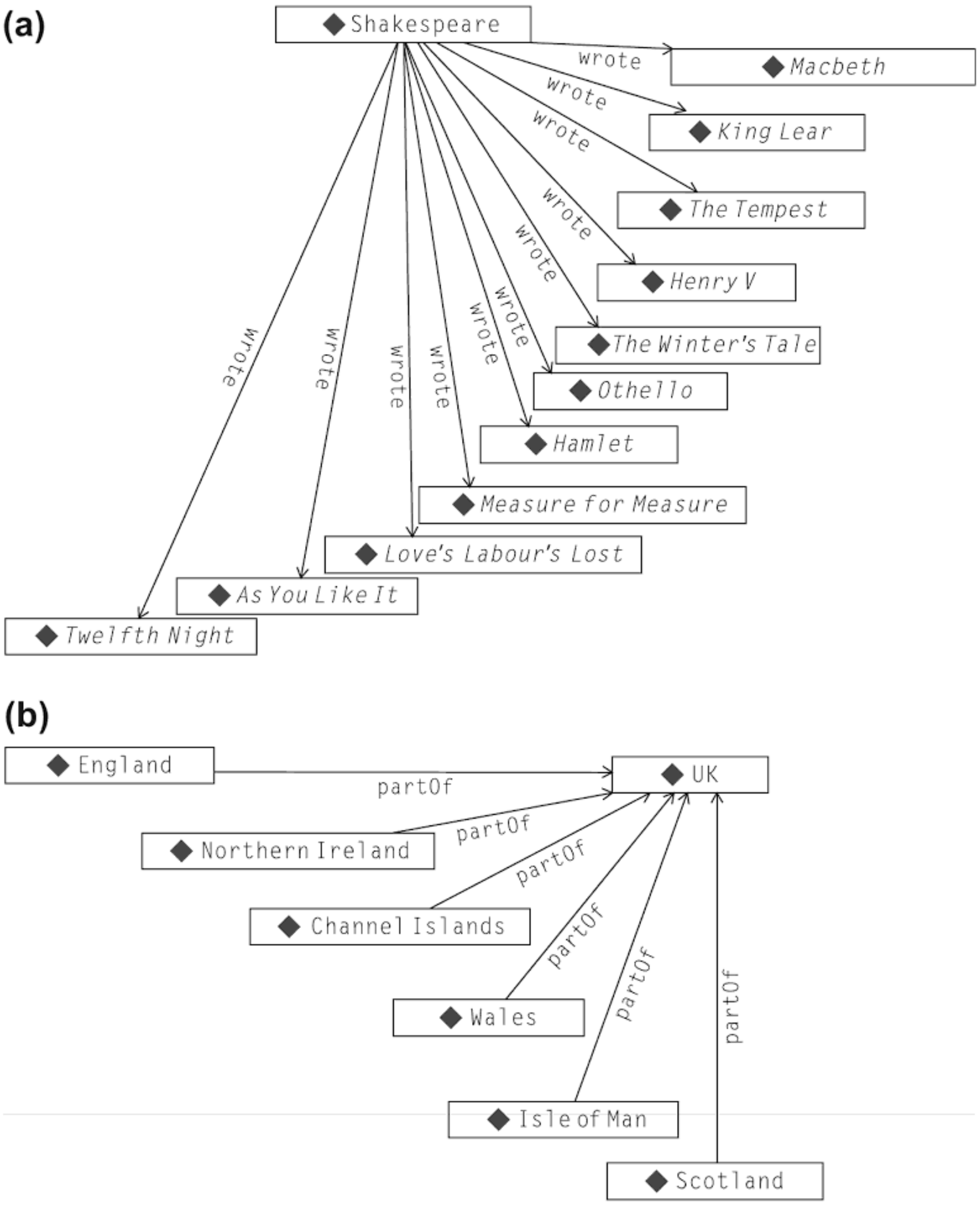


FIGURE 3.5

Graphic representation of triples describing (a) Shakespeare's plays and (b) parts of the United Kingdom.

dereferencing succeeds if all these parts work; the protocol locates the specified server running on the specified port and so on. When this is the case, we can say that the URI is not just a URI, but it also is a URL. From the point of view of modeling, the distinction is not important. But from the point of view of having a model on the Semantic Web, the fact that a URI can potentially be dereferenced allows the models to participate in a global Web infrastructure.

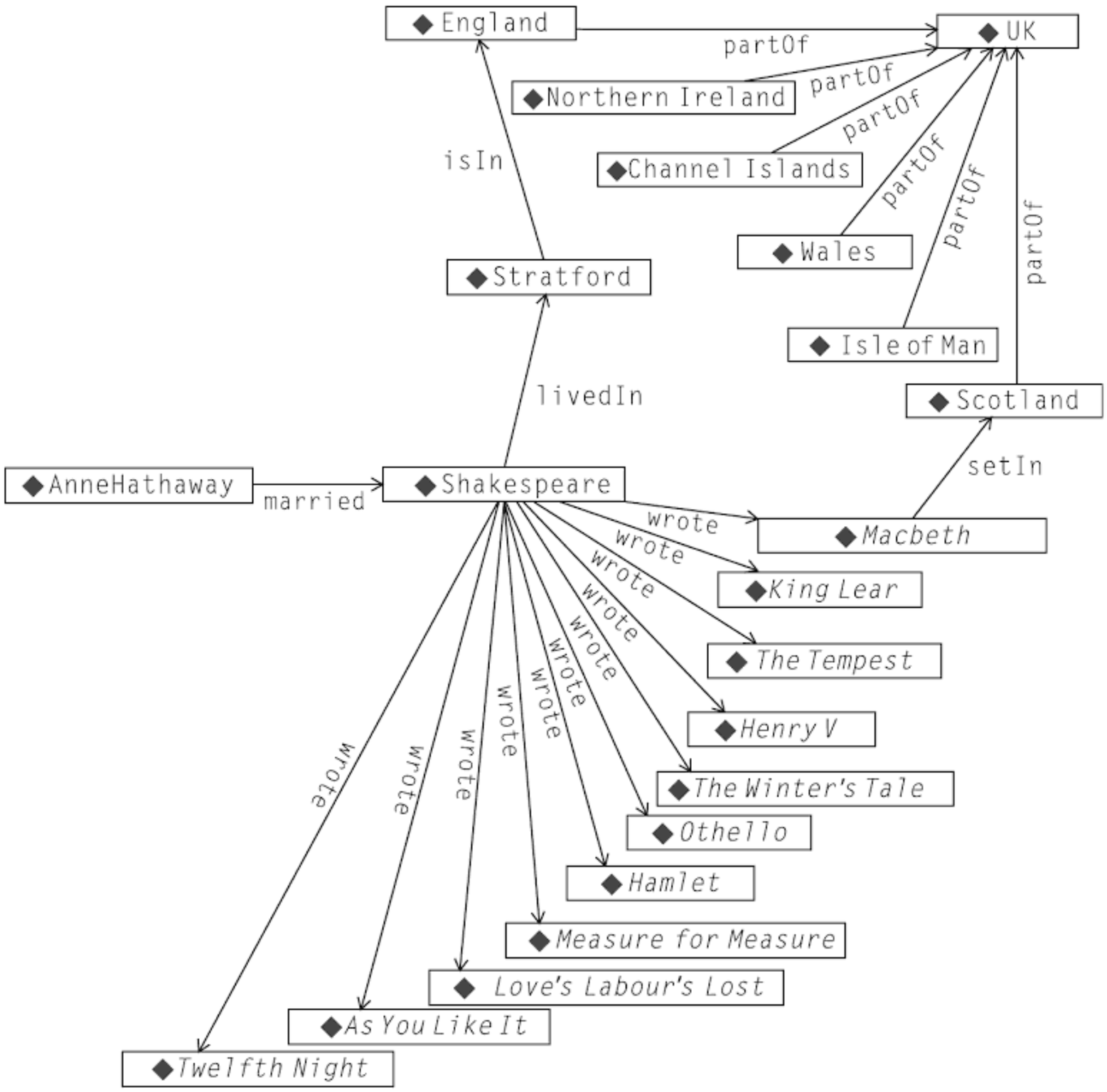


FIGURE 3.6

Combined graph of all triples about Shakespeare and the United Kingdom.

RDF applies the notion of the URI to resolve the identity problem in graph merging. The application is quite simple: A node from one graph is merged with a node from another graph—exactly, if they have the same URI. On the one hand, this may seem disingenuous, "solving" the problem of node identity by relying on another standard to solve it. On the other hand, since issues of identity appear in the Web in general and not just in the Semantic Web, it would be foolish not to use the same strategy to resolve the issue in both cases.

Expressing URIs in print

URIs work very well for expressing identity on the World Wide Web, but they are typically a bit of a pain to write out in detail when expressing models, especially in print. So for the examples in this book, we use a simplified version of a URI abbreviation scheme called *qnames*. In its simplest

form, a URI expressed as a qname has two parts: a namespace and an identifier, written with a colon between. So the qname representation for the identifier *England* in the namespace *geo* is simply `geo:England`. The RDF/XML standard includes elaborate rules that allow programmers to map namespaces to other URI representations (such as the familiar *http://* notation). For the examples in this book, we will use the simple qname form for all URIs. It is important, however, to note that qnames are *not* global identifiers on the Web; only fully qualified URIs (e.g., *http://www.WorkingOntologist.org/Examples/Chapter3/Shakespeare#Shakespeare*) are global Web names. Thus, any representation of a qname must, in principle, be accompanied by a declaration of the namespace correspondence.

It is customary on the Web in general and part of the XML specification to insist that URIs contain no embedded spaces. For example, an identifier "part of " is typically not used in the web. Instead, we follow the InterCap convention (sometimes called CamelCase), whereby names that are made up of multiple words are transformed into identifiers without spaces by capitalizing each word. Thus, "part of " becomes `partOf`, "Great Britain" becomes `GreatBritain`, "Measure for Measure" becomes `MeasureForMeasure`, and so on.

There is no limitation on the use of multiple namespaces in a single source of data, or even in a single triple. Selection of namespaces is entirely unrestricted as far as the data model and standards are concerned. It is common practice, however, to refer to related identifiers in a single namespace. For instance, all of the literary or geographical information from Table 3.4 or Table 3.5 would be placed into one namespace per table, with a suggestive name—say, *lit* or *geo*—respectively. Strictly speaking, these names correspond to fully qualified URIs—for example, *lit* stands for *http://www.WorkingOntologist.com/Examples/Chapter3/Shakespeare#*, and *geo* stands for *http://www.WorkingOntologist.com/Examples/Chapter3/geography#*.

For the purposes of explaining modeling on the Semantic Web, the detailed URIs behind the qnames are not important, so for the most part, we will omit these bindings from now on. In many examples, we will take this notion of abbreviation one step further; in the cases when we use a single namespace throughout one example, we will assume there is a *default* namespace declaration that allows us to refer to URIs simply with a symbolic name preceded by a colon (`:`), such as `:Shakespeare`, `:JamesDean`, `:Researcher`.

Table 3.6 Plays of Shakespeare with Qnames

Subject	Predicate	Object
lit:Shakespeare	lit:wrote	lit:AsYouLikeIt
lit:Shakespeare	lit:wrote	lit:HenryV
lit:Shakespeare	lit:wrote	lit:LovesLaboursLost
lit:Shakespeare	lit:wrote	lit:MeasureForMeasure
lit:Shakespeare	lit:wrote	lit:TwelfthNight
lit:Shakespeare	lit:wrote	lit:WintersTale
lit:Shakespeare	lit:wrote	lit:Hamlet
lit:Shakespeare	lit:wrote	lit:Othello
		etc.

Table 3.7 Geographical Information as Qnames

Subject	Predicate	Object
geo:Scotland	geo:partOf	geo:UK
geo:England	geo:partOf	geo:UK
geo:Wales	geo:partOf	geo:UK
geo:NorthernIreland	geo:partOf	geo:UK
geo:ChannelIslands	geo:partOf	geo:UK
geo:IsleOfMan	geo:partOf	geo:UK

Using qnames, our triple sets now look as shown in Tables 3.6 and 3.7. Compare Table 3.6 with Table 3.4, and compare Table 3.7 with Table 3.5. But it isn't always that simple; some triples will have to use identifiers with different namespaces, as in the example in Table 3.8, which was taken from Table 3.3.

In Table 3.8, we introduced a new namespace, *bio:,* without specifying the actual URI to which it corresponds. For this model to participate on the Web, this information must be filled in. But from the point of view of modeling, this detail is unimportant. For the rest of this book, we will assume that the prefixes of all qnames are defined, even if that definition has not been specified explicitly in print.

Standard namespaces

Using the URI as a standard for global identifiers allows for a worldwide reference for any symbol. This means that we can tell when any two people anywhere in the world are referring to the same thing.

This property of the URI provides a simple way for a standard organization (like the W3C) to specify the meaning of certain terms in the standard. As we will see in coming chapters, the W3C standards provide definitions for terms such as `type`, `subClassOf`, `Class`, `inverseOf`, and so forth. But these standards are intended to apply globally across the Semantic Web, so the standards

Table 3.8 Triples Referring to URIs with a Variety of Namespaces

Subject	Predicate	Object
lit:Shakespeare	lit:wrote	lit:KingLear
lit:Shakespeare	lit:wrote	lit:MacBeth
bio:AnneHathaway	bio:married	lit:Shakespeare
bio:AnneHathaway	bio:livedWith	lit:Shakespeare
lit:Shakespeare	bio:livedIn	geo:Stratford
geo:Stratford	geo:isIn	geo:England
geo:England	geo:partOf	geo:UK
geo:Scotland	geo:partOf	geo:UK

refer to these reserved words in the same way as they refer to any other resource on the Semantic Web, as URIs.

The W3C has defined a number of standard namespaces for use with Web technologies, including `xsd`: for XML schema definition; `xmlns`: for XML namespaces; and so on. The Semantic Web is handled in exactly the same way, with namespace definitions for the major layers of the Semantic Web. Following standard practice with the W3C, we will use qnames to refer to these terms, using the following definitions for the standard namespaces.

***rdf*:** Indicates identifiers used in RDF. The set of identifiers defined in the standard is quite small and is used to define types and properties in RDF. The global URI for the *rdf* namespace is *http://www.w3.org/1999/02/22-rdf-syntax-ns#.*

***rdfs*:** Indicates identifiers used for the RDF Schema language, RDFS. The scope and semantics of the symbols in this namespace are the topics of future chapters. The global URI for the *rdfs* namespace is *http://www.w3.org/2000/01/rdf-schema#.*

***owl*:** Indicates identifiers used for the Web Ontology Language, OWL. The scope and semantics of the symbols in this namespace are the topics of future chapters. The global URI for the *owl* namespace is *http://www.w3.org/2002/07/owl#.*

These URIs provide a good example of the interaction between a URI and a URL. For modeling purposes, any URI in one of these namespaces (e.g., *http://www.w3.org/2000/01/rdf-schema# subClassOf,* or `rdfs:subClassOf` for short) refers to a particular term that the W3C makes some statements about in the RDFS standard. But the term can also be dereferenced—that is, if we look at the server *www.w3.org,* there is a page at the location `2000/01/rdf-schema` with an entry about `subClassOf`, giving supplemental information about this resource. From the point of view of modeling, it is not necessary that it be possible to dereference this URI, but from the point of view of Web integration, it is critical that it is.

IDENTIFIERS IN THE RDF NAMESPACE

The RDF data model specifies the notion of triples and the idea of merging sets of triples as just shown. With the introduction of namespaces, RDF uses the infrastructure of the Web to represent agreements

Table 3.9 Using `rdf:type` to Describe Playwrights

Subject	Predicate	Object
lit:Shakespeare	rdf:type	lit:Playwright
lit:Ibsen	rdf:type	lit:Playwright
lit:Simon	rdf:type	lit:Playwright
lit:Miller	rdf:type	lit:Playwright
lit:Marlowe	rdf:type	lit:Playwright
lit:Wilder	rdf:type	lit:Playwright

Table 3.10 Defining Types of Names

Subject	Predicate	Object
lit:Playwright	rdf:type	bus:Profession
bus:Profession	rdf:type	hr:Compensation

on how to refer to a particular entity. The RDF standard itself takes advantage of the namespace infrastructure to define a small number of standard identifiers in a namespace defined in the standard, a namespace called *rdf*.

`rdf:type` is a property that provides an elementary typing system in RDF. For example, we can express the relationship between several playwrights using type information, as shown in Table 3.9. The subject of `rdf:type` in these triples can be any identifier, and the object is understood to be a type. There is no restriction on the usage of `rdf:type` with types; types can have types ad infinitum, as shown in Table 3.10.

When we read a triple out loud (or just to ourselves), it is understandably tempting to read it (in English, anyway) in subject/predicate/object order so that the first triple in Table 3.9 would read, "Shakespeare type Playwright." Unfortunately, this is pretty fractured syntax no matter how you inflect it. It would be better to have something like "Shakespeare has type Playwright" or maybe "The type of Shakespeare is Playwright."

This issue really has to do with the choice of name for the `rdf:type` resource; if it had been called `rdf:isInstanceOf` instead, it would have been much easier to read out loud in English. But since we never have control over how other entities (in this case, the W3C) chose their names, we don't have the luxury of changing these names. When we read out loud, we just have to take some liberties in adding in connecting words. So this triple can be pronounced, "Shakespeare [has] type Playwright," adding in the "has" (or sometimes, the word "is" works better) to make the sentence into somewhat correct English.

`rdf:Property` is an identifier that is used as a type in RDF to indicate when another identifier is to be used as a predicate rather than as a subject or an object. We can declare all the identifiers we have used as predicates so far in this chapter as shown in Table 3.11.

Table 3.11 `rdf:Property` Assertions for Tables 3.5 to 3.8

Subject	Predicate	Object
lit:wrote	rdf:type	rdf:Property
geo:partOf	rdf:type	rdf:Property
bio:married	rdf:type	rdf:Property
bio:livedIn	rdf:type	rdf:Property
bio:livedWith	rdf:type	rdf:Property
geo:isIn	rdf:type	rdf:Property

CHALLENGE: RDF AND TABULAR DATA

We began this chapter by motivating RDF as a way to distribute data over the Web—in particular, tabular data. Now that we have all of the detailed mechanisms of RDF (including namespaces and triples) in place, we can revisit tabular data and show how to represent it consistently in RDF.

CHALLENGE 1

Given a table from a relational database, describing products, suppliers, and stocking information about the products (see Table 3.12), produce an RDF graph that reflects the content of Table 3.12 in such a way that the information intent is preserved but the data are now amenable for RDF operations like merging and RDF query.

Solution

Each row in the table describes a single entity, all of the same type. That type is given by the name of the table itself, *Product*. We know certain information about each of these items, based on the columns in the table itself, such as the model number, the division, and so on. We want to represent these data in RDF.

Since each row represents a distinct entity, each row will have a distinct URI. Fortunately, the need for unique identifiers is just as present in the database as it is in the Semantic Web, so there is a (locally) unique identifier available—namely, the primary table key, in this case the column called *ID*. For the Semantic Web, we need a globally unique identifier. The simplest way to form such an identifier is by having a single URI for the database itself (perhaps even a URL if the database is on the Web). Use that URI as the namespace for all the identifiers in the database. Since this is a database for a manufacturing company, let's call that namespace `mfg:`.

Table 3.12 Sample Tabular Data for Triples

Product						
ID	**Model Number**	**Division**	**Product Line**	**Manufacture Location**	**SKU**	**Available**
1	ZX-3	Manufacturing support	Paper machine	Sacramento	FB3524	23
2	ZX-3P	Manufacturing support	Paper machine	Sacramento	KD5243	4
3	ZX-3S	Manufacturing support	Paper machine	Sacramento	IL4028	34
4	B-1430	Control engineering	Feedback line	Elizabeth	KS4520	23
5	B-1430X	Control engineering	Feedback line	Elizabeth	CL5934	14
6	B-1431	Control engineering	Active sensor	Seoul	KK3945	0
7	DBB-12	Accessories	Monitor	Hong Kong	ND5520	100
8	SP-1234	Safety	Safety valve	Cleveland	HI4554	4
9	SPX-1234	Safety	Safety valve	Cleveland	OP5333	14

Table 3.14 Triples Representing Type of Information from Table 3.12

Subject	Predicate	Object
mfg:Product1	rdf:type	mfg:Product
mfg:Product2	rdf:type	mfg:Product
mfg:Product3	rdf:type	mfg:Product
mfg:Product4	rdf:type	mfg:Product
mfg:Product5	rdf:type	mfg:Product
mfg:Product6	rdf:type	mfg:Product
mfg:Product7	rdf:type	mfg:Product
mfg:Product8	rdf:type	mfg:Product
mfg:Product9	rdf:type	mfg:Product

HIGHER-ORDER RELATIONSHIPS

It is not unusual for someone who is building a model in RDF for the first time to feel a bit limited by the simple subject/predicate/object form of the RDF triple. They don't want to just say that *Shakespeare wrote Hamlet,* but they want to qualify this statement and say that *Shakespeare wrote Hamlet in 1604* or that *Wikipedia states that Shakespeare wrote Hamlet in 1604*. In general, these are cases in which it is, or at least seems, desirable to make a statement about another statement. This process is called *reification*. Reification is not a problem specific to Semantic Web modeling; the same issue arises in other data modeling contexts like relational databases and object systems. In fact, one approach to reification in the Semantic Web is to simply borrow the standard solution that is commonly used in relational database schemas, using the conventional mapping from relational tables to RDF given in the preceding challenge. In a relational database table, it is possible to simply create a table with more columns to add additional information about a triple. So the statement *Shakespeare wrote Hamlet* is expressed (as in Table 3.1) in a single row of a table, where there is a column for the author of a work and another column for its title. Any further information about this event is done with another column (again, just as in Table 3.1). When this is converted to RDF according to the example in the Challenge, the row is represented by a number of triples, one triple per column in the database. The subject of all of these triples is the same: a single resource that corresponds to the row in the table.

An example of this can be seen in Table 3.13, where several triples have the same subject and one triple apiece for each column in the table. This approach to reification has a strong pedigree in relational modeling, and it has worked well for a wide range of modeling applications. It can be applied in RDF even when the data have not been imported from tabular form. That is, the statement *Shakespeare wrote Hamlet in 1601* (disagreeing with the statement in Table 3.2) can be expressed with these three triples:

```
bio:n1 bio:author lit:Shakespeare.
bio:n1 bio:title "Hamlet".
bio:n1 bio:publicationDate 1601.
```

Let's take the example of *Wikipedia says Shakespeare wrote Hamlet.* Using the RDF standard, we can refer to a triple as follows:

```
q:n1 rdf:subject lit:Shakespeare;
     rdf:predicate lit:wrote;
     rdf:object lit:Hamlet.
```

Then we can express the relation of Wikipedia to this statement as follows:

```
web:Wikipedia m:says q:n1.
```

Notice that just because we have asserted the reification triples about q:n1, it is not necessarily the case that we have also asserted the triple itself:

```
lit:Shakespeare lit:wrote lit:Hamlet.
```

This is as it should be; after all, if an application does not trust information from Wikipedia, then it should not behave as though that triple has been asserted. An application that does trust Wikipedia will want to behave as though it had.

ALTERNATIVES FOR SERIALIZATION

So far, we have expressed RDF triples in subject/predicate/object tabular form or as graphs of boxes and arrows. Although these are simple and apparent forms to display triples, they aren't always the most compact forms, or even the most human-friendly form, to see the relations between entities.

The issue of representing RDF in text doesn't only arise in books and documents about RDF; it also arises when we want to publish data in RDF on the Web. In response to this need, there are multiple ways of expressing RDF in textual form.

N-Triples

The simplest form is called *N-Triples* and corresponds most directly to the raw RDF triples. It refers to resources using their fully unabbreviated URIs. Each URI is written between angle brackets (< and >). Three resources are expressed in subject/predicate/object order, followed by a period (.). For example, if the namespace `mfg` corresponds to *http://www.WorkingOntologist.org/Examples/Chapter3 Manufacture#*, then the first triple from Table 3.14 is written in N-Triples as follows:

```
<http://www.WorkingOntologist.org/Examples/Chapter3Manufacture#

http://www.WorkingOntologist.org/Examples/Chapter3/Manufacture#Product1

http://www.WorkingOntologist.org/Examples/Chapter3/Manufacture#Product
```

It is difficult to print N-Triples on a page in a book—the serialization does not allow for new lines within a triple (as we had to do here, to fit it in the page). An actual ntriple file has the whole triple on a single line.

Turtle

In this book, we use a more compact serialization of RDF called *Turtle*. Turtle combines the apparent display of triples from N-Triples with the terseness of qnames. We will introduce Turtle in this section and describe just the subset required for the current examples. We will describe more of the language as needed for later examples. For a full description of Turtle, see the W3C Turtle team submission.[1]

Since Turtle uses qnames, there must be a binding between the (local) qnames and the (global) URIs. Hence, Turtle begins with a preamble in which these bindings are defined; for example, we can define the qnames needed in the Challenge example with the following preamble:

```
@prefix mfg:
<http://www.WorkingOntologist.com/Examples/Chapter3/Manufac
     turing#>
@prefix rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

Once the local qnames have been defined, Turtle provides a very simple way to express a triple by listing three resources, using qname abbreviations, in subject/predicate/object order, followed by a period, such as the following:

```
mfg:Product1 rdf:type mfg:Product .
```

The final period can come directly after the resource for the object, but we often put a space in front of it, to make it stand out visually. This space is optional.

It is quite common (especially after importing tabular data) to have several triples that share a common subject. Turtle provides for a compact representation of such data. It begins with the first triple in subject/predicate/object order, as before; but instead of terminating with a period, it uses a semicolon (;) to indicate that another triple with the same subject follows. For that triple, only the predicate and object need to be specified (since it is the same subject from before). The information in Tables 3.13 and 3.14 about `Product1` and `Product2` appears in Turtle as follows:

```
mfg:Product1 rdf:type mfg:Product;
     mfg:Product_Division "Manufacturing support";
     mfg:Product_ID "1";
     mfg:Product_Manufacture_Location "Sacramento";
     mfg:Product_ModelNo "ZX-3";
     mfg:Product_Product_Line "Paper Machine";
     mfg:Product_SKU "FB3524";
     mfg:Product_Available "23" .
mfg:Product2 rdf:type mfg:Product;
     mfg:Product_Division "Manufacturing support";
     mfg:Product_ID "2";
     mfg:Product_Manufacture_Location "Sacramento";
     mfg:Product_ModelNo "ZX-3P";
     mfg:Product_Product_Line "Paper Machine";
     mfg:Product_SKU "KD5243";
     mfg:Product_Available "4" .
```

[1] http://www.w3.org/TeamSubmission/turtle/

When there are several triples that share both subject and predicate, Turtle provides a compact way to express this as well so that neither the subject nor the predicate needs to be repeated. Turtle uses a comma (,) to separate the objects. So the fact that Shakespeare had three children named Susanna, Judith, and Hamnet can be expressed as follows:

```
lit:Shakespeare b:hasChild b:Susanna, b:Judith, b:Hamnet.
```

There are actually three triples represented here—namely:

```
lit:Shakespeare b:hasChild b:Susanna.
lit:Shakespeare b:hasChild b:Judith.
lit:Shakespeare b:hasChild b:Hamnet.
```

Turtle provides some abbreviations to improve terseness and readability; in this book, we use just a few of these. One of the most widely used abbreviations is to use the word *a* to mean `rdf:type`. The motivation for this is that in common speech, we are likely to say, "`Product1` is *a* Product" or "Shakespeare is *a* playwright" for the triples,

```
mfg:Product1 rdf:type mfg:Product.
lit:Shakespeare rdf:type lit:Playwright.
```

respectively. Thus we will usually write instead:

```
mfg:Product1 a mfg:Product.
lit:Shakespeare a lit:Playwright.
```

RDF/XML

While Turtle is convenient for human consumption and is more compact for the printed page, many Web infrastructures are accustomed to representing information in HTML or, more generally, XML. For this reason, the W3C has recommended the use of an XML serialization of RDF called RDF/XML. The information about `Product1` and `Product2` just shown looks as follows in RDF/XML. In this example, the subjects (`Product1` and `Product2`) are referenced using the XML attribute `rdf:about`; the triples with each of these as subjects appear as subelements within these definitions. The complete details of the RDF/XML syntax are beyond the scope of this discussion and can be found in *http://www.w3.org/TR/rdf-syntax-grammar/*.

```
<rdf:RDF
xmlns:mfg="http://www.WorkingOntologist.com/Examples/Chapter3/
Manufacturing#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
     ns#">
     <mfg:Product
```

But if we don't want to have an identifier for the mistress, how can we proceed? RDF allows for a "blank node," or *bnode* for short, for such a situation. If we were to indicate a bnode with a ?, the triples would look as follows:

```
? rdf:type bio:Woman;
     bio:livedIn geo:England.
lit:Sonnet78 lit:hasInspiration ?.
```

The use of the bnode in RDF can essentially be interpreted as a logical statement, "there exists." That is, in these statements we assert "there exists a woman, who lived in England, who was the inspiration for 'Sonnet78.'"

But this notation (which does *not* constitute a valid Turtle expression) has a problem: If there is more than one blank node, how do we know which "?" references which node? For this reason, Turtle instead includes a compact and unambiguous notation for describing blank nodes. A blank node is indicated by putting all the triples of which it is a subject between square brackets ([and]), so:

```
[ rdf:type bio:Woman;
  bio:livedIn geo:England ]
```

It is customary, though not required, to leave blank space after the opening bracket to indicate that we are acting *as if* there were a subject for these triples, even though none is specified.

We can refer to this blank node in other triples by including the entire bracketed sequence in place of the blank node. Furthermore, the abbreviation of "a" for `rdf:type` is particularly useful in this context. Thus, our entire statement about the mistress who inspired "Sonnet 78" looks as follows in Turtle:

```
lit:Sonnet78 lit:hasInspiration [a :Woman;
                                 bio:livedIn geo:England].
```

This expression of RDF can be read almost directly as plain English: that is, "Sonnet78 has [as] inspiration a `Woman` [who] lived in `England`." The identity of the woman is indeterminate. The use of the bracket notation for blank nodes will become particularly important when we come to describe OWL, the Web Ontology Language, since it makes very particular use of bnodes.

Ordered information in RDF

The children of Shakespeare appear in a certain order on the printed page, but from the point of view of RDF, they are in no order at all; there are just three triples, one describing the relationship between Shakespeare and each of his children. What if we do want to specify an ordering. How would we do it in RDF?

RDF provides a facility for ordering elements in a list format. An ordered list can be expressed quite easily in Turtle as follows:

```
lit:Shakespeare b:hasChild (b:Susanna b:Judith b:Hamnet).
```

This translates into the following triples, where `_:a`, `_:b`, and `_:c` are bnodes:

```
lit:Shakespeare b:hasChild _:a.
_:a rdf:first b:Susanna.
_:a rdf:rest _:b.
_:b rdf:first b:Judith.
_:b rdf:rest _:c.
_:c rdf:rest rdf:nil.
_:c rdf:first b:Hamnet.
```

This rendition preserves the ordering of the objects but at a cost of considerable complexity of representation. Fortunately, the Turtle representation is quite compact, so it is not usually necessary to remember the details of the RDF triples behind it.

SUMMARY

RDF is, first and foremost, a system for modeling data. It gives up in compactness what it gains in flexibility. Every relationship between any two data elements is explicitly represented, allowing for a very simple model of merging data. There is no need to arrange the columns of tables so that they "match up" or to worry about data "missing" from a particular column. A relationship (expressed in a familiar form of subject/predicate/object) is either present or it is not. Merging data is thus reduced to a simple matter of considering all such statements from all sources, together in a single place.

The only challenge that remains in such a system is the challenge of *identity.* How do we have a global notation for the identity of any entity? Fortunately, this problem is not unique to the RDF data model. The infrastructure of the Web itself has the same issue and has a standard solution: the URI. RDF borrows this solution.

Since RDF is a Web language, a fundamental consideration is the distribution of information from multiple sources, across the Web. On the Web, the AAA slogan holds: Anyone can say Anything about Any topic. RDF supports this slogan by allowing any data source to refer to resources in any namespace. Even a single triple can refer to resources in multiple namespaces.

As a data model, RDF provides a clear specification of what has to happen to merge information from multiple sources. It does not provide algorithms or technology to implement those processes. These technologies are the topic of the next chapter.

Fundamental concepts

The following fundamental concepts were introduced in this chapter.

RDF (Resource Description Framework)—This distributes data on the Web.
Triple—The fundamental data structure of RDF. A triple is made up of a subject, predicate, and object.
Graph—A nodes-and-links structural view of RDF data.
Merging—The process of treating two graphs as if they were one.
URI (Uniform Resource Indicator)—A generalization of the URL (Uniform Resource Locator), which is the global name on the Web.

CHAPTER

4 Semantic Web application architecture

CHAPTER OUTLINE

So far, we have seen how RDF can represent data in a distributed way across the Web. As such, it forms the basis for the Semantic Web, a web of data in which Anyone can say Anything about Any topic. The focus of this book is modeling on the Semantic Web: describing and defining distributed data in such a way that the data can be brought back together in a useful and meaningful way. In a book about only modeling, one could say that there is no room for a discussion of system architecture—the components of a computer system that can actually use these models in useful applications. But this book is for the working ontologist who builds models so that they can be used. But used for what? These models are used to build some application that takes advantage of information distributed over the Web. In short, to put the Semantic Web to work, we need to describe, at least at a high level, the structure of a Semantic Web application—in particular, the components that comprise it, the kinds of inputs it gets (and from where), how it takes advantage of RDF, and why this is different from other application architectures.

Many of the components of a Semantic Web application are provided both as supported products by companies specializing in Semantic Web technology and as free software under a variety of licenses. New software is being developed both by research groups as well as product companies on an ongoing basis. We do not describe any particular tools in this chapter, but rather we discuss the types of components that make up a Semantic Web deployment and how they fit together.

RDF Parser/Serializer We have already seen a number of serializations of RDF, including the W3C standard serialization in XML. An RDF parser reads text in one (or more) of these formats and interprets it as triples in the RDF data model. An RDF serializer does the

reverse; it takes a set of triples and creates a file that expresses that content in one of the serialization forms.

RDF Store We have seen how RDF distributes data in the form of triples. An RDF store (sometimes called a triple store) is a database that is tuned for storing and retrieving data in the form of triples. In addition to the familiar functions of any database, an RDF store has the additional ability to merge information from multiple data sources, as defined by the RDF standard.

RDF Query Engine Closely related to the RDF store is the RDF query engine. The query engine provides the capability to retrieve information from an RDF store according to structured queries.

Application An application has some work that it performs with the data it processes: analysis, user interaction, archiving, and so forth. These capabilities are accomplished using some programming language that accesses the RDF store via queries (processed with the RDF query engine).

Most of these components have corresponding components in a familiar relational data-backed application. The relational database itself corresponds to the RDF store in that it stores the data. The database includes a query language with a corresponding query engine for accessing this data. In both cases, the application itself is written using a general-purpose programming language that makes queries and processes their results. The parser/serializer has no direct counterpart in a relational data-backed system, at least as far as standards go. There is no standard serialization of a relational database that will allow it to be imported into a competing relational database system without a change of semantics. (This is a key advantage of RDF stores over traditional data stores.)

In the following sections, we examine each of these capabilities in detail. Since new products in each of these categories are being developed on an ongoing basis, we describe them generically and do not refer to specific products.

RDF PARSER/SERIALIZER

How does an RDF-based system get started? Where do the triples come from? There are a number of possible answers for this, but the simplest one is to find them directly on the Web.

Google can find millions of files with the extension `.rdf`. Any of these could be a source of data for an RDF application. But these files are useless unless we have a program that can read them. That program is an RDF parser. RDF parsers take as their input a file in some RDF format. Most parsers support the standard RDF/XML format, which is compatible with the more widespread XML standard. An RDF parser takes such a file as input and converts it into an internal representation of the triples that are expressed in that file. At this point, the triples are stored in the triple store and are available for all the operations of that store.

The triples at this point could also be serialized back out, either in the same text form or in another text form. This is done using the reverse operation of the parser: the serializer. It is possible to take a "round-trip" with triples using a parser and serializer; if you serialize a set of triples, then you parse the resulting string with a corresponding parser (e.g., a Turtle parser for a Turtle serialization), and the result is the same set of triples that the process began with. Notice that this is not necessarily true if you start with a text file that represents some triples. Even in a single format, there can be many distinct files that represent the same set of triples. Thus, it is not, in general, possible to read in an RDF file,

export it again, and be certain that the resulting file will be identical (character by character) to the input file.

Other data sources

Parsers and serializers based on the standard representations of RDF are useful for the systematic processing and archiving of data in RDF. While there are considerable data available in these formats, even more data are not already available in RDF. Fortunately, for many common data formats (e.g., tabular data), it is quite easy to convert these formats into RDF triples.

We already saw how tabular data can be mapped into triples in a natural way. This approach can be applied to relational databases or spreadsheets. Tools to perform a conversion based on this mapping, though not strictly speaking parsers, play the same role as a parser in a semantic solution: They connect the triple store with sources of information in the form of triples. Most RDF systems include a table input converter of some sort. Some tools specifically target relational databases, including appropriate treatment of foreign key references, whereas others work more directly with spreadsheet tables. Tools of this sort are called *converters,* since they typically convert information from some form into RDF and often into a standard form of RDF like Turtle. This allows them to be used with any other RDF. Another rich source of data for the Semantic Web can be found in existing web pages—that is, in HTML pages. Such pages often include structured information, like contact information, descriptions of events, product descriptions, publications, and so on. This information can be combined in novel ways on the Semantic Web once it is available in RDF.

A related approach to encoding information in web pages is a trend that goes by the name of *microformats.* The idea of a microformat is that some web page authors might be willing to embed structured information in their web pages. To enable them to do this, a standard vocabulary (usually embedded in HTML as special tag attributes that have no impact on how a browser displays a page) is developed for commonly used items on a web page. Some of the first microformats were for business cards (including, in the controlled vocabulary, names, positions, companies, and phone numbers) and events (including location, start time, and end time). One limitation of microformats is the need to specify a controlled vocabulary and provide a parser that can process that vocabulary. Wouldn't it be better if, instead, someone (like the W3C) would simply specify a single syntax for marking up HTML pages with RDF data? Then there would be a single processing script for all microformats.

The W3C has proposed just such a format called RDFa. The idea behind RDFa is quite simple: Use the attribute tags in HTML to embed information that can be parsed into RDF. Just like microformats, RDFa has no effect on how a browser displays a page. A number of search engines (Google and Yahoo!) and retailers (BestBuy, Overstock.com) have begun to adopt RDFa to provide machine processable Semantic Web data. Facebook has adopted a variant of RDFa as part of the Open Graph Protocol—a network of information available in Facebook.

RDFa provides two advantages for sharing data on the Web. First, from the point of view of data consumers, it is easier to harvest the RDF data from pages that were marked up with structured data extraction in mind, than from sources that were developed without this intention. But, more important, from the point of view of the content author, it allows them to express the intended meaning of a web page inside the web page itself. This ensures that the RDF data in the document matches the intended meaning of the document itself. This really is the spirit of the word *semantic* in the Semantic

problems, including the replication of information in the first column and the difficulty of building indices around string values like URIs.

It is not the purpose of this discussion to go into details of the possible optimizations of the RDF store. These details are the topic of the particular (sometimes proprietary) solutions provided by a vendor of an off-the-shelf RDF store. In particular, the issue of building indices that work on URIs can be solved with a number of well-understood data organization algorithms. Serious providers of RDF stores differentiate their offerings based on the scalability and efficiency of these indexing solutions.

RDF data standards and interoperability of RDF stores

RDF stores bear considerable similarity to relational stores, especially in terms of how the quality of a store is evaluated. A notable distinction of RDF stores results from the standardization of the RDF data model and RDF/XML serialization syntax. Several competing vendors of relational data stores dominate the market today, and they have for several decades. While each of these products is based on the same basic idea of the relational algebra for data representation, it is a difficult process to transfer a whole database from one system to another. That is, there is no standard serialization language with which one can completely describe a relational database in such a way that it can be automatically imported into a competitor's system. Such a task is possible, but it typically requires a database programmer to track down the particulars of the source database to ensure that they are represented faithfully in the target system.

The standardization effort for RDF makes the situation very different when it comes to RDF stores. Just as for relational stores, there are several competing vendors and projects. In stark contrast to the situation for relational databases, the underlying RDF data model is shared by all of these products, and, even more specifically, all of them can import and export their data sets in any of the standard formats (RDF/XML or Turtle). This makes it a routine task to transfer an RDF data set—or several RDF data sets—from one RDF store to another. This feature, which is a result of an early and aggressive standardization process, makes it much easier to begin with one RDF store, secure in the knowledge that the system can be migrated to another as the need arises. It also simplifies the issue of federating data that are housed in multiple RDF stores, possibly coming from different vendor sources.

RDF query engines

An RDF store is typically accessed using a query language. In this sense, an RDF store is similar to a relational database or an XML store. Not surprisingly, in the early days of RDF, a number of different query languages were available, each supported by some RDF-based product or open-source project. From the common features of these query languages, the W3C has undertaken the process of standardizing an RDF query language called SPARQL. We cover the details of the SPARQL query language in the next chapter.

An RDF query engine is intimately tied to the RDF store. To solve a query, the engine relies on the indices and internal representations of the RDF store: the more finely tuned the store is to the query engine, the better its performance. For large-scale applications, it is preferable to have an RDF store and query engine that retain their performance even in the face of very large data sets. For smaller applications, other features (e.g., cost, ease of installation, platform, open-source status, and built-in integration with other enterprise systems) may dominate.

The SPARQL query language includes a protocol for communicating queries and results so that a query engine can act as a web service. This provides another source of data for the semantic web—the so-called *SPARQL endpoints* provide access to large amounts of structured RDF data. It is even possible to provide SPARQL access to databases that are not triple stores, effectively translating SPARQL queries into the query language of the underlying store. The W3C has recently begun the process to standardize a translation from SPARQL to SQL for relational stores.

Comparison to relational queries

In many ways, an RDF query engine is very similar to the query engine in a relational data store: It provides a standard interface to the data and defines a formalism by which data are viewed. A relational query language is based on the relational algebra of joins and foreign key references. RDF query languages look more like statements in predicate calculus. Unification variables are used to express constraints between the patterns.

A relational query describes a new data table that is formed by combining two or more source tables. An RDF query (whether in SPARQL or another RDF query language) can describe a new graph that is formed by describing a subset of a source RDF graph. That graph, in turn, may be the result of having merged together several other graphs. The inherently recursive nature of graphs simplifies a number of detailed issues that arise in table-based queries. For instance, an RDF query language like SPARQL has little need for a subquery construct; in many cases, the same effect can be achieved with a single query. Similarly, there is nothing corresponding to the special case of an SQL "self-join" in SPARQL.

In the special case in which an RDF store is implemented as a single table in a relational database, any graph pattern match in such a scenario will constitute a self-join on that table. Some end-developers choose to work this way in a familiar SQL environment. Oracle takes another approach to making RDF queries accessible to SQL programmers by providing its own SPARQL extension to its version of SQL, optimized for graph queries. Their SPARQL engine is smoothly integrated with the table/join structure of their SQL scripting language.

APPLICATION CODE

Database applications include more than just a database and query engine; they also include some application code, in an application environment, that performs some analysis on or displays some information from the database. The only access the application has to the database is through the query interface, as shown in Figure 4.1.

An RDF application has a similar architecture, but it includes the RDF parser and serializer, converters, the RDF merge functionality, and the RDF query engine. These capabilities interact with the application itself and the RDF store as shown in Figure 4.2.

The application itself can take any of several forms. Most commonly, it is written in a conventional programming language (Java, C, Python, and Perl are popular options). In this case, the RDF capabilities are provided as API bindings for that language. It is also common for an RDF store to provide a scripting language as part of the query system, which gives programmatic access to these capabilities in a way that is not unlike how advanced dialects of SQL provide scripting capabilities for relational database applications.

Depending on the volatility of the data, some of this process may even happen offline (e.g., locations of subway stations in New York, entries in the Sears catalog, analyses of common chemical structures, etc., don't change very rapidly; these sorts of data could be imported into RDF entirely outside of an application context), whereas other data (like calendar data of team members, transactional sales data, experimental results) will have to be updated on a regular basis. Some data can remain in the RDF store itself (private information about this team, order information, patented chemical formulas); other data could be published in RDF form for other applications to use (train timetables, catalog specials, FDA findings about certain chemicals).

Once all the required data sources have been converted, fetched, or parsed, the application uses the merge functionality of the RDF store to produce a single, federated graph of all the merged data. It is this federated graph that the application will use for all further queries. There is no need for the queries themselves to be aware of the federation strategy or schedule; the federation has already taken place when the RDF merge was performed.

From this point onward, the application behaves very like any other database application. A web page to display the appointments of any member of a team will include a query for that information. Even if the appointments came from different sources and the information about team membership from still another source, the query is made against the federated information graph.

RDF-backed web portals

When the front end of an application is a web server, the architecture (shown in Figure 4.1) is well known for a database-backed web portal. The pages are generated using any of a number of technologies (e.g., CGI, ASP, JSP, ZOPE) that allow web pages to be constructed from the results of queries against a database. In the earliest days of the web, web pages were typically stored statically as files in a file system. The move to database-backed portals was made to allow web sites to reflect the complex interrelated structure of data as it appears in a relational database.

The system architecture outlined in Figure 4.2 can be used the same way to implement a Semantic Web portal. The RDF store plays the same role that the database plays in database-backed portals. It is important to note that because of the separation between the presentation layer in both Figures 4.1 and 4.2, it is possible to use all the same technologies for the actual web page construction for a Semantic Web portal as those used in a database-backed portal. But, in contrast to conventional data-backed web portals, and because of the distributed nature of the RDF store that backs a Semantic Web portal, information on a single RDF-backed web page typically comes from multiple sources. The merge capability of an RDF store supports this sort of information distribution as part of the infrastructure of the web portal. When the portal is backed by RDF, there is no difference between building a distributed web portal and one in which all the information is local. Using RDF, federated web portals are as easy as siloed portals.

DATA FEDERATION

The RDF data model was designed from the beginning with data federation in mind. Information from any source is converted into a set of triples so that data federation of any kind—spreadsheets and XML, database tables and web pages—is accomplished with a single mechanism. As shown in Figure 4.2,

this strategy of federation converts information from multiple sources into a single format and then combines all the information into a single store. This is in contrast to a federation strategy in which the application queries each source using a method corresponding to that format. RDF does not refer to a file format or a particular language for encoding data but rather to the data model of representing information in triples. It is this feature of RDF that allows data to be federated in this way. The mechanism for merging this information, and the details of the RDF data model, can be encapsulated into a piece of software—the RDF store—to be used as a building block for applications.

The strategy of federating information first and then querying the federated information store separates the concerns of data federation from the operational concerns of the application. Queries written in the application need not know where a particular triple came from. This allows a single query to seamlessly operate over multiple data sources without elaborate planning on the part of the query author. This also means that changes to the application to federate further data sources will not impact the queries in the application itself.

This feature of RDF applications forms the key to much of the discussion that follows. In our discussion of RDFS and OWL, we will assume that any federation necessary for the application has already taken place; that is, all queries and inferences will take place on the *federated graph.* The federated graph is simply the graph that includes information from all the federated data sources over which application queries will be run.

SUMMARY

The components described in this chapter—RDF parsers, serializers, stores, and query engines—are not semantic models in themselves but the components of a system that will include semantic models. Even the information represented in RDF is not necessarily a semantic model. These are the building blocks that go into making and using a semantic model. The model will be represented in RDF, to be sure. As we shall see, the semantic modeling languages of the W3C, RDFS, and OWL are built entirely in RDF, and they can be federated just like any other RDF data.

Where do semantic models fit into the application architecture of Figure 4.2? As data expressed in RDF, they will be housed in the RDF store, along with all other data. But semantic models go beyond just including data that will be used to answer a query, like the list of plays that Shakespeare wrote or the places where paper machines are kept. Semantic models also include meta-data; data that help to organize other data. When we federate information from multiple sources, the RDF data model allows us to represent all the data in a single, uniform way. But it does nothing to resolve any conflicts of meaning between the sources. Do two states have the same definitions of "marriage"? Is the notion of "writing" a play the same as the notion of "writing" a song? It is the semantic models that give answers to questions like these. A semantic model acts as a sort of glue between disparate, federated data sources so we can describe how they fit together.

Just as Anyone can say Anything about Any topic, so also can anyone say anything about a model; that is, anyone can contribute to the definition and mapping between information sources. In this way, not only can a federated, RDF-based, semantic application get its information from multiple sources, but it can even get the instructions on how to combine information from multiple sources. In this way, the Semantic Web really is a web of meaning, with multiple sources describing what the information on the Web means.

Fundamental concepts

The following fundamental concepts were introduced in this chapter:

RDF parser/serializer—A system component for reading and writing RDF in one of several file formats.
RDF store—A database that works in RDF. One of its main operations is to merge RDF graphs.
RDF query engine—This provides access to an RDF store, much as an SQL engine provides access to a relational store.
SPARQL—The W3C standard query language for RDF.
SPARQL endpoint—An application that can answer a SPARQL query, including one where the native encoding of information is not in RDF.
Application interface—The part of the application that uses the content of an RDF store in an interaction with some user.
Converter—A tool that converts data from some form (e.g., tables) into RDF.
RDFa—Standard for encoding and retrieving RDF metadata from HTML pages.

RDF throughout this book). The queried RDF graph can be created from one kind of data or merged from many; in either case, SPARQL is the way to query it.

This chapter gives examples of the SPARQL query language. Most of the examples are based on version 1.0 of the standard, released in 2008. At the time of this writing, a new version of SPARQL is under development. The advanced examples will use features from the new standard (version 1.1) and will be indicated in the heading of each section as **(SPARQL 1.1)**. Features described in sections without this indication are available in both SPARQL 1.0 as well as SPARQL 1.1. Since this recommendation is still in progress, the final version might deviate in small ways from the examples given here.

SPARQL is a query language and shares many features with other query languages like XQUERY and SQL. But it differs from each of these query languages in important ways (as they differ from one another). Since we don't want to assume that a reader has a background in any specific query language (or even query languages at all), we begin with a gentle introduction to querying data. We start with the most basic information retrieval system, which we call a *Tell-and-Ask* system.

TELL-AND-ASK SYSTEMS

A Tell-and-Ask system is a simple system—you *tell* it some facts, and then it can answer questions you *ask* based on what you told it. Consider the following simple example:

Tell: James Dean played in the movie *Giant*.

Then you could ask questions like:

Ask: Who played in *Giant*?
Answer: James Dean
Ask: James Dean played in what?
Answer: *Giant*

You might tell it some more things, too, like:

Tell: James Dean played in *East of Eden*.
James Dean played in *Rebel Without a Cause*.

Then if you ask:

Ask: James Dean played in what?
Answer: *Giant, East of Eden, Rebel Without a Cause.*

One could imagine a sophisticated Tell-and-Ask system that understands natural language and can cope with questions like

Ask: What movies did James Dean star in?
Answer: *Giant, East of Eden, Rebel Without a Cause.*

Instead of using the simplified language in these examples. As we shall see, most real Tell-and-Ask systems don't do anything with natural language processing at all. In more complex Tell-and-Ask

systems, it can be quite difficult to be very specific about just what you really want to ask, so they usually use languages that are quite precise and technical.

On the face of it, it might seem that Tell-and-Ask systems aren't very interesting. They don't figure out anything you didn't tell them yourself. They don't do any calculations, they don't do any analysis. But this judgment is premature; even very simple Tell-and-Ask systems are quite useful. Let's have a look at one—a simple address book.

You have probably used an address book at some point in your life. Even a paper-and-pencil address book is a Tell-and-Ask system, though the process of telling it something (i.e., writing down an address) and the process of asking it something (looking up an address) take a lot of human effort. Let's think instead of a computer program that does the job of an address book. How does it work?

Like a paper address book, you tell it names and addresses (and probably phone numbers and email addresses and other information). Unlike the sample Tell-and-Ask system that we used to talk about James Dean and movies, you probably don't talk to your address book in anything that remotely resembles English; you probably fill out a form, with a field for a name, and another for the parts of the address, and so on. You "tell" it an address by filling in a form.

How do you ask a question? If you want to know the address of someone, you type in their name, perhaps into another form, very similar to the one you filled in to tell it the information in the first place. Once you have entered the name, you get the address.

The address book only gives you back what you told it. It does no calculations, and draws no conclusions. How could this be an interesting system? Even without any ability to do computations, address books are useful systems. They help us organize certain kinds of information that is useful to us and to find it when we need it. It is a simple Tell-and-Ask system—you tell it things, then you ask questions.

Even the address book is a bit more advanced than the simplest Tell-and-Ask system. When you look up an address in an address book, you usually get a lot more information than just the address. It is as if you asked a whole set of questions:

Ask: What is the address of Maggie Smith?
Ask: What is the phone number of Maggie Smith?
Ask: What is the email address of Maggie Smith?
And so on.

How can we make our address book system a bit more useful? There are a number of ways to enhance its behavior (and many of these are available in real address book applications). One way to make the address book "smarter" is to require less of the user who is asking questions. For instance, instead of typing in "Maggie Smith" when looking for an address, the system could let you just type in "Maggie," and look for any address where the name of the addressee *contains* the word "Maggie." Now it is as if you have asked

Ask: What is the address of everyone whose name includes "Maggie"?
You might get more answers if you do this—if, for instance, you also have an address for Maggie King, you'll get both addresses in response to your question.
You can go even further—you can ask your question based on some other information. You could ask about the address instead of the name, by filling in information in the address field:
Ask: Who lives at an address that contains "Downing"?

Common tell-and-ask infrastructure—spreadsheets

The address book was an example of a special-purpose tell-and-ask system; it is aimed at a single task, and has a fixed structure to the information you can tell it and ask it about. A spreadsheet is an example of a tell-and-ask system that is highly configurable and can be applied to a large number of situations. Spreadsheets are often cited as the most successful "killer application" ever; putting data management into the hands of intelligent people without the need to learn any heavy-duty technical skills. Spreadsheets apply the notion of WYSIWYG (What You See Is What You Get) to data management; a visual representation of data.

The "language" for telling information to a spreadsheet and asking information of a spreadsheet is visual; information is entered into a particular row and column and is retrieved by visually inspecting the table.

Since spreadsheets are primarily a visual presentation of data, you don't communicate with them in any particular language—much less natural language. You don't write "where does Maggie live?" to a spreadsheet; instead you search for Maggie in the "Name" column, and look into the "address" column to answer your question.

Spreadsheets become more cumbersome when the data aren't conveniently represented in a single table. Probably the simplest example of data that don't fit into a table is multiple values. Suppose we have more than one email address for Maggie Smith. How do we deal with this? We could have multiple email columns, like this:

Name	Email1	Email2
Maggie Smith	MSmith@acme.com	maggie@gmail.com

This solution works as long as nobody has three email addresses, etc. Another solution is to have a new row for Maggie, for each email address

Name	Email
Maggie Smith	MSmith@acme.com
Maggie Smith	maggie@gmail.com

This is a bit confusing, in that it is unclear whether we have one contact named "Maggie Smith" with two emails, or two contacts who happen to have the same name, one with each email address.

Spreadsheets also start to break down when an application requires highly interconnected data. Consider a contacts list that maintains names of people and the companies they work for. Then they maintain separate information for the companies—billing information, contract officer, etc. If this information is put into a single table, the relationship between the company and its information will be duplicated for each contact that works at that company, as illustrated in the table below:

Name	Email	Company	Contract Officer	Headquarters
Maggie Smith	MSmith@acme.com	ACME Product Inc.	Cynthia Wiley	Pittsburgh
Maggie King	MKing@acme.com	ACME Product Inc.	Cynthia Wiley	Pittsburgh

Both Maggies work for ACME, where the Contract Officer is Cynthia and the headquarters is in Pittsburgh. Duplicating information in this manner is error-prone as well as wasteful; for instance, if ACME gets a new contract officer, all the contact records for people who work for ACME need to be changed.

A common solution to this problem is to separate out the company information from the contact information into two tables, e.g.:

Name	Email	Company
Maggie Smith	MSmith@acme.com	ACME Product Inc.
Maggie King	MKing@acme.com	ACME Product Inc.

Company	Contract Officer	Headquarters
ACME Product Inc.	Cynthia Wiley	Pittsburgh

This sort of solution is workable in modern spreadsheet software, but begins to degrade the main advantages of spreadsheets; we can no longer use visualization to answer questions. Its structure relies on cross-references that are not readily visible by examining the spreadsheet. In fact, this sort of solution moves the tell-and-ask system from spreadsheets into a more structured form of tell-and-ask system, a relational database.

Advanced tell-and-ask infrastructure—relational database

Relational databases form the basis for most large-scale tell-and-ask systems. They share with spreadsheets a tabular presentation of data, but include a strong formal system (based on a mathematical formalism called the "relational algebra") that provides a systematic way to link tables together. This facility, along with some well-defined methodological support, allows relational databases to represent highly structured data, and respond to very detailed, structured questions.

Tell: Maggie King works for Acme Product Inc.
Tell: The contract officer for Acme Product Inc. is Cynthia Wiley
Tell: Cynthia Wiley's email address is CJWiley@acme.com
Ask: What is the email address for the contract officer at the company where Maggie King works?
Answer: CJWiley@acme.com

This sort of detailed structure comes at a price—asking a question becomes a very detailed process, requiring a specialized language. Such a language is called a *query language*.

In the query language for relational databases, links from one table to another are done by cross-referencing a closer rendering of the question above, in a query language for a relational database, would be:

Ask: What is the email address for the person matched by the "contract officer" reference for the company matched by the "works for" reference for the person whose name is "Maggie King"?
Answer: CJWiley@acme.com

This might seem like a needlessly wordy way to ask the question, but this is how you pose questions precisely enough to recover information from a complex database structure.

RDF AS A TELL-AND-ASK SYSTEM

RDF is also a tell-and-ask system. Like a relational database, RDF can represent complex structured data. Also like a relational database, RDF requires a precise query language to specify questions. Unlike a relational database, the cross-references are not visible to the end user, and there is no need to explicitly represent them in the query language.

As discussed in previous chapters, in RDF, relationships are represented as triples. Asserting a triple amounts to TELLing the triple store a fact.

Tell: James Dean played in the movie *Giant*.

How do we **ASK** questions of this? Even with a single triple, there are already some questions we could ask:

Ask: What did James Dean play in?
Ask: Who played in *Giant*?
Ask: What did James Dean do in *Giant*?

All of these questions can be asked in SPARQL in a simple way, by replacing part of the triple with a question word, like Who, What, Where, etc. SPARQL doesn't actually distinguish between question words, so we can choose words that make sense in English. In SPARQL, question words are written with a question mark at the start, e.g., *?who*, *?where*, *?when*, etc.

Ask: James Dean played in ?what
Answer: *Giant*
Ask: ?who played in *Giant*
Answer: James Dean
Ask: James Dean ?what *Giant*
Answer: played in

This is the basic idea behind SPARQL—that you can write a question that looks a lot like the data, with a question word standing in for the thing you want to know. Like query languages for relational databases and spreadsheets, SPARQL makes no attempt to mimic the syntax of natural language, but it does use the idea that a question can look just like a statement, but with a question word to indicate what we want to know.

SPARQL—QUERY LANGUAGE FOR RDF

The syntax of SPARQL actually looks like Turtle. So these examples really look more like this:

Tell: `:JamesDean :playedIn :Giant .`
Ask: `:JamesDean :playedIn ?what .`
Answer: `:Giant`
Ask: `?who :playedIn :Giant .`
Answer: `:JamesDean`
Ask: `:JamesDean ?what :Giant .`
Answer: `:playedIn`

Before we go further, let's talk a bit about the syntax of a SPARQL query. We'll start with a simple form, the SELECT query. Readers familiar with SQL will notice a lot of overlap with SPARQL syntax (e.g., keywords like SELECT and WHERE). This is not coincidental; SPARQL was designed to be easily learned by SQL users.

A SPARQL SELECT query has two parts; a set of question words, and a question pattern. The keyword WHERE indicates the selection pattern, written in braces. We have already seen some question patterns, e.g.,

```
WHERE {:JamesDean :playedIn ?what .}
WHERE {?who :playedIn :Giant .}
WHERE {:JamesDean ?what :Giant .}
```

The query begins with the word SELECT followed by a list of the question words. So the queries for the questions above are

```
SELECT ?what WHERE {:JamesDean ?playedIn ?what .}
SELECT ?who WHERE {?who :playedIn :Giant .}
SELECT ?what WHERE {:JamesDean ?what :Giant .}
```

It might seem that listing the question words in the SELECT part is redundant—after all, they appear in the patterns already. To some extent, this is true, but we'll see later how modifying this list can be useful.

RDF (and SPARQL) deals well with multiple values. If we TELL the system that James Dean played in multiple movies, we can do this without having to make any considerations in the representation:

```
Tell: :JamesDean :playedIn :Giant .
Tell: :JamesDean :playedIn :EastOfEden .
Tell: :JamesDean :playedIn :RebelWithoutaCause .
```

Now if we ASK a question with SPARQL

```
Ask: SELECT ?what WHERE {:JamesDean :playedIn ?what}
Answer: :Giant, :EastOfEden, :RebelWithoutaCause.
```

The WHERE clause of a SPARQL query can be seen as a *graph pattern*, that is, a pattern that is matched against the data graph. In this case, the pattern has just one triple, `:JamesDean` as the subject, `:playedIn` as the predicate, and a question word as the object. The action of the query engine is to find all matches for the pattern in the data, and to return all the values that the question word matched.

We can see this as a graph—Figure 5.1 shows the James Dean data in the form of a graph, and the WHERE clause as a graph pattern. There are three matches for this pattern in the graph, where a match has to match resources in the pattern exactly, but anything can match the question word.

Suppose we follow the data along further. Each of these movies is directed by someone. Some of them might be directed by more than one person, as shown in Figure 5.2. Who were the directors that

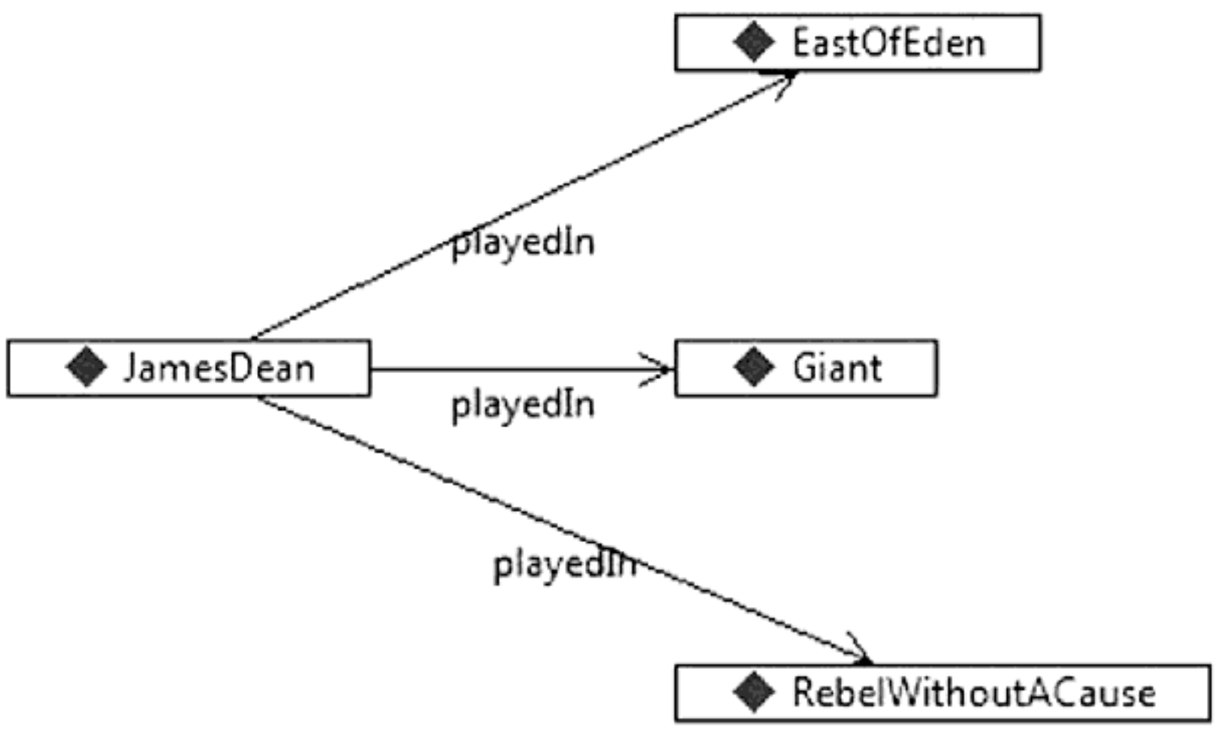


FIGURE 5.1

James Dean data in a graph, and a query to fetch it.

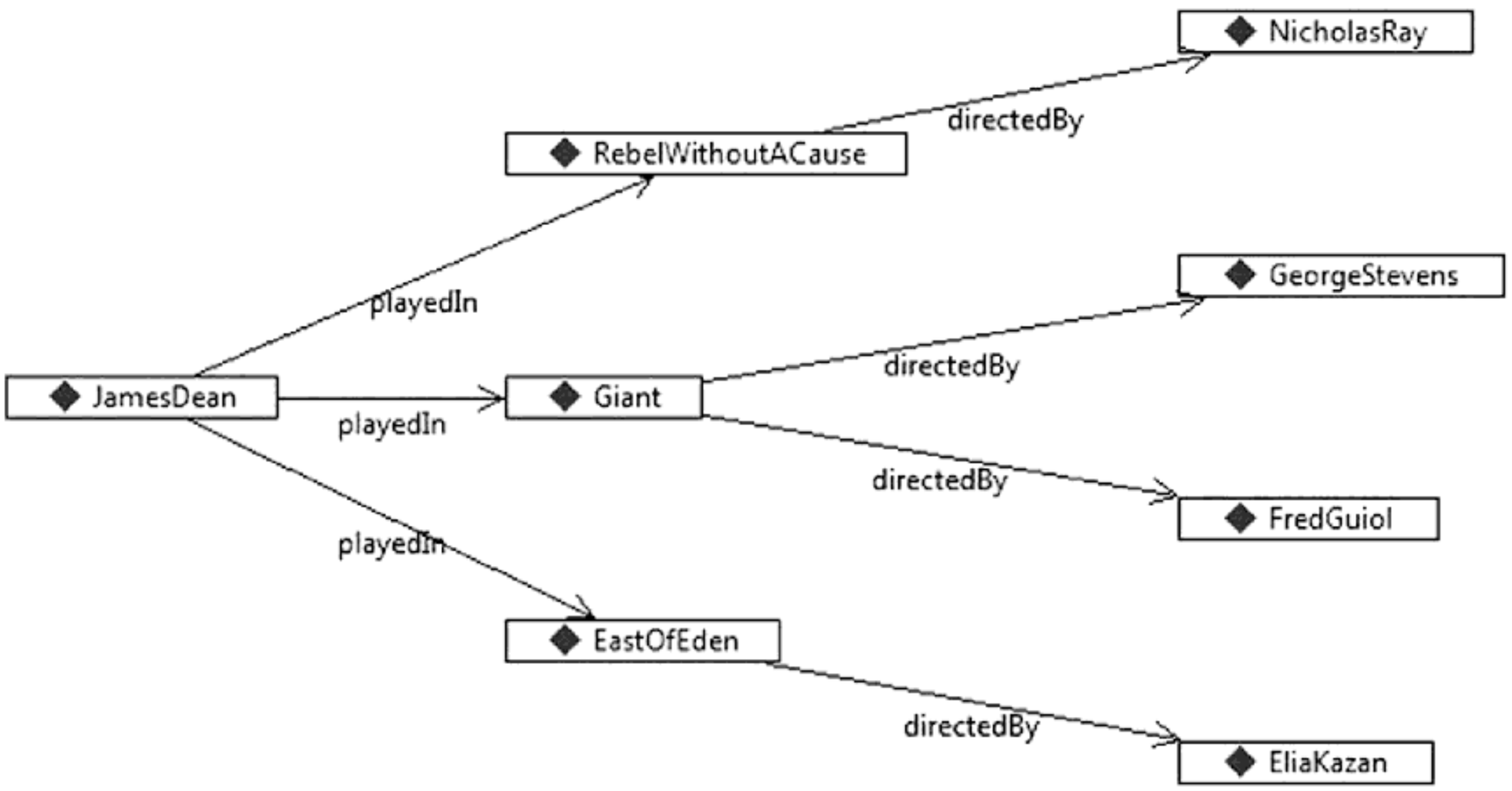


FIGURE 5.2

James Dean's movies and their directors.

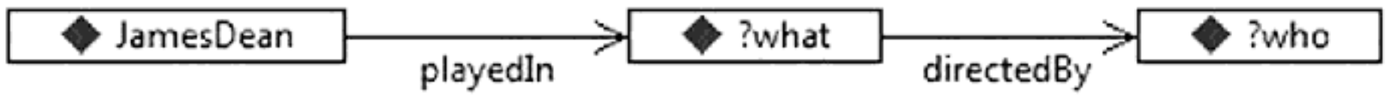


FIGURE 5.3

Graphic version of a query to find James Dean's directors.

James Dean worked with? We can query this by asking who directed the movies that James Dean played in. The graph pattern for this query has two triples:

```
:JamesDean :playedIn ?what .
?what :directedBy ?who .
```

Since the variable *?what* appears in both triples, the graph pattern is joined at that point. We can draw a graph pattern the same way we draw a graph. Figure 5.3 shows this graph pattern. There are two triples in the pattern and two triples in the diagram.

This graph pattern has two question words, so the query engine will find all matches for the pattern, with both question words being free to match any resource. This results in several matches:

```
?what=:Giant                    ?who=:GeorgeStevens
?what=:Giant                    ?who=:FredGuiol
?what=:EastOfEden               ?who=:EliaKazan
?what=:RebelWithoutaCause       ?who=:NicholasRay
```

When we have more than one question word, we might actually only be interested in one of them. In this case, we asked what directors James Dean had worked with; the movies he played in were just a means to that end. This is where the details of the SELECT clause come in—we can specify which question words we are interested in. So the complete query to find James Dean's directors looks like this:

Ask:

```
SELECT ?who
WHERE {:JamesDean :playedIn ?what .
       ?what :directedBy ?who .}
```

Answer:

```
:GeorgeStevens,    :EliaKazan,    :NicholasRay,    :FredGuiol
```

Since a query in SPARQL can have several question words and several answers, it is convenient to display the answers in a table with one column for each question word and one row for each answer.

?who
:GeorgeStevens
:EliaKazan
:NicholasRay
:FredGuiol

If we decide to keep both question words in the SELECT, we will have more columns in the table

Ask:

```
SELECT ?what ?who
WHERE {:JamesDean :playedIn ?what .
       ?what :directedBy ?who .}
```

?what	?who
:Giant	:GeorgeStevens
:Giant	:FredGuiol
:EastOfEden	:EliaKazan
:RebelWithoutaCause	:NicholasRay

Naming question words in SPARQL

In English, we have a handful of question words—who, what, where, etc. A question doesn't make sense if we use some other word. But in SPARQL, a question word is just signaled by the *?* at the start—any word would do just as well. If we look at the output table above, the question words *?who* and *?what* are not very helpful in describing what is going on in the table. If we remember the question, we know what they mean (*?what* is a movie, and *?who* is its director). But we can make the table more understandable, even out of the context of the question, by selecting descriptive question words. It is customary in SPARQL to do this, to communicate the intention of a question word to someone who might want to read the query. For example, we might write this query as

Ask:

```
SELECT ?movie ?director
WHERE {:JamesDean :playedIn ?movie .
       ?movie :directedBy ?director .}
```

Answer:

?movie	?director
:Giant	:GeorgeStevens
:Giant	:FredGuiol
:EastOfEden	:EliaKazan
:RebelWithoutaCause	:NicholasRay

The graph pattern we just saw was a simple chain—James Dean played in some movie that was directed by someone. Graph patterns in SPARQL can be as elaborate as the graphs they match against. For instance, given a more complete set of information about James Dean movies we could find the actresses who worked with him with a graph pattern:

Ask:

```
SELECT ?actress ?movie
WHERE {:JamesDean :playedIn ?movie .
       ?actress :playedIn ?movie .
       ?actress rdf:type :Woman }
```

Answer:

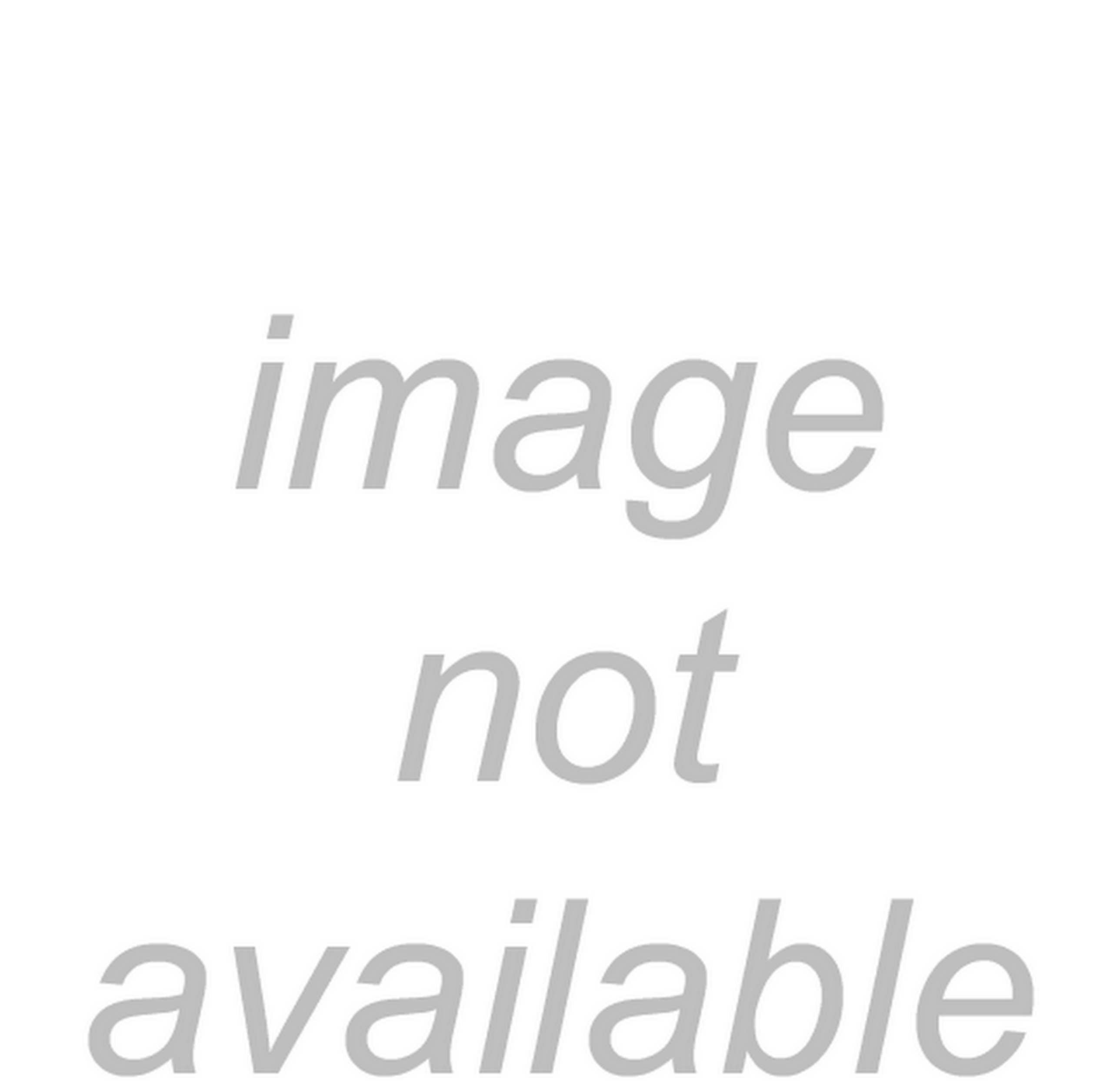


image
not
available

Remember that *?actress* is just a question word like *?who*, renamed to be more readable; the only reason *?actress* doesn't match `:RockHudson` is because the data do not support the match. That is, the meaning of *?actress* is not given by its name, but instead by the structure of the graph pattern.

With this observation, one might wonder how we know that *?movie* is sure to come up with movies? And indeed this is a consideration; in this example, the only things that James Dean played in were movies, so it really isn't an issue. But on the Semantic Web, we could have more information about things that James Dean played in. So we really should restrict the value of the question word *?movie* to be a member of the class movie. We can do this by adding one more triple to the pattern:

Ask:

```
SELECT ?actress
WHERE {:JamesDean :playedIn ?movie .
       ?movie rdf:type :Movie .
       ?actress :playedIn ?movie .
       ?actress rdf:type :Woman }
```

This query pattern is shown graphically in Figure 5.5.
Triples like

```
?movie rdf:type :Movie .
```

can seem a bit confusing at first—with two uses of the word *movie*, what does this mean? But now that we know that *?movie* is just a generic question word with a name that prints well in a table, we can see that this triple is how we tell SPARQL what we really mean by *?movie*. The meaning isn't in the name, so we have to put it in a triple.

You might come upon a model that gives properties the same name as the class of entity they are intended to point to—so that instead of a property called `:directedBy` in this example, a model might call it simply `:director`. In such a case, the query to find the people who directed James Dean movies would look like this:

```
SELECT ?director
WHERE {:JamesDean :playedIn ?movie .
?movie :director ?director .}
```

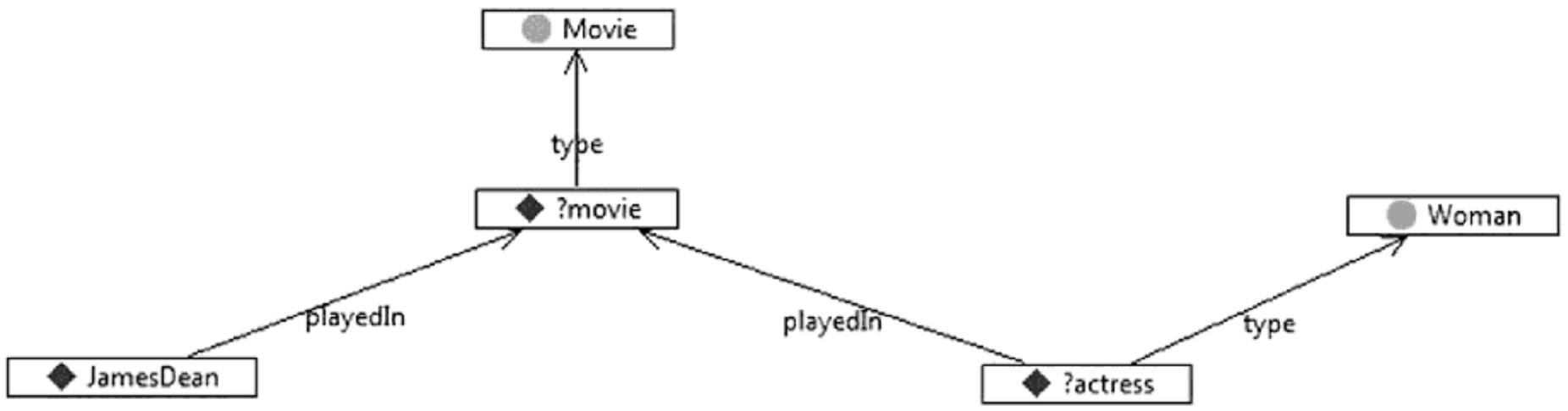


FIGURE 5.5

Extended query pattern that includes the fact that ?movie has type Movie.

This can look a bit daunting—what is the difference between *?director* and `:director`? As we've already seen, `:director` refers to a particular resource (using the default namespace—that's what the ":" means) On the other hand, *?director* is a question word—it could have been *?foo* or *?bar* just as easily, but we chose *?director* to remind us of its connection with a movie director, when we see it out of the context of the query. If you have to write (or read!) a query written for a model that names its properties in this way, don't panic! Just remember that the *?* marks a question word—the name *?director* is just there to let you (and whoever else reads the query) know what *?director* is expected to mean. If you are creating your own model, we recommend that you use property names like `:directedBy` instead of `:director` so that you don't invite this confusion in the people who want to query data using your model.

Query structure vs. data structure

In Figure 5.4, we saw how the graph pattern in a query looks a lot like the data graph it matches against. This is true of graph patterns in general. The complexity of a question that can be specified with SPARQL is limited only by the complexity of the data. We could, for instance, ask about actresses who played in a movie with James Dean who themselves were in a movie directed by John Ford:

Ask:

```
SELECT ?actress ?movie
WHERE {:JamesDean :playedIn ?movie .
       ?actress :playedIn ?movie .
       ?actress a :Woman .
       ?actress :playedIn ?anotherMovie .
       ?anotherMovie :directedBy :JohnFord .}
```

Answer:

?actress
NatalieWood
CarrollBaker

Figure 5.6 shows this query as a graph. In the text version of the query, we often see the same question word appear in multiple triples, and some of them even refer to the same kind of thing (?movie, *?anotherMovie*). In the graph version, we see that these are the points where two triples must refer to the same thing. For instance, we know that James Dean and *?actress* played in the same movie, because both triples use the same question word (*?movie*) for that movie. Similarly, that *?actress* is the same one who played in *?anotherMovie*, because there the same question word *?actress* appears in those two triples. All these relationships are visibly apparent in Figure 5.6, where we see that *?movie* is the connection between James Dean and *?actress*, *?actress* is the connection between *?movie* and *?anotherMovie*, and *?anotherMovie* is the connection between *?actress* and John Ford.

If we look at the information supporting the answer "Natalie Wood," we see that the data graph looks just like the graph pattern—this should come as no surprise, since that is how the pattern works. But we can use this feature to our advantage when writing queries. One way to write a complex query like the one in Figure 5.6 is to walk through the question we want to answer, writing down triples as we

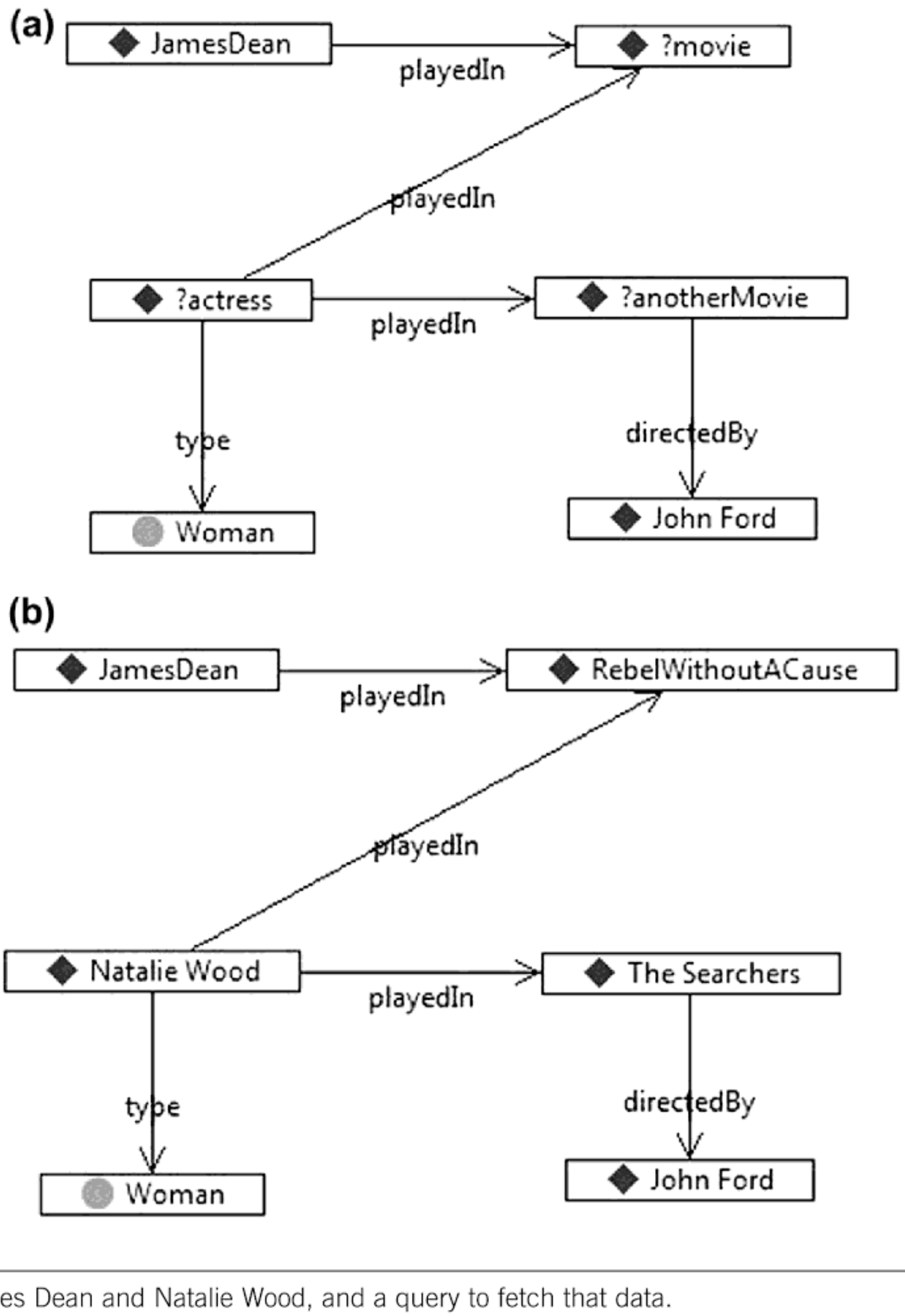


FIGURE 5.6

Data about James Dean and Natalie Wood, and a query to fetch that data.

go until we have the full query. But another way is to start with the data. Suppose we have an example of what we want to search for, for example, we know that Natalie Wood played in *The Searchers,* which was directed by John Ford. Next, we show how we can use the close match between graphs and patterns to construct the pattern from the example.

Since the example from the data graph matches the graph pattern triple for triple, we already know a lot about the graph pattern we want to create. The only thing we need to specify is which values from the example we want to keep as literal values in the pattern, and which ones we want to replace with

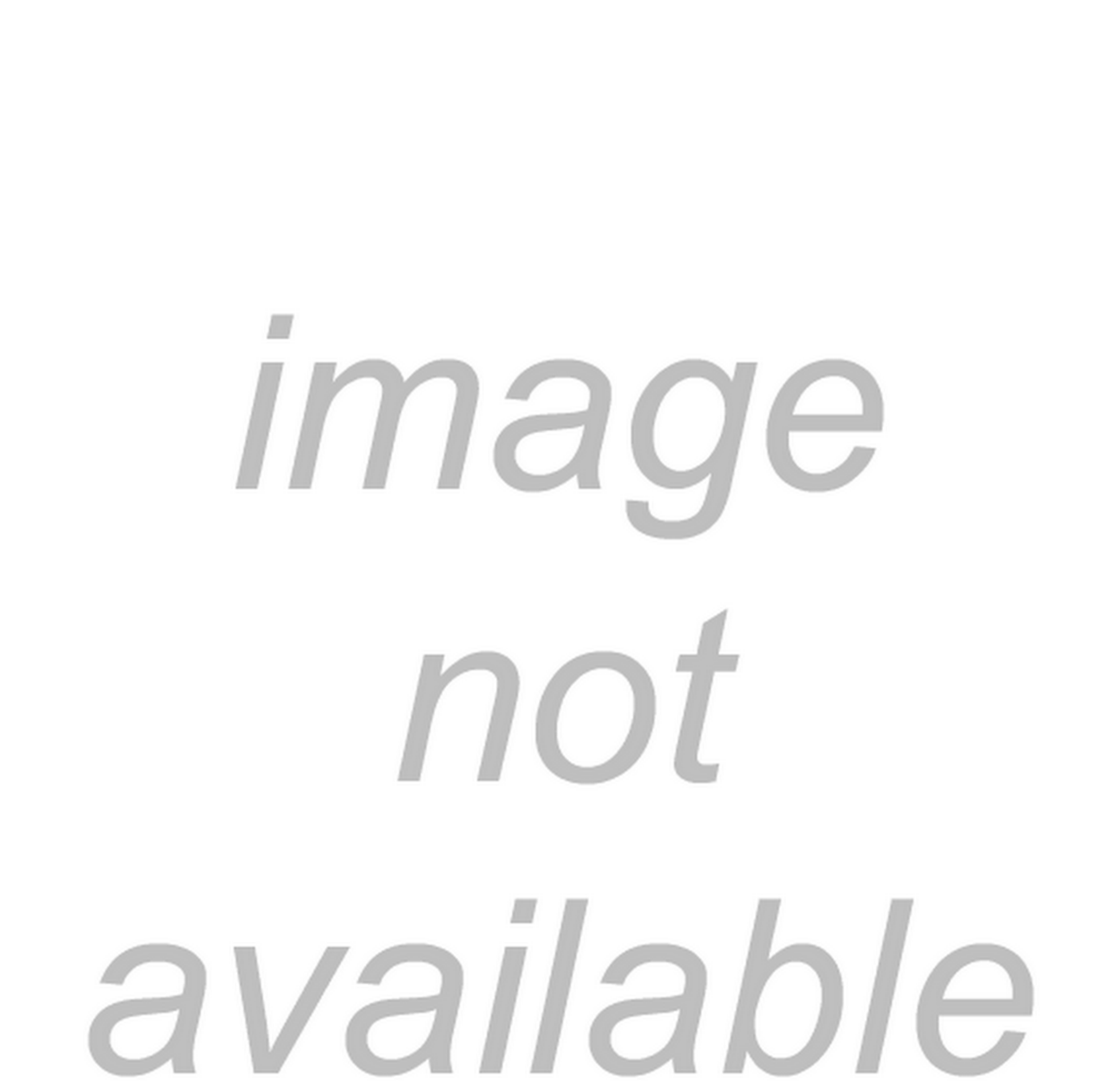


image
not
available

Now we can create our SPARQL query by simply copying down the graph pattern in Turtle. Each arrow in the graph becomes a triple. If a particular entity (either a literal resource or a question word) participates in more than one triple in the graph, then it will appear more than one time in the Turtle rendering of the pattern. The graph diagram in Figure 5.7(b) has five connecting arrows; the corresponding query will have the same number of triples:

```
{       :JamesDean :playedIn ?q1 .
        ?q3 :playedIn ?q1 .
        ?q3 rdf:type :Woman .
        ?q3 :playedIn ?q2 .
        ?q2 :directedBy :JohnFord .}
```

To complete the query, simply SELECT the question word(s) you want to report on, and perhaps give it a meaningful name. This brings us back to a query very like the original query (differing only in the names of the unselected question words)—but this time, it was generated from a pattern in the data.

```
SELECT ?actress
WHERE {:JamesDean :playedIn ?q1 .
       ?actress :playedIn ?q1 .
       ?actress rdf:type :Woman .
       ?actress :playedIn ?q2 .
       ?q2 :directedBy :JohnFord .}
```

As we saw above, there are two matches for this query in the sample data, Natalie Wood (no surprise there—after all, it was her performance that we used as a model for this query) and Carroll Baker. Carroll Baker is similar to Natalie Wood, in that she is also a woman, she also played alongside James Dean in a movie, and she was also directed by John Ford. She is similar to Natalie Wood in exactly the features specified in the query.

This method for creating queries can be seen as a sort of "more like this" capability; once you have one example of something you are interested in, you can ask for "more like this," where the notion of "like this" is made specific by including particular triples in the example, and hence in the graph pattern.

For example, we just wrote a query that found actresses who played in James Dean movies and also played in movies directed by John Ford. How do we know that the results are limited to actresses? In the example, Natalie Wood is an actress. But she is one of the resources that we replaced by a question word—how do we know that all the things that the pattern matches will also be actresses? We know this because we included the triple

```
?q3 rdf:type :Woman .
```

in the example, and hence in the pattern.

What would happen if we left that triple out? Natalie Wood is, of course, still a Woman in the data graph, but we haven't included that fact in our example. So that fact does not get copied into the query. Our new query looks like this:

engine will draw exactly the same conclusions from an inferred triple as it would have done, had that same triple been asserted.

EXAMPLE Asserted versus Inferred Triples

Even with a single inference rule like the type propagation rule, we can show the distinction of asserted vs. inferred triples. Suppose we have the following triples in a triple store:

```
shop:Henleys rdfs:subClassOf shop:Shirts.
shop:Shirts rdfs:subClassOf shop:MensWear.
shop:Blouses rdfs:subClassOf shop:WomensWear.
shop:Oxfords rdfs:subClassOf shop:Shirts.
shop:Tshirts rdfs:subClassOf shop:Shirts.
shop:ChamoisHenley rdf:type shop:Henleys.
shop:ClassicOxford rdf:type shop:Oxfords.
shop:ClassicOxford rdf:type shop:Shirts.
shop:BikerT rdf:type shop:Tshirts.
shop:BikerT rdf:type shop:MensWear.
```

These triples are shown graphically in Figure 6.2.

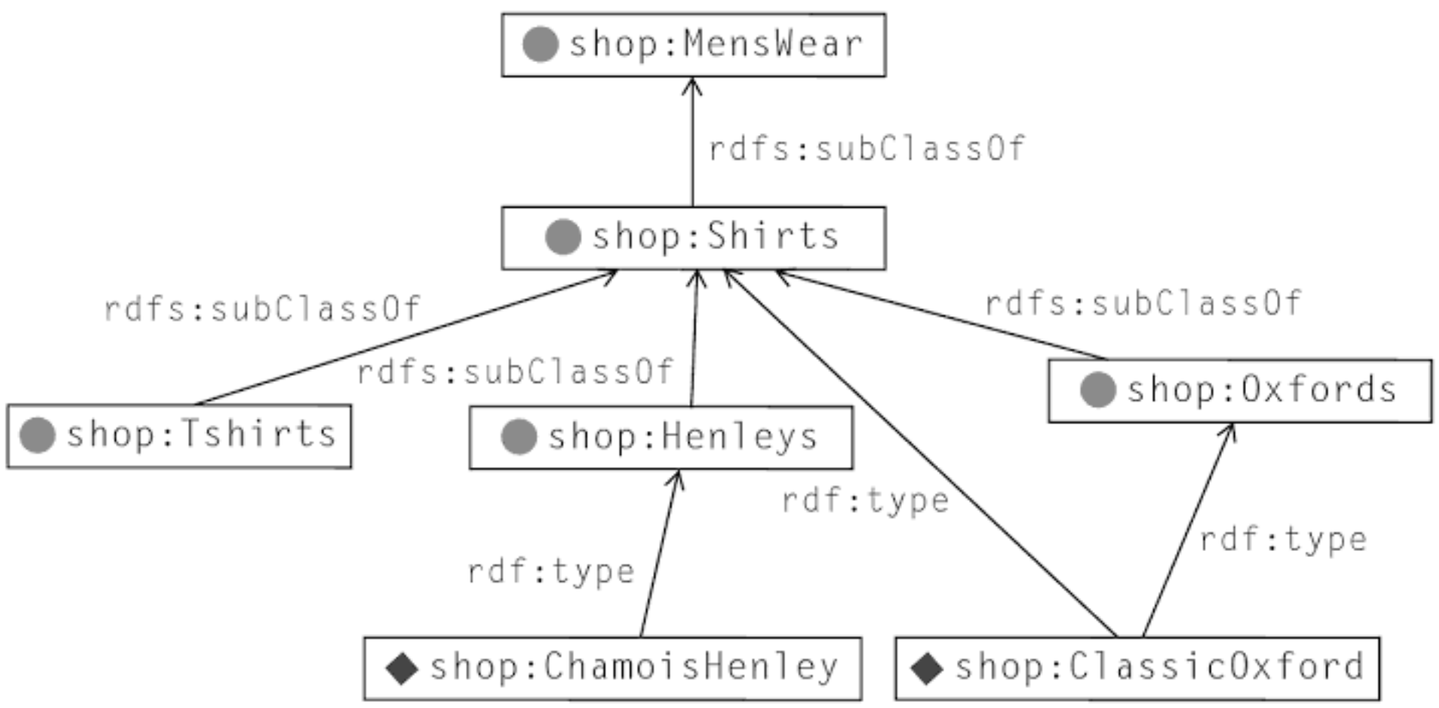


FIGURE 6.2

Asserted triples in the catalog model.

An inferencing query engine that enforces just the type propagation rule will draw the following inferences:

```
shop:ChamoisHenley rdf:type shop:Shirts.
shop:ChamoisHenley rdf:type shop:MensWear.
shop:ClassicOxford rdf:type shop:Shirts.
shop:ClassicOxford rdf:type shop:MensWear.
shop:BikerT rdf:type shop:Shirts.
shop:BikerT rdf:type shop:MensWear.
```

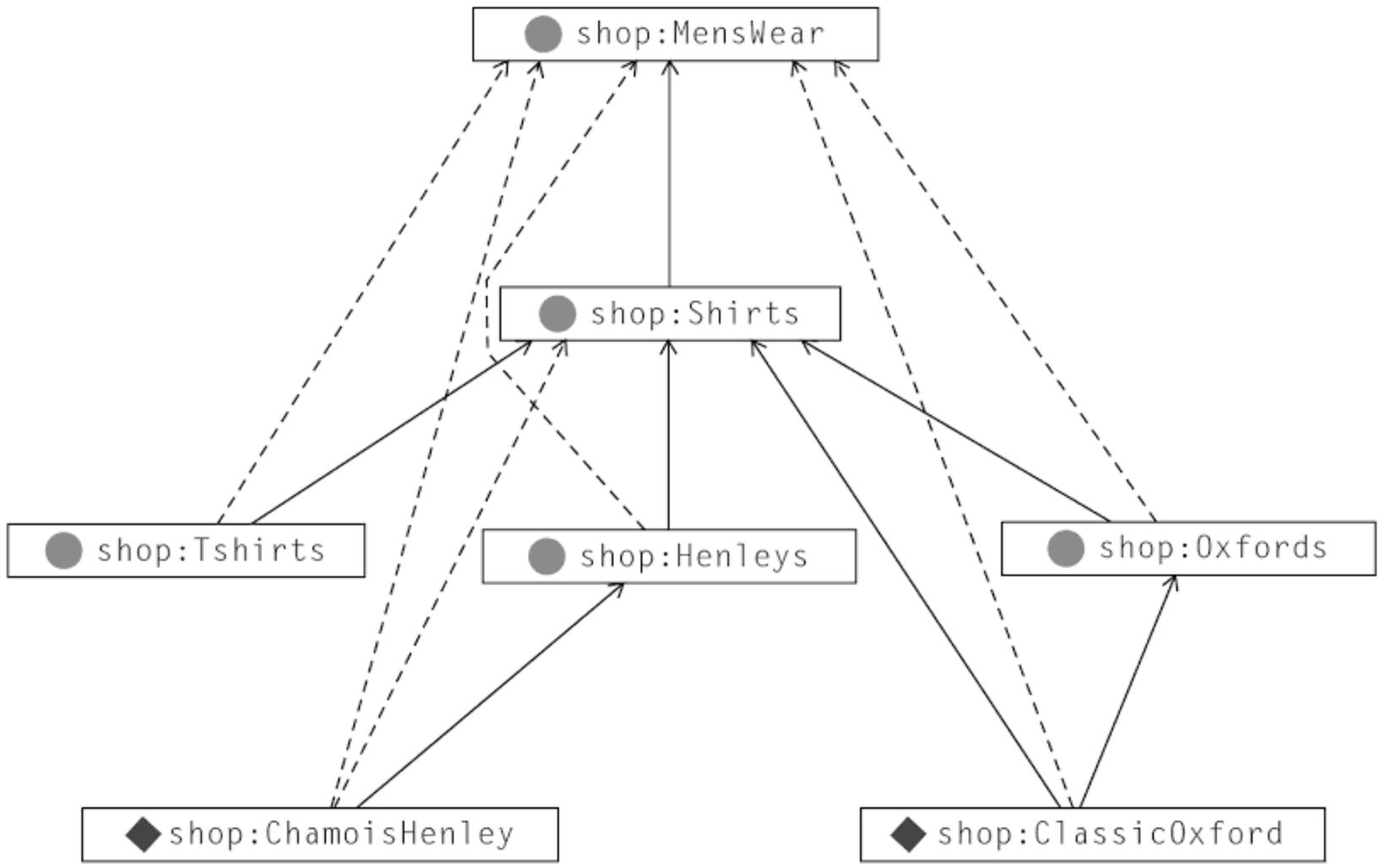


FIGURE 6.3

All triples in the catalog model. Inferred triples are shown as dashed lines.

Some of these triples were also asserted; the complete set of triples over which queries will take place is as follows, with inferred triples in italics:

```
shop:Henleys rdfs:subClassOf shop:Shirts.
shop:Shirts rdfs:subClassOf shop:MensWear.
shop:Blouses rdfs:subClassOf shop:WomensWear.
shop:Oxfords rdfs:subClassOf shop:Shirts.
shop:TShirts rdfs:subClassOf shop:Shirts.
shop:ChamoisHenley rdf:type shop:Henleys.
shop:ChamoisHenley rdf:type shop:Shirts.
shop:ChamoisHenley rdf:type shop:MensWear.
shop:ClassicOxford rdf:type shop:Oxfords.
shop:ClassicOxford rdf:type shop:Shirts.
shop:ClassicOxford rdf:type shop:MensWear.
shop:BikerT rdf:type shop:Tshirts.
shop:BikerT rdf:type shop:Shirts.
shop:BikerT rdf:type shop:MensWear.
```

All triples in the model, both asserted and inferred, are shown in Figure 6.3. We use the convention that asserted triples are printed with unbroken lines, and inferred triples are printed with dashed lines. This convention is used throughout the book.

The situation can become a bit more subtle when we begin to merge information from multiple sources in which each source itself is a system that includes an inference engine. Most RDF implementations

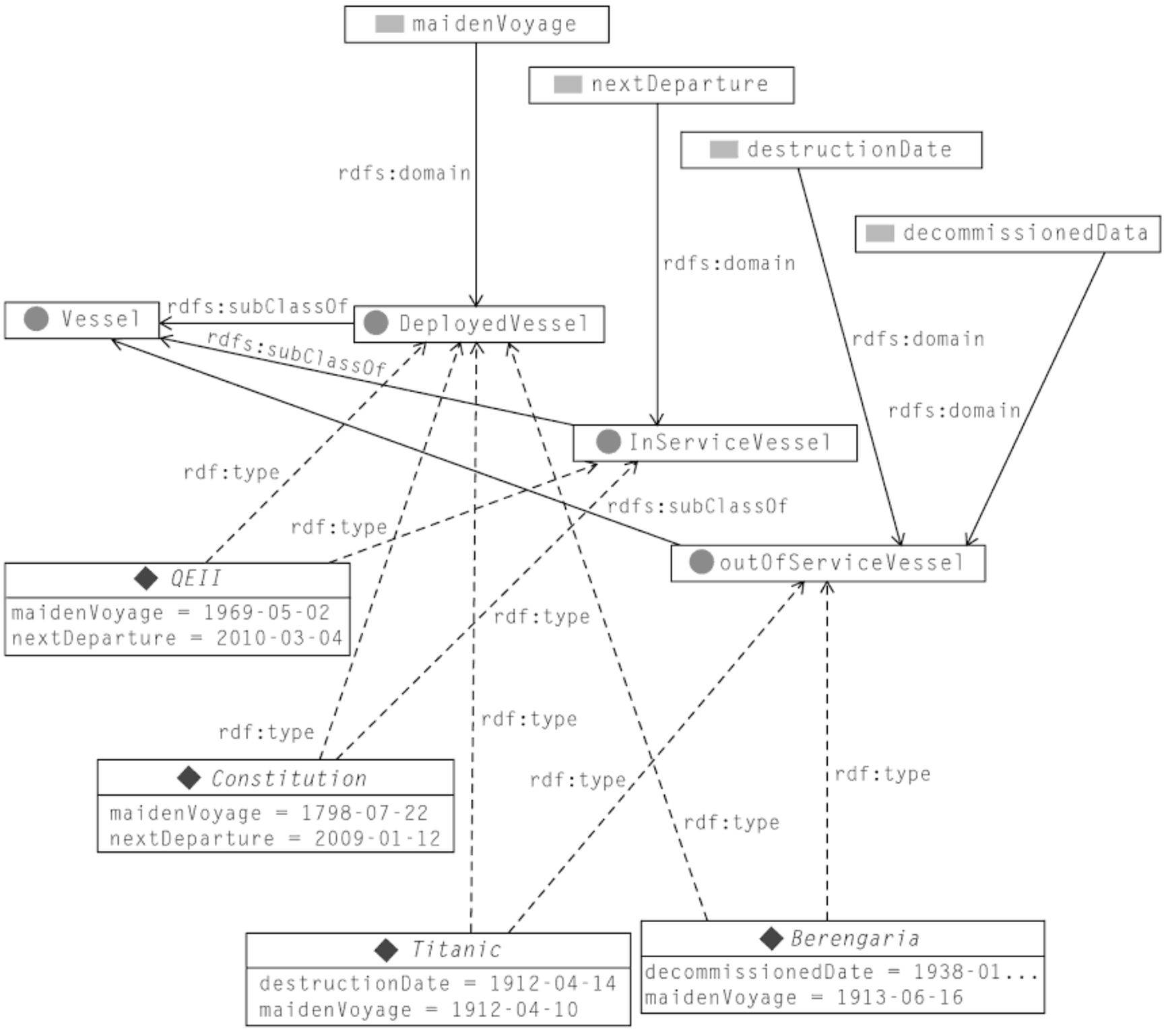


FIGURE 7.3

Inferring classes of vessels from the information known about them.

From the information in Table 7.1, we can infer that all of `Johnson`, `Warwick`, `Smith`, and `Preble` are members of the class `ship:Captain`. These inferences, as well as the triples that led to them, can be seen in Figure 7.4.

Filtering undefined data

A related challenge is to sort out individuals based on the information that is defined for them. The set of individuals for which a particular value is defined should be made available for future processing; those for which it is undefined should not be processed.

CHALLENGE 12

In the preceding example, the set of vessels for which `nextDeparture` is defined could be used as input to a scheduling system that plans group tours. Ships for which no `nextDeparture` is known should not be considered.

But in many cases, modern versions of such systems do model the schema in the same form as the data; the meta-object protocol of Common Lisp and the introspection API of Java represent the object models as objects themselves. The XML Stylesheet Definition defines XML Styles in an XML language.

In the case of RDF, the schema language was defined in RDF from the very beginning. That is, all schema information in RDFS is defined with RDF triples. The relationship between "plain" resources in RDF and schema resources is made with triples, just like relationships between any other resources. This elegance of design makes it particularly easy to provide a formal description of the semantics of RDFS, simply by providing inference rules that work over patterns of triples. While this is good engineering practice (in some sense, the RDF standards committee learned a lesson from the issues that the XML standards had with DTDs), its significance goes well beyond its value as good engineering. In RDF, everything is expressed as triples. The meaning of asserted triples is expressed in new (inferred) triples. The structures that drive these inferences, that describe the meaning of our data, are also in triples. This means that this process can continue as far as it needs to; the schema information that provides context for information on the Semantic Web can itself be distributed on the Semantic Web.

We can see this in action by showing how a set is defined in RDFS. The basic construct for specifying a set in RDFS is called an `rdfs:Class`. Since RDFS is expressed in RDF, the way we express that something is a class is with a triple—in particular, a triple in which the predicate is `rdf:type`, and the object is `rdfs:Class`. Here are some examples that we will use in the following discussion:

```
:AllStarPlayer rdf:type rdfs:Class.
:MajorLeaguePlayer rdf:type rdfs:Class.
:Surgeon rdf:type rdfs:Class.
:Staff rdf:type rdfs:Class.
:Physician rdf:type rdfs:Class.
```

These are triples in RDF just like any other; the only way we know that they refer to the schema rather than the data is because of the use of the term in the `rdfs:` namespace, `rdfs:Class`. But what is new here? In Chapter 3, we already discussed the notion of `rdf:type`, which we used to specify that something was a member of a set. What do we gain by specifying explicitly that something is a set? We gain a description of the meaning of membership in a set. In RDF, the only "meaning" we had for set membership was given by the results of some query; `rdf:type` actually didn't behave any differently from any other (even user-defined) property. How can we specify what we *mean* by set membership? In RDFS, we express meaning through the mechanism of inference.

THE RDF SCHEMA LANGUAGE

RDFS "extends" RDF by introducing a set of distinguished resources into the language. This is similar to the way in which a traditional programming language can be extended by defining new language-defined keywords. But there is an important difference: In RDF, we already had the capability to use

Table 11.2 Sample Triples

Subject	Predicate	Object
mfg:Product1	rdf:type	mfg:Product
mfg:Product1	mfg:Product_ID	1
mfg:Product1	mfg:Product_ModelNo	ZX-3
mfg:Product1	mfg:Product_Division	Manufacturing support
mfg:Product1	mfg:Product_Product_Line	Paper machine
mfg:Product1	mfg:Product_Manufacture_Location	Sacramento
mfg:Product1	mfg:Product_SKU	FB3524
mfg:Product1	mfg:Product_Available	23
mfg:Product2	rdf:type	mfg:Product
mfg:Product2	mfg:Product_ID	2
mfg:Product2	mfg:Product_ModelNo	ZX-3P
mfg:Product2	mfg:Product_Division	Manufacturing support
mfg:Product2	mfg:Product_Product_Line	Paper machine
mfg:Product2	mfg:Product_Manufacture_Location	Sacramento
mfg:Product2	mfg:Product_SKU	KD5243
mfg:Product2	mfg:Proudct_Available	4
mfg:Product3	rdf:type	mfg:Product
mfg:Product4	rdf:type	mfg:Product
mfg:Product5	rdf:type	mfg:Product …

This is a common situation when actually importing information from a table. It is quite common for type information to appear as a particular column in the table. If we use a single method for importing tables, all the rows become individuals of the same type. A software-intensive solution would be to write a more elaborate import mechanism that allows a user to specify which column should specify the type. A model-based solution would use a model in OWL and an inference engine to solve the same problem.

CHALLENGE 28

Build a model in OWL so we can infer the type information for each individual, based on the value in the "Product Line" field but using just the simple imported triples described in Chapter 3.

Solution

Since the classes of which the rows will be members (i.e., the product lines) are already known, we first define those classes:

```
ns:Paper_Machine rdf:type owl:Class.
ns:Feedback_Line rdf:type owl:Class.
ns:Active_Sensor rdf:type owl:Class.
ns:Monitor rdf:type owl:Class.
ns:Safety_Valve rdf:type owl:Class.
```

Formally, this is the same as asserting the 28 `owl:differentFrom` triples, one for each pair of individuals in the list. In the case of James Dean's movies, we can assert that the three movies are distinct in the same way:

```
[a owl:AllDifferent;
 owl:distinctMembers (:EastOfEden
                      :Giant
                      :Rebel)].
```

The view of this bit of N3 in terms of triples is shown in Figure 12.2. The movies are referenced in an RDF list (using `rdf:first` and `rdf:rest` to chain the entities together). For longer lists (like the planets), the chain continues for each entity in the list.

Earlier we saw that the class `JamesDeanMovie` was defined using `owl:oneOf` to indicate that these are the only James Dean movies in existence. Now we have gone on to say that additionally these three movies are distinct. It is quite common to use `owl:oneOf` and `owl:AllDifferent` together in this way to say that a class is made up of an enumerated list of distinct elements.

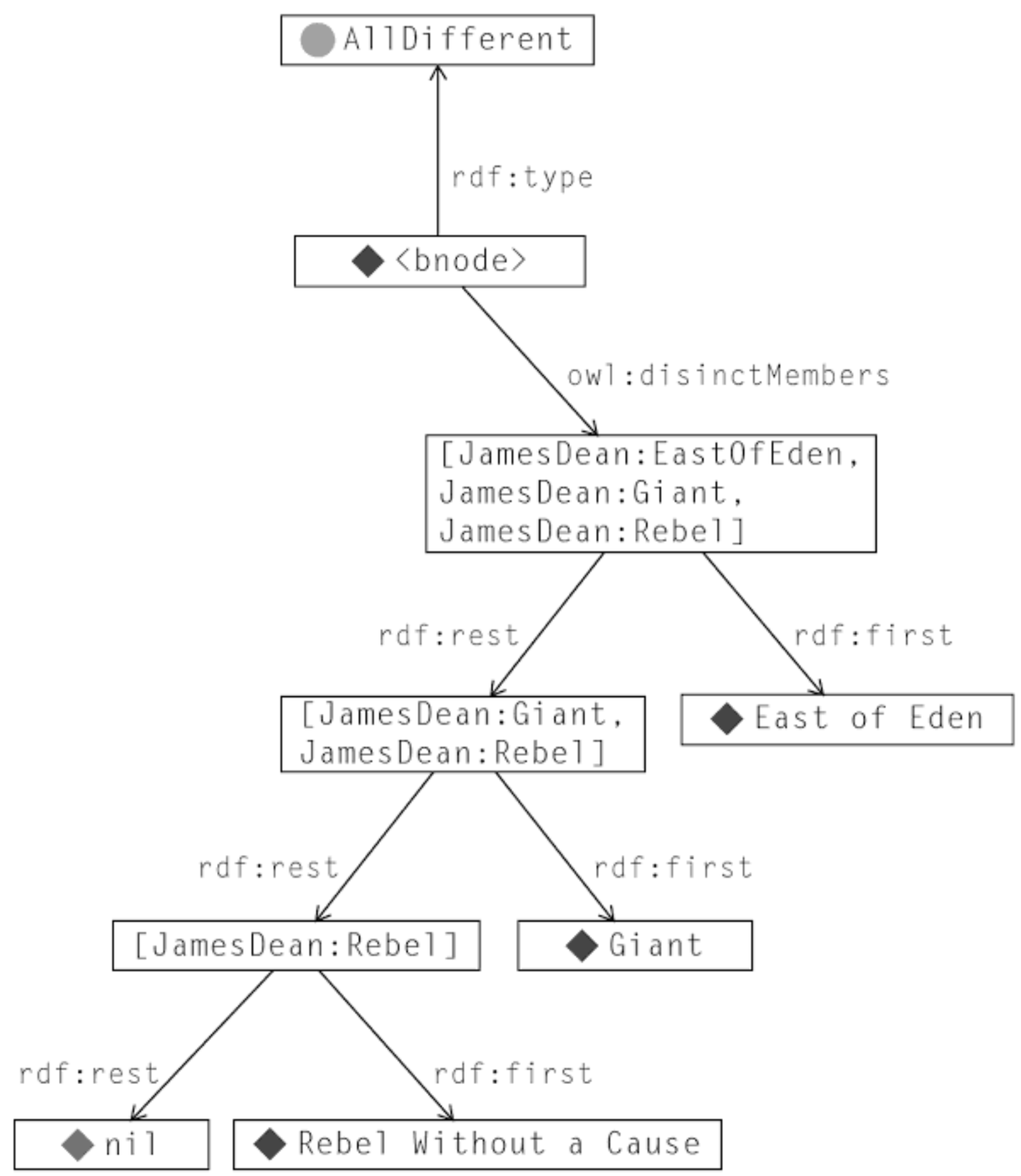


FIGURE 12.2

Using `owl:AllDifferent` and `owl:distinctMembers` to indicate that the three James Dean movies are distinct. The movies are referred to in an RDF list.

CHAPTER

RDF schema

7

CHAPTER OUTLINE

Just as Semantic Web modeling in RDF is about graphs, Semantic Web modeling in the RDF Schema Language (RDFS) is about sets. Some aspects of set membership can be modeled in RDF alone, as we have seen with the `rdf:type` built-in property. But RDF itself simply creates a graph structure to represent data. RDFS provides some guidelines about how to use this graph structure in a disciplined way. It provides a way to talk about the vocabulary that will be used in an RDF graph. Which individuals are related to one another, and how? How are the properties we use to define our individuals related to other sets of individuals and, indeed, to one another? RDFS provides a way for an

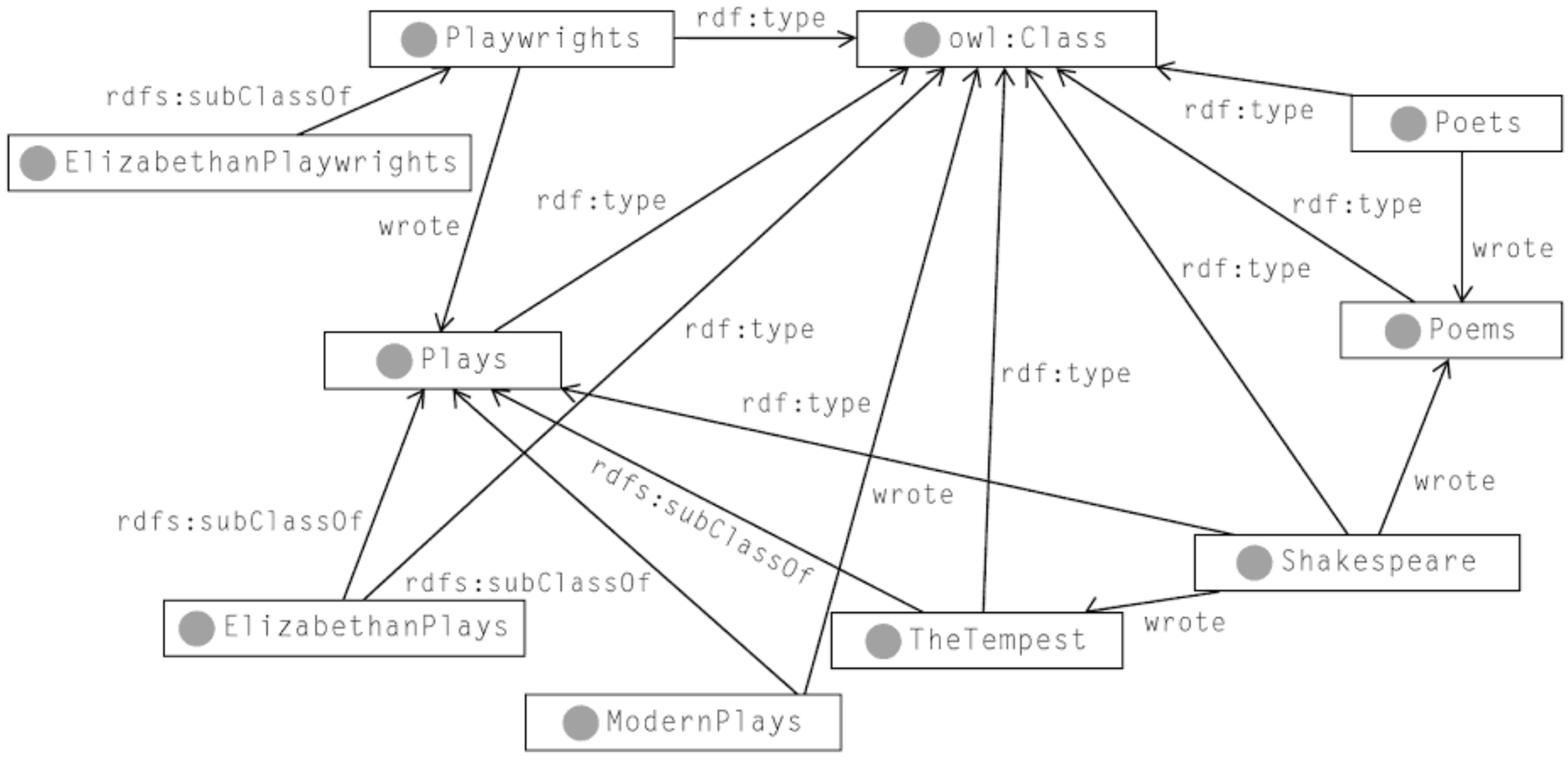


FIGURE 14.1

Sample model displaying rampant classism. Every node in this model has `rdf:type owl: Class`.

Suppose we want to go beyond mere questions and evaluate how the model organizes different points of view. It seems on the face of it that a model like this should be able to make sure that the answer to a question like "What type of thing wrote Elizabethan plays?" would at the very least include the class of playwrights, since playwrights are things that wrote plays and Elizabethan plays are plays. Can this model support this condition? Let's look at the relevant triples and see what inferences can be drawn:

```
:Playwrights a owl:Class;
   :wrote :Plays.
:ElizabethanPlays rdfs:subClassOf :Plays.
```

None of the inference patterns we have learned for OWL or RDFS applies here. In particular, there is *no* inference of the form

```
:Playwrights :wrote :ElizabethanPlays.
```

Another test criterion that this model might be expected to pass is whether it can distinguish between plays and types of plays. We do have some plays and types of plays in this model: *The Tempest* is a play, and Elizabethan play and modern play are types of plays. The model cannot distinguish between these two cases. Any query that returns *The Tempest* (as a play) will also return modern plays. Any query that returns Elizabethan play (as a type of play) will also return *The Tempest*. The model has not made enough distinctions to be responsive to this criterion.

If we think about these statements in terms of the interpretation of classes as sets, none of these results should come as a surprise. In this model, playwrights and plays are sets. The statement