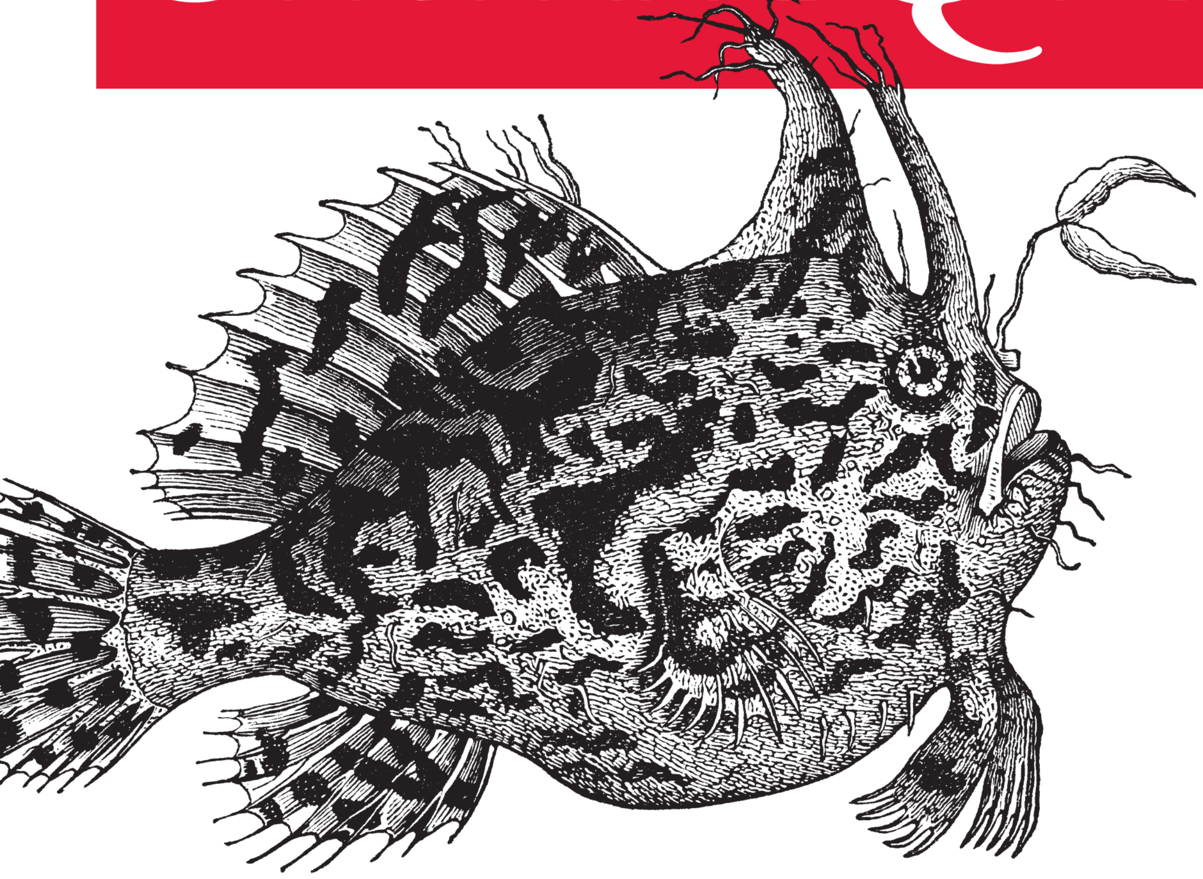
Querying and Updating with SPARQL 1.1

Learning SPARQL



O'REILLY®

Bob DuCharme

Learning SPARQL

Get hands-on experience with SPARQL, the RDF query language that's become a key component of the semantic web. With this concise book, you will learn how to use the latest version of this W3C standard to retrieve and manipulate the increasing amount of public and private data available via SPARQL endpoints. Several open source and commercial tools already support SPARQL, and this introduction gets you started right away.

Begin by learning how to write and run simple SPARQL 1.1 queries, then dive into the language's powerful features and capabilities for manipulating the data you retrieve. Learn what you need to know to add to, update, and delete data in RDF datasets, and give web applications access to this data.

- Understand SPARQL's connection with RDF, the semantic web, and related specifications
- Query and combine data from local and remote sources
- Copy, convert, and create new RDF data
- Learn how datatype metadata, standardized functions, and extension functions contribute to your queries
- Incorporate SPARQL queries into web-based applications

"It's excellent—very well organized and written, a completely painless read. I not only feel like I understand SPARQL now, but I have a much better idea why RDF is useful (I was a little skeptical before!)."

—Priscilla Walmsley
Consultant and Managing Director, Datypic, Inc.

US $29.99 CAN $34.99
ISBN: 978-1-449-30659-5

52999
9 781449 306595

Learning SPARQL

Learning SPARQL

Querying and Updating with SPARQL 1.1

Bob DuCharme

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Learning SPARQL
by Bob DuCharme

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Simon St. Laurent
Production Editor: Jasmine Perez
Proofreader: Jasmine Perez
Indexer: Bob DuCharme
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:

July 2011:	First Edition.

Portions of Chapter 7 were first published by IBM developerWorks in the article "Build Wikipedia query forms with semantic technology."

ISBN: 978-1-449-30659-5

[LSI]

1310137130

For my mom and dad, Linda and Bob Sr., who always supported any ambitious projects I attempted, even when I left college because my bandmates and I thought we were going to become big stars. (We didn't.)

Table of Contents

Preface

It is hardly surprising that the science they turned to for an explanation of things was divination, the science that revealed connections between words and things, proper names and the deductions that could be drawn from them ...

—Henri-Jean Martin,
The History and Power of Writing

Why Learn SPARQL?

More and more people are using the query language SPARQL (pronounced "sparkle") to pull data from a growing collection of public and private data. Whether this data is part of a semantic web project or an integration of two inventory databases on different platforms behind the same firewall, SPARQL is making it easier to access it. In the words of W3C Director and Web inventor Tim Berners-Lee, "Trying to use the Semantic Web without SPARQL is like trying to use a relational database without SQL."

SPARQL was not designed to query relational data, but to query data conforming to the RDF data model. RDF-based data formats have not yet achieved the mainstream status that XML and relational databases have, but an increasing number of IT professionals are discovering that tools using the RDF data model let them expose diverse sets of data (including, as we'll see, relational databases) with a common, standardized interface. Both open source and commercial software have become available with SPARQL support, so you don't need to learn new programming language APIs to take advantage of these data sources. This data and tool availability has led to SPARQL letting people access a wide variety of public data and providing easier integration of data silos within an enterprise.

Although this book's table of contents, glossary, and index let it serve as a reference guide when you want to look up the syntax of common SPARQL tasks, it's not a *complete* reference guide—if it covered every corner case that might happen when you use strange combinations of different keywords, it would be a much longer book.

Instead, the book's primary goal is to quickly get you comfortable using SPARQL to retrieve and update data and to make the best use of the retrieved data. Once you can do this, you can take advantage of the extensive choice of tools and application libraries that use this query language to retrieve, update, and mix and match the huge amount of RDF-accessible data out there.

1.1 Alert

The W3C made SPARQL 1.0 a Recommendation, or official standard, in January 2008. The language and implementations matured quickly, and as of this writing, SPARQL 1.1 (along with updated implementations) is nearly ready. This version adds new features to SPARQL, such as new functions to call, greater control over variables, and the ability to update data. This book's discussions of new 1.1 features will be highlighted with "1.1 Alert" boxes like this. The free software described in this book lets you try all of the new features.

Organization of This Book

Chapter 1, *Jumping Right In: Some Data and Some Queries*
: Writing and running a few simple queries before getting into more detail on the background and use of SPARQL.

Chapter 2, *The Semantic Web, RDF, and Linked Data (and SPARQL)*
: The bigger picture: the semantic web, related specifications, and what SPARQL adds to and gets out of them.

Chapter 3, *SPARQL Queries: A Deeper Dive*
: Building on Chapter 1, a broader introduction to the query language.

Chapter 4, *Copying, Creating, and Converting Data (and Finding Bad Data)*
: Using SPARQL to copy data from a dataset, to create new data, and to find bad data.

Chapter 5, *Datatypes and Functions*
: How datatype metadata, standardized functions, and extension functions can contribute to your queries.

Chapter 6, *Updating Data with SPARQL*
: Using SPARQL's update facility to add to and change data in a dataset instead of just retrieving it.

Chapter 7, *Building Applications with SPARQL: A Brief Tour*
: How you can incorporate SPARQL queries into web-based applications.

Glossary
: A glossary of terms and acronyms used when discussing SPARQL and the semantic web.

You'll also find an index at the back of the book to help you quickly locate explanations for SPARQL and RDF keywords and concepts.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Documentation Conventions

Variables and prefixed names are written in a monospace font `like this`. (If you don't know what prefixed names are, you'll learn in Chapter 2.) Sample data, queries, code, and markup are also shown in this font. Sometimes these include bolded text to highlight important parts that the surrounding discussion refers to, like the quoted string in the following:

```
# filename: ex001.rq

PREFIX d: <http://learningsparql.com/ns/demo#>
SELECT ?person
WHERE
{ ?person d:homeTel "(229) 276-5135" . }
```

When including punctuation at end of a quoted phrase, I put it inside the quotation marks in the American publishing style, "like this," unless the quoted string represents a specific value that would be changed if it included the punctuation. For example, if your password on a system is "swordfish", I don't want you to think that the comma is part of the password.

The following icons alert you to details that are worth a little extra attention:

An important point that might be easy to miss or a tip that can make your development or your queries more efficient.

A tip that can make your development or your queries more efficient.

A warning about a common problem or an easy trap to fall into.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning SPARQL* by Bob DuCharme (O'Reilly). Copyright 2011 Bob DuCharme, 978-1-449-30659-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

You'll also find a zip file of all of this book's sample code and data files at *http://www.learningsparql.com*, along with links to free SPARQL software and other resources.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/0636920020547

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

For their excellent contributions to the great improvements made to this book in the last two months, I'd like to thank the book's technical reviewers (Dean Allemang, Andy Seaborne, and Paul Gearon) and sample audience reviewers (Priscilla Walmsley, Eric Rochester, Peter DuCharme, and David Germano).

For helping me to get to know SPARQL well, I'd like to thank my colleagues at TopQuadrant: Irene Polikoff, Robert Coyne, Ralph Hodgson, Jeremy Carroll, Holger Knublauch, Scott Henninger, and the aforementioned Dean Allemang.

I'd also like to thank Dave Reynolds and Lee Feigenbaum for straightening out some of the knottier parts of SPARQL for me, and O'Reilly's Simon St. Laurent, Sarah Schneider, Sanders Kleinfeld, and Jasmine Perez for helping me turn this into an actual book.

Mostly, I'd like to thank my wife Jennifer and my daughters Madeline and Alice for putting up with me as I researched and wrote and tested and rewrote and rewrote this.

CHAPTER 1

Jumping Right In: Some Data and Some Queries

Chapter 2 provides some background on RDF, the semantic web, and where SPARQL fits in, but before going into that, let's start with a bit of hands-on experience writing and running SPARQL queries to keep the background part from looking too theoretical.

But first, what is SPARQL? The name is a recursive acronym for SPARQL Protocol and RDF Query Language, which is described by a set of specifications from the W3C.

The W3C, or World Wide Web Consortium, is the same standards body responsible for HTML, XML, and CSS.

As you can tell from the "RQL" part of its name, SPARQL is designed to query RDF, but you're not limited to querying data stored in one of the RDF formats. Commercial and open source utilities are available to treat relational data, XML, spreadsheets, and other formats as RDF so that you can issue SPARQL queries against these data sources —or against combinations of these sources, which is one of the most powerful aspects of the SPARQL/RDF combination.

The "Protocol" part of SPARQL's name refers to the rules for how a client program and a SPARQL processing server exchange SPARQL queries and results. These rules are specified in a separate document from the query specification document and are mostly an issue for SPARQL processor developers. You can go far with the query language without worrying about the protocol, so this book doesn't go into any detail about it.

The Data to Query

Chapter 2 describes more about RDF and all the things that people do with it, but to summarize: RDF isn't a data format, but a data model with a choice of syntaxes for storing data files. In this data model, you express facts with three-part statements known as *triples*. Each triple is like a little sentence that states a fact. We call the three parts of the triple the *subject*, *predicate*, and *object*, but you can think of them as the identifier of the thing being described (the "resource"; RDF stands for "Resource Description Format"), a property name, and a property value:

subject (resource identifier)	predicate (property name)	object (property value)
richard	homeTel	(229) 276-5135
cindy	email	cindym@gmail.com

The ex002.ttl file below has some triples expressed using the *Turtle* RDF format. (We'll learn about Turtle and other formats in Chapter 2.) This file stores address book data using triples that make statements such as "richard's homeTel value is (229) 276-5135" and "cindy's email value is cindym@gmail.com." RDF has no problem with assigning multiple values for a given property to a given resource, as you can see in this file, which shows that Craig has two email addresses:

```
# filename: ex002.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .

ab:richard ab:homeTel "(229) 276-5135" .
ab:richard ab:email   "richard49@hotmail.com" .

ab:cindy ab:homeTel "(245) 646-5488" .
ab:cindy ab:email   "cindym@gmail.com" .

ab:craig ab:homeTel "(194) 966-1505" .
ab:craig ab:email   "craigellis@yahoo.com" .
ab:craig ab:email   "c.ellis@usairwaysgroup.com" .
```

Like a sentence written in English, Turtle (and SPARQL) triples usually end with a period. The spaces you see before the periods above are not necessary, but are a common practice to make the data easier to read. As we'll see when we learn about the use of semicolons and commas to write more concise datasets, an extra space is often added before each of these as well.

Comments in Turtle data and SPARQL queries begin with the hash (#) symbol. Each query and sample data file in this book begins with a comment showing the file's name so that you can easily find it in the zip file of the book's sample data.

The first nonblank line of the data above, after the comment about the filename, is also a triple ending with a period. It tells us that the prefix “ab” will stand in for the URI *http://learningsparql.com/ns/addressbook#*, just as an XML document might tell us with the attribute setting `xmlns:ab="http://learningsparql.com/ns/addressbook#"`. An RDF triple’s subject and predicate must each belong to a particular namespace in order to prevent confusion between similar names if we ever combine this data with other data, so we represent them with URIs. Prefixes save you the trouble of writing out the full namespace URIs over and over.

A URI is a Uniform Resource Identifier. URLs (Uniform Resource Locators), also known as web addresses, are one kind of URI. A locator helps you find something, like a web page (for example, *http://www.learningsparql.com/resources/index.html*), and an identifier identifies something. So, for example, the unique identifier for Richard in my address book database is *http://learningsparql.com/ns/addressbook#richard*. A URI may look like a URL, and there may actually be a web page at that address, but there might not be; its primary job is to provide a unique name for something, not to tell you about a web page where you can send your browser.

Querying the Data

A SPARQL query typically says “I want these pieces of information from the subset of the data that meets these conditions.” You describe the conditions with *triple patterns*, which are similar to RDF triples but may include variables to add flexibility in how they match against the data. Our first queries will have simple triple patterns, and we’ll build from there to more complex ones.

The ex003.rq file below has our first SPARQL query, which we’ll run against the ex002.ttl address book data shown above.

The SPARQL Query Language specification recommends that files storing SPARQL queries have an extension of .rq, in lowercase.

The following query has a single triple pattern, shown in bold, to indicate the subset of the data we want. This triple pattern ends with a period, like a Turtle triple, and has a subject of `ab:craig`, a predicate of `ab:email`, and a variable in the object position.

A variable is like a powerful wildcard. In addition to telling the query engine that triples with any value at all in that position are OK to match this triple pattern, the values that show up there get stored in the `?craigEmail` variable so that we can use them elsewhere in the query:

```
# filename: ex003.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
```

```
SELECT ?craigEmail
WHERE
{ ab:craig ab:email ?craigEmail . }
```

This particular query is doing this to ask for any `ab:email` values associated with the resource `ab:craig`. In plain English, it's asking for any email addresses associated with Craig.

Spelling SPARQL query keywords such as PREFIX, SELECT, and WHERE in uppercase is only a convention. You may spell them in lower- or mixed case.

In a set of data triples or query triple patterns, the period after the last one is optional, so the single triple pattern above doesn't really need it. Including it is a good habit, though, because adding new triple patterns after it will be simpler. In this book's examples, you will occasionally see a single triple pattern between curly braces with no period at the end.

As illustrated in Figure 1-1, a SPARQL query's WHERE clause says "pull this data out of the dataset," and the SELECT part names which parts of that pulled data you actually want to see.

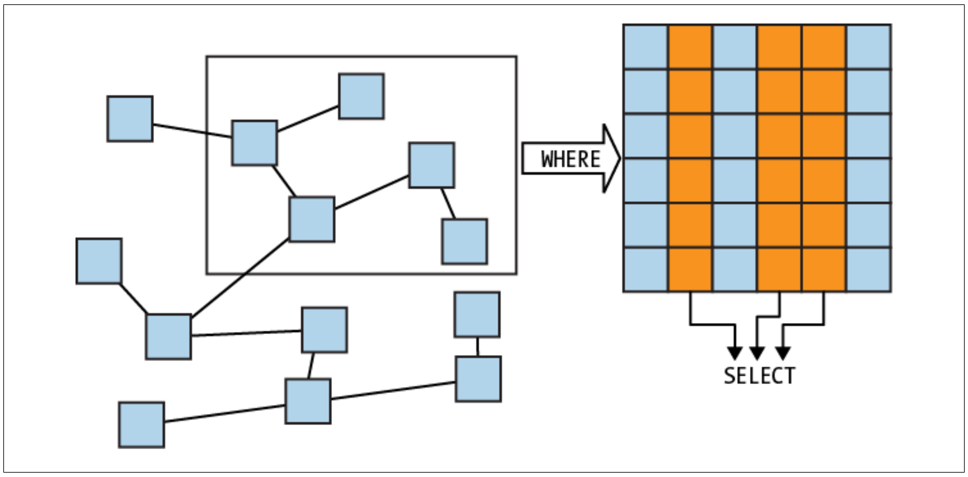


Figure 1-1. WHERE specifies data to pull out; SELECT picks which data to display

What information does the query above select from the triples that match its single triple pattern? Anything that got assigned to the `?craigEmail` variable.

As with any programming or query language, a variable name should give a clue about the variable's purpose. Instead of calling this variable `?craigEmail`, I could have called it `?zxzwzyx`, but that would make it more difficult for human readers to understand the query.

A variety of SPARQL processors are available for running queries against data both locally and remotely. (You will hear the terms *SPARQL processor* and *SPARQL engine*, but they mean the same thing: a program that can apply a SPARQL query against a set of data and let you know the result.) For queries against a data file on your own hard disk, the free, Java-based program ARQ makes it pretty simple. (You can download ARQ from *http://jena.sourceforge.net/ARQ/*.)

ARQ includes a batch file and a shell script that both let you run the ex003.rq query against the ex002.ttl data with the following command at your shell prompt or Windows command line:

```
arq --data ex002.ttl --query ex003.rq
```

ARQ's default output format shows the name of each selected variable across the top and lines drawn around each variable's results using the hyphen, equals, and pipe symbols:

```
--------------------------------
| craigEmail                   |
================================
| "c.ellis@usairwaysgroup.com" |
| "craigellis@yahoo.com"       |
--------------------------------
```

The following revision of the ex003.rq query uses full URIs to express the subject and predicate of the query's single triple pattern instead of prefixed names. It's essentially the same query, and gets the same answer from ARQ:

```
# filename: ex006.rq

SELECT ?craigEmail
WHERE
{
  <http://learningsparql.com/ns/addressbook#craig>
  <http://learningsparql.com/ns/addressbook#email>
  ?craigEmail .
}
```

The differences between this query and the first one demonstrate two things:

- You don't need to use prefixes in your query, but they can make the query more compact and easier to read than one using full URIs. When you do use a full URI, enclose it in angle brackets to show the processor that it's a URI.
- White space doesn't affect SPARQL syntax. The new query has carriage returns between the three parts of the triple pattern, and it still works just fine.

The formatting of this book's query examples follow the conventions in the SPARQL specification, which aren't particularly consistent anyway. In general, important keywords such as SELECT and WHERE go on a new line. A pair of curly braces and their contents are written on a single line if they fit there (typically, if the contents consist of a single triple pattern, like in the ex003.rq query) and are otherwise broken out with each curly brace on its own line, like in example ex006.rq.

The ARQ command above specified the data to query on the command line. SPARQL's FROM keyword lets you specify the dataset to query as part of the query itself. If you omitted the `--data ex002.ttl` parameter shown in that ARQ command line and used this next query, you'd get the same result, because the FROM keyword names the ex002.ttl data source right in the query:

```
# filename: ex007.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?craigEmail FROM <ex002.ttl>
WHERE
{ ab:craig ab:email ?craigEmail . }
```

(The angle brackets around "ex002.ttl" tell the SPARQL processor to treat it as a URI. Because it's just a filename and not a full URI, ARQ assumes that it's a file in the same directory as the query itself.)

If you specify one dataset to query in the query itself with the FROM keyword and another when you actually call the SPARQL processor (or, as the SPARQL query specification says, "in a SPARQL protocol request"), the one specified in the protocol request overrides the one specified in the query.

The queries we've seen so far had a variable in the triple pattern's object position (the third position) but you can put them in any or all of the three positions. For example, let's say someone called my phone from the number (229) 276-5135 and I didn't answer. I want to know who tried to call me, so I create the following query for my address book database, putting a variable in the subject position instead of the object position:

```
# filename: ex008.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?person
WHERE
{ ?person ab:homeTel "(229) 276-5135" . }
```

When I have ARQ run this query against the ex002.ttl address book data, it gives me this response:

```
--------------
| person     |
==============
| ab:richard |
--------------
```

Triple patterns in queries often have more than one variable. For example, I could list everything in my address book about Cindy with the following query, which has a `?propertyName` variable in the predicate position of its one triple pattern:

```
# filename: ex010.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?propertyName ?propertyValue
WHERE
{ ab:cindy ?propertyName ?propertyValue . }
```

The query's SELECT clause asks for values of the `?propertyName` and `?propertyValue` variables, and ARQ shows them as a table with a column for each one:

```
-------------------------------------
| propertyName | propertyValue      |
=====================================
| ab:email     | "cindym@gmail.com" |
| ab:homeTel   | "(245) 646-5488"   |
-------------------------------------
```

Out of habit from writing relational database queries, experienced SQL users might put commas between variable names in the SELECT part of their SPARQL queries, but this will cause an error.

More Realistic Data and Matching on Multiple Triples

In most RDF data, the subjects of the triples won't be names that are so understandable to the human eye, like the ex002.ttl dataset's `ab:richard` and `ab:cindy` resource names. They're more likely to be identifiers assigned by some process, similar to the values a relational database assigns to a table's unique ID field. Instead of storing someone's name as part of the subject URI, as our first set of sample data did, more typical RDF triples would have subject values that make no human-readable sense outside of their important role as unique identifiers. First and last name values would then be stored using separate triples, just like the `homeTel` and `email` values were stored in the sample dataset.

Another unrealistic detail of ex002.ttl is the way that resource identifiers like `ab:richard` and property names like `ab:homeTel` come from the same namespace—in this case, the *http://learningsparql.com/ns/addressbook#* namespace that the `ab:` prefix represents. A vocabulary of property names typically has its own namespace to make it easier to use it with other sets of data.

In semantic web development, a *vocabulary* is a set of terms stored using a standard format that people can reuse.

When we revise the sample data to use realistic resource identifiers, to store first and last names as property values, and to put the data values in their own separate *http://learningsparql.com/ns/data* namespace, we get this set of sample data:

```
# filename: ex012.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName "Richard" .
d:i0432 ab:lastName  "Mutt" .
d:i0432 ab:homeTel   "(229) 276-5135" .
d:i0432 ab:email     "richard49@hotmail.com" .

d:i9771 ab:firstName "Cindy" .
d:i9771 ab:lastName  "Marshall" .
d:i9771 ab:homeTel   "(245) 646-5488" .
d:i9771 ab:email     "cindym@gmail.com" .

d:i8301 ab:firstName "Craig" .
d:i8301 ab:lastName  "Ellis" .
d:i8301 ab:email     "craigellis@yahoo.com" .
d:i8301 ab:email     "c.ellis@usairwaysgroup.com" .
```

The query to find Craig's email addresses would then look like this:

```
# filename: ex013.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?craigEmail
WHERE
{
  ?person ab:firstName "Craig" .
  ?person ab:email ?craigEmail .
}
```

Although the query uses a `?person` variable, this variable isn't in the list of variables to SELECT (a list of just one variable, `?craigEmail`, in this query) because we're not interested in the `?person` variable's value. We're just using it to tie together the two triple patterns in the WHERE clause. If the SPARQL processor finds a triple with a predicate of `ab:firstName` and an object of "Craig", it will assign (or *bind*) the URI in the subject of that triple to the variable `?person`. Then, wherever else `?person` appears in the query, it will look for triples that have that URI there.

Let's say that our SPARQL processor has looked through our address book dataset triples and found a match for that first triple pattern in the query: the triple `ab:i8301 ab:firstName "Craig"`. It will bind the value `ab:i8301` to the `?person` variable, because `?person` is in the subject position of that first triple pattern, just as `ab:i8301` is in the subject position of the triple that the processor found in the dataset to match this triple pattern.

For queries like ex013.rq that have more than one triple pattern, once a query processor has found a match for one triple pattern, it moves on to the other triple patterns to see if they also have matches, but only if it can find a set of triples that match the set of triple patterns as a unit. This query's one remaining triple pattern has the `?person` and `?craigEmail` variables in the subject and object positions, but the processor won't go looking for a triple with any old value in the subject, because the `?person` variable already has `ab:i8301` bound to it. So, it looks for a triple with that as the subject, a predicate of `ab:email`, and any value in the object position, because this triple pattern introduces a new variable there: `?craigEmail`. If the processor finds a triple that fits this pattern, it will bind that triple's object to the `?craigEmail` variable… which is what the SELECT clause of this query is asking for.

As it turns out, two triples in ex012.ttl have `ab:i8301` as a subject and `ab:email` as a predicate, so the query returns two `?craigEmail` values: "craigellis@yahoo.com" and "c.ellis@usairwaysgroup.com".

```
--------------------------------
| craigEmail                   |
================================
| "c.ellis@usairwaysgroup.com" |
| "craigellis@yahoo.com"       |
--------------------------------
```

A set of triple patterns between curly braces in a SPARQL query is known as a *graph pattern*. "Graph" is the technical term for a set of RDF triples. While there are utilities to turn an RDF graph into a picture, it doesn't really refer to a graph in the visual sense, but as a data structure. A graph is like a tree data structure without the hierarchy—any node can connect to any other one. In an RDF graph, nodes represent subject or object resources, and the predicates are the connections between those nodes.

The ex013.rq query used the `?person` variable in two different triple patterns to find connected triples in the data being queried. As queries get more complex, this technique of using a variable to connect up different triple patterns becomes more common. When you progress to querying data that comes from multiple sources, you'll find that this ability to find connections between triples from different sources is one of SPARQL's best features.

If your address book had more than one Craig, and you specifically wanted the email addresses of Craig Ellis, you would just add one more triple to the pattern:

```
# filename: ex015.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?craigEmail
WHERE
{
  ?person ab:firstName "Craig" .
  ?person ab:lastName  "Ellis" .
  ?person ab:email ?craigEmail .
}
```

This gives us the same answer that we saw before.

If my phone showed me that someone at "(229) 276-5135" called my phone and I used the same ex008.rq query about that number that I used before, but queried the more detailed ex012.ttl data instead, the answer would show me the subject of the triple that had `ab:homeTel` as a predicate and "(229) 276-5135" as an object, just as the query asks for:

```
---------------------------------------------
| person                                    |
=============================================
| <http://learningsparql.com/ns/data#i0432> |
---------------------------------------------
```

If I really want to know who called me, "http://learningsparql.com/ns/data#i0432" isn't a very helpful answer.

Although the ex008.rq query doesn't return a very human-readable answer from the ex012.ttl dataset, we just took a query designed around one set of data and used it with a different set that had a different structure, and we at least got a sensible answer instead of an error. This is rare among standardized query languages and one of SPARQL's great strengths: queries aren't as closely tied to specific data structures as they are with a query language like SQL.

What I want is the first and last name of the person with that phone number, so this next query asks for that:

```
# filename: ex017.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last
WHERE
{
  ?person ab:homeTel "(229) 276-5135" .
  ?person ab:firstName ?first .
  ?person ab:lastName  ?last .
}
```

ARQ responds with a more readable answer:

```
----------------------
| first     | last   |
======================
| "Richard" | "Mutt" |
----------------------
```

Revising our query to find out everything about Cindy in the ex012.ttl data is similar: we ask for all the predicates and objects (stored in the `?propertyName` and `?propertyValue` variables) associated with the subject that has an `ab:firstName` of "Cindy" and an `ab:lastName` of "Marshall":

```
# filename: ex019.rq

PREFIX a: <http://learningsparql.com/ns/addressbook#>

SELECT ?propertyName ?propertyValue
WHERE
{
  ?person a:firstName "Cindy" .
  ?person a:lastName  "Marshall" .
  ?person ?propertyName ?propertyValue .
}
```

In the response, note that the values from the ex012.ttl file's new `ab:firstName` and `ab:lastName` properties appear in the `?propertyValue` column. In other words, their values got bound to the `?propertyValue` variable, just like the `ab:email` and `ab:homeTel` values:

```
------------------------------------
| propertyName | propertyValue     |
====================================
| a:email      | "cindym@gmail.com" |
| a:homeTel    | "(245) 646-5488"   |
| a:lastName   | "Marshall"         |
| a:firstName  | "Cindy"            |
------------------------------------
```

The `a:` prefix used in the ex019.rq query was different from the `ab:` prefix used in the ex012.ttl data being queried, but `ab:firstName` in the data and `a:firstName` in this query still refer to the same thing: `http://learningsparql.com/ns/addressbook#firstName`. What matters are the URIs represented by the prefixes, not the prefixes themselves, and this query and this dataset happen to use different prefixes to represent the same namespace.

Searching for Strings

What if you want to check for a piece of data, but you don't even know what subject or property might have it? The following query only has one triple pattern, and all three parts are variables, so it's going to match every triple in the input dataset. It won't return them all, though, because it has something new called a FILTER that instructs the query processor to only pass along triples that meet a certain condition. In this FILTER, the condition is specified using `regex()`, a function that checks for strings matching a certain pattern. (We'll learn more about FILTERs in Chapter 3 and `regex()` in Chapter 5.) This particular call to `regex()` checks whether the object of each matched triple has the string "yahoo" anywhere in it:

```
# filename: ex021.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT *
WHERE
{
  ?s ?p ?o .
  FILTER (regex(?o, "yahoo","i"))
}
```

It's a common SPARQL convention to use `?s` as a variable name for a triple pattern subject, `?p` for a predicate, and `?o` for an object.

The query processor finds a single triple that has "yahoo" in its object value:

```
---------------------------------------------------------------------------------
| s                                        | p        | o                       |
=================================================================================
| <http://learningsparql.com/ns/data#i8301> | ab:email | "craigellis@yahoo.com" |
---------------------------------------------------------------------------------
```

Something else new in this query is the use of the asterisk instead of a list of specific variables in the SELECT list. This is just a shorthand way to say "SELECT all variables that get bound in this query." As you can see, the output has a column for each variable used in the WHERE clause.

This use of the asterisk in a SELECT list is handy when you're doing a few ad hoc queries to explore a dataset or trying out some ideas as you build to a more complex query.

What Could Go Wrong?

Let's modify a copy of the ex015.rq query that asked for Craig Ellis's email addresses to also ask for his home phone number. (If you review the ex012.ttl data, you'll see that Richard and Cindy have `ab:homeTel` values, but not Craig.)

```
# filename: ex023.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?craigEmail ?homeTel
WHERE
{
  ?person ab:firstName "Craig" .
  ?person ab:lastName  "Ellis" .
  ?person ab:email ?craigEmail .
  ?person ab:homeTel ?homeTel .
}
```

When I ask ARQ to apply this query to the ex012.ttl data, it gives me headers for the variables I asked for but no data underneath them:

```
------------------------
| craigEmail | homeTel |
========================
------------------------
```

Why? The query asked the SPARQL processor for the email address and phone number of anyone who met the four conditions listed, and even though resource `ab:i8301` met the first three conditions (that is, the data has triples with `ab:i8301` as a subject that matched the first three triple patterns), no resource in the data met all four conditions, because no one with an `ab:firstName` of "Craig" and an `ab:lastName` of "Ellis" had an `ab:homeTel` value set. So, the SPARQL processor didn't return any data.

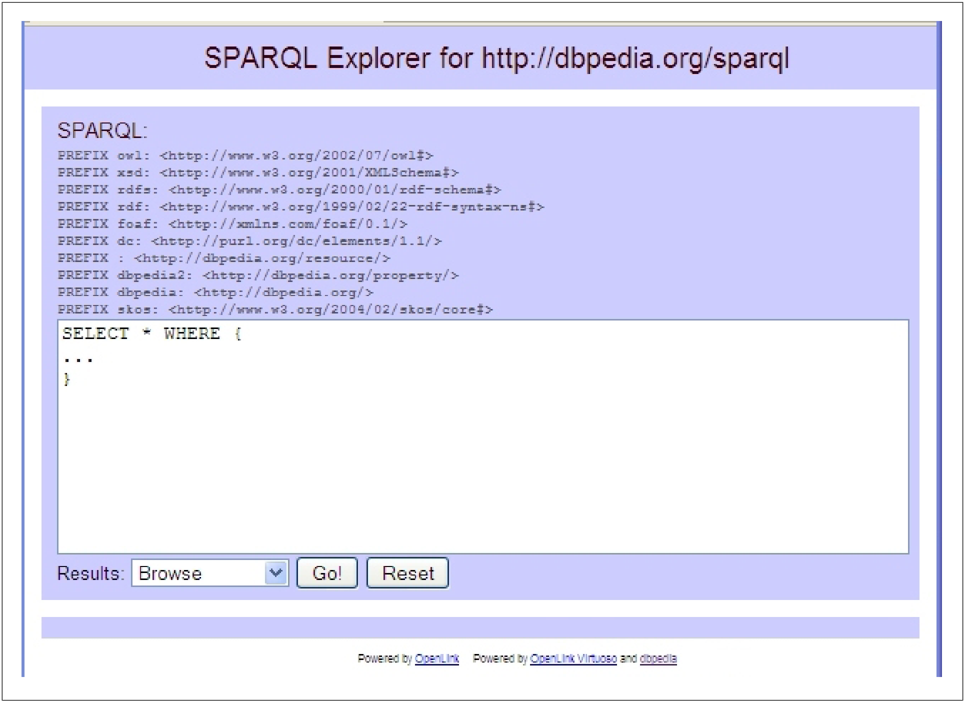
In Chapter 3, we'll learn about SPARQL's OPTIONAL keyword, which lets you make requests like "Show me the `?craigEmail` value and, if it's there, the `?homeTel` value as well."

Without the OPTIONAL keyword, a SPARQL processor will only return data for a graph pattern if it can match every single triple pattern in that graph pattern.

Querying a Public Data Source

Querying data on your own hard drive is useful, but the real fun of SPARQL starts when you query public data sources. You need no special software, because these data collections are often made publicly available through a *SPARQL endpoint*, which is a web service that accepts SPARQL queries.

The most popular SPARQL endpoint is DBpedia, a collection of data from the gray infoboxes of fielded data that you often see on the right side of Wikipedia pages. Like many SPARQL endpoints, DBpedia includes a web form where you can enter a query and then explore the results, making it very easy to explore its data. DBpedia uses a program called SNORQL to accept these queries and return the answers on a web page. If you send a browser to *http://dbpedia.org/snorql/*, you'll see a form where you can enter a query and select the format of the results you want to see, as shown in Figure 1-2. For our experiments, we'll stick with "Browse" as our result format.


SPARQL Explorer for http://dbpedia.org/sparql

SPARQL:

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT * WHERE {
...
}
```

Results: Browse

Figure 1-2. DBpedia's SNORQL web form

I want DBpedia to give me a list of albums produced by the hip-hop producer Timbaland and the artists who made those albums. If Wikipedia has a page for Some Topic at *http://en.wikipedia.org/wiki/Some_Topic*, the DBpedia URI to represent that resource is usually *http://dbpedia.org/resource/Some_Topic*, so after finding the Wikipedia page for the producer at *http://en.wikipedia.org/wiki/Timbaland*, I sent a browser to *http://dbpedia.org/resource/Timbaland*, found plenty of information (although it was redirected to *http://dbpedia.org/page/Timbaland*, because when a browser asks for the information, DBpedia redirects it to the HTML version of the data), and knew that this was the right URI to represent him in queries.

I see on the upper half of the SNORQL query in Figure 1-2 that *http://dbpedia.org/resource/* is already declared with a prefix of just ":", so I know that I can refer to the producer as `:Timbaland` in my query.

A namespace prefix can simply be a colon. This is popular for namespaces that are used often in a particular document because the reduced clutter makes it easier for human eyes to read.

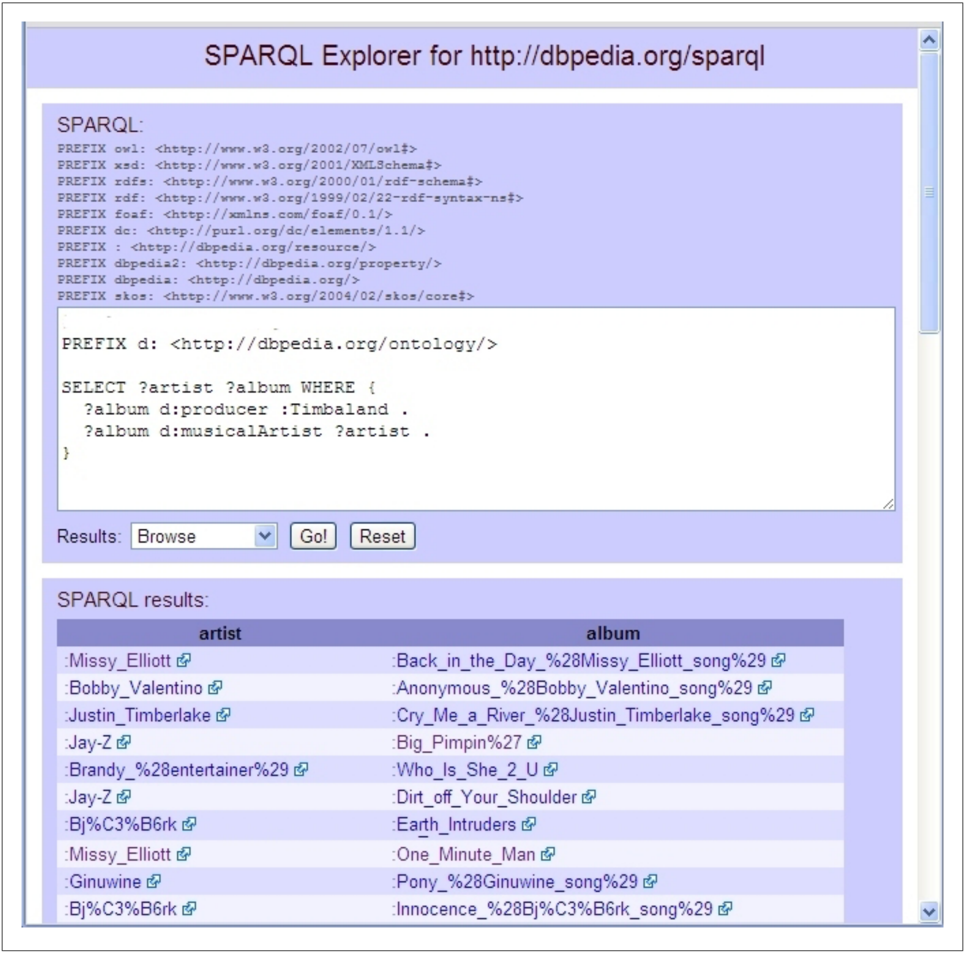
The `producer` and `musicalArtist` properties that I plan to use in my query are from the *http://dbpedia.org/ontology/* namespace, which is not declared on the SNORQL query input form, so I included a declaration for it in my query:

```
# filename: ex025.rq

PREFIX d: <http://dbpedia.org/ontology/>

SELECT ?artist ?album
WHERE
{
  ?album d:producer :Timbaland .
  ?album d:musicalArtist ?artist .
}
```

This query pulls out triples about albums produced by Timbaland and the artists listed for those albums and asks for the values that got bound to the `?artist` and `?album` variables. When I replace the default query on the SNORQL web page with this one and click the Go button, SNORQL displays the results to me underneath the query, as shown in Figure 1-3.

The scroll bar on the right shows that this list of results is only the beginning of a much longer list, and even that may not be complete—remember, Wikipedia is maintained by volunteers, and while there are some quality assurance efforts in place, they are dwarfed by the scale of the data to work with.

Also note that it didn't give us the actual names of the albums or artists, but names mixed with punctuation and various codes. Remember how `:Timbaland` in my query was an abbreviation of a full URI representing the producer? Names such as `:Bj%C3%B6rk` and `:Cry_Me_a_River_%28Justin_Timberlake_song%29` in the result are abbreviations of URIs as well. These artists and songs have their own Wikipedia pages and associated data, and the associated data includes more readable versions of the names that we can ask for in a query. We'll learn about the `rdfs:label` property that often stores these more readable labels in Chapters 2 and 3.



SPARQL Explorer for http://dbpedia.org/sparql

SPARQL:

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
```

```
PREFIX d: <http://dbpedia.org/ontology/>

SELECT ?artist ?album WHERE {
  ?album d:producer :Timbaland .
  ?album d:musicalArtist ?artist .
}
```

Results: Browse

SPARQL results:

artist	album
:Missy_Elliott	:Back_in_the_Day_%28Missy_Elliott_song%29
:Bobby_Valentino	:Anonymous_%28Bobby_Valentino_song%29
:Justin_Timberlake	:Cry_Me_a_River_%28Justin_Timberlake_song%29
:Jay-Z	:Big_Pimpin%27
:Brandy_%28entertainer%29	:Who_Is_She_2_U
:Jay-Z	:Dirt_off_Your_Shoulder
:Bj%C3%B6rk	:Earth_Intruders
:Missy_Elliott	:One_Minute_Man
:Ginuwine	:Pony_%28Ginuwine_song%29
:Bj%C3%B6rk	:Innocence_%28Bj%C3%B6rk_song%29

Figure 1-3. SNORQL displaying results of a query

Summary

In this chapter, we learned:

- What SPARQL is.
- The basics of RDF.
- The meaning and role of URIs.
- The parts of a simple SPARQL query.
- How to execute a SPARQL query with ARQ.
- How the same variable in multiple triple patterns can connect up the data in different triples.
- What can lead to a query returning nothing.
- What SPARQL endpoints are and how to query the most popular one, DBpedia.

Later chapters describe how to create more complex queries, how to modify data, how to build applications around your queries, and how it all fits into the semantic web, but if you can execute the queries shown in this chapter, you're ready to put SPARQL to work for you.

CHAPTER 2

The Semantic Web, RDF, and Linked Data (and SPARQL)

SPARQL is a query language for data that follows a particular model, but the semantic web isn't about the query language or about the model—it's about the data. The booming amount of data becoming available on the semantic web is making great new kinds of applications possible, and as a well-implemented, mature standard designed with the semantic web in mind, SPARQL is the best way to get that data and put it to work in your applications.

What Exactly Is the "Semantic Web"?

As excitement over the semantic web grows, some vendors use the phrase to sell products with strong connections to the ideas behind the semantic web, and others use it to sell products with weaker connections. This can be confusing for people trying to understand the semantic web landscape.

I like to define the semantic web as *a set of standards and best practices for sharing data and the semantics of that data over the web for use by applications*. Let's look at this definition one or two phrases at a time, and then we'll look at these issues in more detail.

A set of standards

Before Tim Berners-Lee invented the World Wide Web, more powerful hypertext systems were available, but he built his around simple specifications that he published as public standards. This made it possible for people to implement his system on their own (that is, to write their own web servers, web browsers, and especially web pages), and his system grew to become the biggest one ever. Berners-Lee founded the W3C to oversee these standards, and the semantic web is also built on W3C standards: the RDF data model, the SPARQL query language, and the RDF Schema and OWL standards for storing vocabularies and ontologies. A product or project may deal with semantics, but if it doesn't use these standards, it can't connect to and be part of the semantic web any more than a 1985 hypertext system could link to a page on the World Wide Web without using the HTML or HTTP standards. (There are those who disagree on this last point.)

best practices for sharing data over the web... for use by applications

Berners-Lee's original web was designed to deliver human-readable documents. If you want to fly from one airport to another next Sunday afternoon, you can go to an airline website, fill out a query form, and then read the query results off the screen with your eyes. Airline comparison sites have programs that retrieve web pages from multiple airline sites and extract the information they need, in a process known as "screen scraping," before using the data for their own web pages. Before writing such a program, a developer at the airline comparison website must analyze the HTML structure of each airline's website to determine where the screen scraping program should look for the data it needs. If one airline redesigns their website, the developer must update their screen scraping program to account for these differences.

Berners-Lee came up with the idea of *Linked Data* as a set of best practices for sharing data across the web infrastructure so that applications can more easily retrieve data from public sites with no need for screen scraping—for example, to let your calendar program get flight information from multiple airline websites in a common, machine-readable format. These best practices recommend the use of URIs to name things and the use of standards such as RDF and SPARQL. They provide excellent guidelines for the creation of an infrastructure for the semantic web.

and the semantics of that data

The idea of "semantics" is often defined as "the meaning of words." Linked Data principles and the related standards make it easier to share data, and the use of URIs can provide a bit of semantics by providing the context of a term. For example, even if I don't know what "sh98003588#concept" refers to, I can see from the URI *http://id.loc.gov/authorities/sh98003588#concept* that it comes from the US Library of Congress. Storing the complete meaning of words so that computers can "understand" these meanings may be asking too much of current computers, but the W3C Web Ontology Language (also known as *OWL*) already lets us store valuable bits of meaning so that we can get more out of our data. For example, when we know that the term "spouse" is symmetrical (that is, that if A is the spouse of B then B is the spouse of A), or that zip codes are a subset of postal codes, or that "sell" is the opposite of "buy," we know more about the resources that have these properties and the relationships between these resources.

Let's look at these components of the semantic web in more detail.

URLs, URIs, IRIs, and Namespaces

When Berners-Lee invented the Web, along with writing the first web server and browser, he developed specifications for three things so that all the servers and browsers could work together:

- A way to represent document structure, so that a browser would know which parts of a document were paragraphs, which were headers, which were links, and so forth. This specification is the Hypertext Markup Language, or HTML.
- A way for client programs such as web browsers and servers to communicate with each other. The Hypertext Transfer Protocol, or HTTP, consists of a few short commands and three-digit codes that essentially let a client program such as a web browser say things like "Hey www.learningsparql.com server, send me the `index.html` file from the `resources` directory!" and let the server say "OK, here you go!" or "Sorry, I don't know about that resource."
- A compact way for the client to specify which resource it wants—for example, the name of a file, the directory where it's stored, and the server that has that file system. You could call this a web address, or you could call it a resource locator. Berners-Lee called a server-directory-resource name combination that a client sends using a particular internet protocol (for example, *http://www.learningsparql.com/resources/index.html*) a Uniform Resource Locator, or URL.

When you own a domain name like learningsparql.com or redcross.org, you control the directory structure and file names used to store resources there. This ability of a domain name owner to control the naming scheme (similarly to the way that Java package names build on domain names) led developers to use these names for resources that weren't necessarily web addresses. For example, the Friend of a Friend (FOAF) vocabulary uses *http://xmlns.com/foaf/0.1/Person* to represent the concept of a person, but if you send your browser to that "address," it will just be redirected to the spec's home page.

This confused many people, because they assumed that anything that began with "http://" was the address of a web page that they could view with their browser. This confusion led two engineers from MIT and Xerox to write up a specification for Universal Resource Names, or URNs. An URN might take the form *urn:isbn:006251587X* to represent a particular book or *urn:schemas-microsoft-com:office:office* to refer to Microsoft's schema for describing the structure of Microsoft Office files.

The term Universal Resource Identifier was developed to encompass both URLs and URNs. This means that a URL is also a URI. URNs didn't really catch on, though. So, because hardly anyone uses URNs, most URIs are URLs, and that's why people sometimes use the terms interchangeably. It's still very common to refer to a web address as a URL, and it's fairly typical to refer to something like *http://xmlns.com/foaf/0.1/*

Person as a URI instead, because it's just an identifier—even though it begins with "http://".

As if this wasn't enough names for variations on URLs, the Internet Engineering Task Force released a spec for the concept of Internationalized Resource Identifiers. IRIs are URIs that allow a wider range of characters to be used in order to accommodate other writing systems. For example, an IRI can have Chinese or Cyrillic characters, and a URI can't. In general usage, "IRI" means the same thing as "URI." The SPARQL Query Language specification refers to IRIs when it talks about naming resources (or about special functions that work with those resource names), and not to URIs or URLs, because it's the broadest term.

URIs helped to solve another problem. As the XML markup language became more popular, XML developers began to combine collections of elements from different domains to create specialized documents. This led to a difficult question: what if two sets of elements for two different domains use the same name for two different things? For example, if I want to say that Tim Berners-Lee's title at the W3C is "Director" and that the title of the book he wrote is "Weaving the Web," I need to distinguish between these two senses of the word "title." Computer science has used the term "namespace" for years to refer to a set of names used for a particular purpose, so the W3C released a spec describing how XML developers could say that certain terms come from specific namespaces. This way, they could distinguish between different senses of a word like "title."

How do we name a namespace and refer to it? With a URI, of course. For example, the name for the Dublin Core standard set of basic metadata terms is the URI *http://purl.org/dc/elements/1.1/*. An XML document's main enclosing element often includes the attribute setting `xmlns:dc="http://purl.org/dc/elements/1.1/"` to indicate that the `dc` prefix will stand for the Dublin Core namespace URI in that document. Imagine that an XML processor found the following element in such a document:

```
<dc:title>Weaving the Web</dc:title>
```

It would knows that it meant "title" in the Dublin Core sense—the title of a work.

If the document's main element also declared a `v` namespace prefix with `xmlns:v="http://www.w3.org/2006/vcard/"`, an XML processor seeing the following element would know that it meant "title" in the sense of "job title," because it comes from the vCard vocabulary for specifying business card information:

```
<v:title>Director</v:title>
```

There's nothing special about the particular prefixes used. If you define `dc:` as the prefix for *http://www.w3.org/2006/vcard/* in an XML document or for a given set of triples, then a processor would understand `dc:title` as referring to a vCard title, not a Dublin Core one. This would be confusing to people reading it, so it's not a good idea, but remember: prefixes don't identify namespaces. They stand in for URIs that do.

We saw in Chapter 1 that an RDF statement has three parts. We also saw that the names of the subject and predicate parts must each belong to specific namespaces so that no person or process confuses those names with similar ones—especially if that data gets combined with other data. Like XML, RDF lets you define a prefix to represent a namespace URI so that your data doesn't get too verbose, but unlike XML, RDF lets you use full URIs with the names instead of prefixes. After declaring that the prefix `v:` refers to the namespace *http://www.w3.org/2006/vcard/*, an RDF dataset could say that Berners-Lee has a `v:title` of "Director", but it could also say that he has a `<http://www.w3.org/2006/vcard/title>` of "Director", using the entire namespace URI instead of the prefix.

RDF-related syntaxes such as Turtle, N3, and SPARQL use the `<>` brackets to tell a processor that something is an actual URI and not just some string of characters that begins with "http://".

A prefixed name is sometimes called a qualified name, or *qname*. Because "qname" is actually an XML term whose meaning is slightly different, "prefixed name" is the more correct term when discussing RDF resources.

Just about anywhere in RDF and SPARQL where you can use a URI, you can use a prefixed name instead, as long as its prefix has been properly declared. We refer to the part of the name after the colon as the *local name*; it's the name used from the prefix's namespace. For example, in `dc:title`, the local name is `title`.

To summarize, people sometimes use the terms URI and URL interchangeably, but in RDF it's URIs that matter, because we want to identify resources and information about those resources. A URI usually looks like a URL, and there may even be a web page at that address, but there might not be; the URI's primary job is to provide a unique name for a resource or property, not to tell you about a web page where you can send your browser. Technical discussions may use the term "IRI," but it's a variation on "URI."

A SPARQL query's WHERE clause describes the data to pull out of a dataset, and unambiguous identifiers are crucial for this. The URIs in some RDF triples may not be the parts that your query's SELECT clause chooses to show in the results, but they're necessary to identify which data to retrieve and how to cross-reference different bits of data with each other to connect them up. This means that a good understanding of the role of URIs gives you greater control over your queries.

The URIs that identify RDF resources are like the unique ID fields of relational database tables, except that they're universally unique, which lets you link data from different sources around the world instead of just linking data from different tables in the same database.

The Resource Description Format (RDF)

In Chapter 1, we learned the following about the Resource Description Format:

- It's a data model in which the basic unit of information is known as a triple.
- A triple consists of a subject, a predicate, and an object. You can also think of these as a resource identifier, an attribute or property name, and an attribute or property value.
- To remove any ambiguity from the information stated by a given triple, the triple's subject and predicate must be URIs. (We've since learned that we can use prefixed names in place of URIs.)

In this section, we'll learn more about different ways to store and use RDF and how subjects and objects can be more than URIs representing specific resources or simple strings.

Storing RDF in Files

The technical term for saving RDF as a string of bytes that can be saved on a disk is *serialization*. We use this term instead of "files" because there have been operating systems that didn't use the term "file" for a named collection of saved data, but in practice, all RDF serializations so far have been text files with different syntaxes used to represent the triples. (Although they may be files sitting on disks, they may also be generated dynamically.) The serialization format you'll see most often, especially in this book, is called Turtle. Older ones may come up as well, and they provide some historical context for Turtle.

Most RDF tools can read and write all of the formats described in this section, and many tools are available to convert between them. A query tool like ARQ, which lets you query data sitting on a disk file, has an RDF parser built in to read that data and then hand it to the SPARQL query engine. It's the parser's job to worry about the serialization (or serializations) that the data uses, not yours. You may need to tell ARQ the dataset's serialization format, but these processors can usually guess from the file extension.

This section gives you some background about the kinds of things you might see in a file of RDF data. There's no need to learn all the details, but sometimes it's handy to know which serialization is which. We'll look at how several formats represent the following three facts:

- The book with ISBN 006251587X has the creator Tim Berners-Lee.
- The book with ISBN 006251587X has the title "Weaving the Web".
- Tim Berners-Lee's title is "Director".

The examples use the URI *http://www.w3.org/People/Berners-Lee/card#i* to represent Berners-Lee, because that's the URI he uses to represent himself in his FOAF file. The examples use the URN *urn:isbn:006251587X* to represent the book.

A FOAF file is an RDF collection of facts about a person such as where they work, where they went to school, and who their friends are. The FOAF project's original goal was to provide the foundation for a distributed, RDF-based social networking system (and it's actually three years older than Facebook) but its vocabulary identifies such basic facts about people that it gets used for much more than FOAF files.

The simplest format is called *N-Triples*. (It's actually a subset of another serialization called N3 that we'll come to shortly.) In N-Triples, you write out complete URIs inside of angle brackets and strings inside of quotation marks. Each triple is on its own line with a period at the end.

For example, we can represent the three facts above like this in N-Triples:

```
# The hash symbol is the comment delimiter in N-Triples.
# filename: ex028.nt

<urn:isbn:006251587X> <http://purl.org/dc/elements/1.1/creator>
  <http://www.w3.org/People/Berners-Lee/card#i> .

<urn:isbn:006251587X> <http://purl.org/dc/elements/1.1/title> "Weaving the Web" .

<http://www.w3.org/People/Berners-Lee/card#i> <http://www.w3.org/2006/vcard/title>
  "Director" .
```

(The first and third triples here are actually not legal N-Triples triples, because I had to insert line breaks to fit them on this page, but the ex028.nt file included with the book's sample code has each triple on its own line.)

Like the rows of an SQL table, the order of a set of triples does not matter. If you moved the third triple in the N-Triples example or any RDF serialization example in this section to be first, the set of information would be considered exactly the same.

The simplicity of N-Triples makes it popular for teaching people about RDF, and some parsers can read it more quickly because they have less work to do, but the format's verbosity makes it less popular than the other formats.

The oldest RDF serialization, *RDF/XML*, was part of the original RDF specification in 1999. Before we look at some examples of RDF/XML, keep in mind that I'm only showing them so that you'll recognize the general outline of an RDF/XML file when you see one. The details are something for an RDF parser, and not you, to worry about, and once Turtle becomes a W3C standard, we'll see less and less use of RDF/XML. As of this writing, though, it's still the only standardized RDF serialization format.

Here are the three facts above in RDF/XML:

```
<!-- Being XML, RDF/XML uses regular XML comment delimiters. -->
<!-- filename: ex029.rdf -->

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:v="http://www.w3.org/2006/vcard/">

  <rdf:Description rdf:about="urn:isbn:006251587X">
    <dc:title>Weaving the Web</dc:title>
    <dc:creator rdf:resource="http://www.w3.org/People/Berners-Lee/card#i"/>
  </rdf:Description>

  <rdf:Description rdf:about="http://www.w3.org/People/Berners-Lee/card#i">
    <v:title>Director</v:title>
  </rdf:Description>

</rdf:RDF>
```

There are a few things to note about this:

- The element containing all the triples must be an `RDF` element from the *http://www.w3.org/1999/02/22-rdf-syntax-ns#* namespace.
- The subject of each triple is identified in the `rdf:about` attribute of a `rdf:Description` element.
- The example could have had a separate `rdf:Description` element for each triple, but it expressed two triples about the resource *urn:isbn:006251587X* by putting two child elements inside the same `rdf:Description` element—a `dc:title` element and a `dc:creator` element.
- The objects of the `dc:title` and `v:title` triples are expressed as plain text (or, in XML terms, as PCDATA) between the start- and end-tags. To show that the `dc:creator` value is a resource and not a string, it's in an `rdf:resource` attribute of the `dc:creator` element.

The following demonstrates some other ways to express the exact same information in RDF/XML that we see above:

```
<!-- filename: ex030.rdf -->

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:v="http://www.w3.org/2006/vcard/">

  <rdf:Description rdf:about="urn:isbn:006251587X" dc:title="Weaving the Web">
    <dc:creator>
      <rdf:Description rdf:about="http://www.w3.org/People/Berners-Lee/card#i">
        <v:title>Director</v:title>
      </rdf:Description>
    </dc:creator>
  </rdf:Description>

</rdf:RDF>
```

In ex030.rdf, the `dc:title` value is an attribute value, and not a child element, of the *urn:isbn:006251587X* resource's `rdf:Description` element. An even bigger change is that the *urn:isbn:006251587X* resource's `dc:creator` object value is not expressed as an `rdf:resource` attribute value, but as another `rdf:Description` element inside the `dc:creator` element. We can do this because *http://www.w3.org/People/Berners-Lee/card#i* is the object of one triple and the subject of another.

In practice, this nesting of elements in RDF/XML known as *striping* is usually more trouble than it's worth when you consider that an RDF processor will find the same triples in a simpler representation.

RDF/XML never became popular with XML people because of the potential complexity and the difficulty of processing it—for example, if a piece of information such as the `dc:title` value above may appear as either a child element or an attribute value of the `rdf:Description` element, XSLT stylesheets and other tools for processing this information have a lot of extra things to check for to process this small collection of information.

A big driver for XML's early success was that developers had seen many different data formats in many different syntaxes, each requiring a new parser. As a nonproprietary W3C standard way to represent a broad variety of information, XML seemed like a logical approach for serializing RDF when RDF was a new W3C standard. RDF/XML never became very popular, though, for several reasons. One was complications arising from its bad fit with XML document types that included lots of narrative content and inline elements (the kind of documents that XML was designed for); another was limitations imposed by XML element naming rules on URI local names. Yet another was the difficulty described above of processing it with popular XML tools. There are plenty more reasons.

Another serialization format is *N3*, which is short for "Notation 3." This was a personal project by Tim Berners-Lee ("with his director hat off") that he described as "basically equivalent to RDF in its XML syntax, but easier to scribble when getting started." It combines the simplicity of N-Triples with RDF/XML's ability to abbreviate long URIs with prefixes, and it adds a lot more. We can represent the three facts above like this in N3:

```
# The hash symbol is the comment delimiter in n3.
# filename: ex031.n3

@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix v:  <http://www.w3.org/2006/vcard/> .

<http://www.w3.org/People/Berners-Lee/card#i>
```

```
      v:title "Director" .

<urn:isbn:006251587X>
      dc:creator <http://www.w3.org/People/Berners-Lee/card#i> ;
      dc:title "Weaving the Web" .
```

It looks very similar to N-Triples, except that extra white space is allowed for nicer formatting, and you can use namespace prefixes to make names shorter. It too must declare the prefixes first; note that these declarations are also expressed as triples, complete with periods at the end.

Like RDF/XML, N3 offers some shortcuts for describing multiple facts about the same subject. The example above only shows the identifier `<urn:isbn:006251587X>` once, and after the `<http://www.w3.org/People/Berners-Lee/card#i>` object of the first triple about this book, you see a semicolon. This means that another predicate and object are coming for the same subject. You can look at this new triple as another related idea expressed as part of the same "sentence," just like in written English. The final object has a period after it to show that the sentence is really finished. So, you could say that the last three lines of ex031.n3 tell us "resource urn:isbn:006251587X has a dc:creator value of http://www.w3.org/People/Berners-Lee/card#i; also, it has a dc:title value of 'Weaving the Web'."

A comma in N3 means "the next triple has the same subject and predicate as the last one, but the following new object." For example, the following lists two `dc:creator` values for the book with an ISBN value of 0123735564:

```
#filename: ex032.n3

@prefix dc: <http://purl.org/dc/elements/1.1/> .

<urn:isbn:0123735564> dc:creator
  <http://www.topquadrant.com/people/dallemang/foaf.rdf#me> ,
  <http://www.cs.umd.edu/~hendler/2003/foaf.rdf#jhendler> .
```

N3 has other interesting features, such as the ability to refer to a graph of triples as a resource in and of itself so that you could say "this graph of triples has a `dc:creator` value of 'Jane Smith'." You can also specify rules, which let you infer new triples based on true or false conditions in an existing set of triples—for example, to infer that if Bridget's father is Peter and Peter's father is Henry, then Bridget's grandfather is Henry. Inferencing often plays an important role in semantic web applications.

N3 never became a standard, and no one really uses these extra features because they inspired separate work at the W3C that did become standardized. (Later in this book, we'll see how to refer to graphs and do inferencing with SPARQL.)

If you use N3 without these extra features, then you are already using our next serialization format: *Turtle*. There's no need to show examples here, because any software that understands Turtle will understand the two N3 examples above, which don't use the extra features. Despite its name, Turtle is moving the quickest among RDF serialization formats in the race for popularity, and the W3C plans to standardize it.

For the rest of this book, unless otherwise specified, all data samples will be shown using Turtle syntax.

Another increasingly popular way to store triples is the W3C standard *RDFa*. This lets you store subjects, predicates, and objects in an XML document that wasn't designed to accommodate RDF, as well as in HTML documents.

The "a" in "RDFa" stands for "attributes." RDFa defines a few new attributes and specifies several existing HTML ones as places to store or identify subjects, predicates, and objects mixed in with the data in that XML or HTML document. RDFa's supplemental role in XML and HTML documents makes it excellent for metadata about content in those documents, and utilities are available to pull the triples out of RDFa attributes in a format that lets you query them with SPARQL.

RDFa's ability to embed triples in HTML makes it great for sharing machine-readable data in web pages so that automated processes gathering that data don't need to do screen scraping of those pages.

Storing RDF in Databases

If you need to store a very large number of triples, keeping them as Turtle or RDF/XML in one big text file may not be your best option, because a system that indexes data and decides which data to load into memory when—that is, a database management system—can be more efficient. There are ways to store RDF in a relational database manager such as MySQL or Oracle, but the best way is a database manager optimized for RDF triples. We call this a *triplestore*, and both commercial and open source triplestores are available.

When evaluating a triplestore, along with typical database manager issues such as size, speed, platform availability, and cost, there are several SPARQL-related issues to consider:

- Does it support real SPARQL, or some "SPARQL-like" query and update language that the triplestore's developers made up themselves?
- How easy is it to give the triplestore a SPARQL query and to then get the result back, both interactively and programatically?
- Can you create your own SPARQL endpoint with the triplestore?
- Does the triplestore support the latest SPARQL standard?

1.1 Alert

SPARQL 1.1's UPDATE facility is fun to play with on small files of data, but you'll see its real value when you use a triplestore. The ability to add, delete, and change data is important with any database management program, and when using a triplestore you want to do this using a recognized standard and not something that only works with that particular triplestore.

Data Typing

So far, we've seen that a triple's subject and predicate must be URIs and that its object can be a URI or a string. The object can actually be more than a simple string, because you can assign it a specific datatype or a tag that identifies the text as being in a particular language such as Canadian French or Brazilian Portuguese.

The technical term for these non-URI values is *literals*. A typed literal has a datatype assigned to it, usually from the selection offered by the W3C's XML Schema Part 2 specification. When a program knows that a given value is a number or a date, it knows that it can perform math or other specialized processing with it, which expands the possibilities for how the data gets used.

Different RDF serializations have different conventions for specifying datatypes. The following shows a few triples in Turtle with datatypes assigned:

```
# filename: ex033.ttl

@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix d:   <http://learningsparql.com/ns/data#> .
@prefix dm:  <http://learningsparql.com/ns/demo#> .

d:item342 dm:shipped     "2011-02-14"^^<http://www.w3.org/2001/XMLSchema#date> .
d:item342 dm:quantity    "4"^^xsd:integer .
d:item342 dm:invoiced    "false"^^xsd:boolean .
d:item342 dm:costPerItem "3.50"^^xsd:decimal .
```

As you can see, the name of the datatype can be a full URI or a prefixed name. Like the prefixes used elsewhere in the data, the `xsd:` prefix on the datatype must also be declared.

When you omit the quotation marks from a Turtle literal, a processor makes certain assumptions about its type if the value is the word "true" or "false" or a number. This means that a SPARQL processor would interpret the following the same way as the previous example:

```
# filename: ex034.ttl

@prefix d:   <http://learningsparql.com/ns/data#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .

d:item342 dm:shipped     "2011-02-14"^^<http://www.w3.org/2001/XMLSchema#date> .
```

```
d:item342 dm:quantity    4 .
d:item342 dm:invoiced    true .
d:item342 dm:costPerItem 3.50 .
```

Assignment of datatypes is pretty straightforward in RDF/XML: store the qname of the datatype in an `rdf:datatype` attribute on the element storing the value. Here's the same data as above expressed as RDF/XML:

```
<!-- filename: ex035.rdf -->

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dm="http://learningsparql.com/ns/demo#"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema#">

  <rdf:Description rdf:about="http://learningsparql.com/ns/demo#item342">
    <dm:shipped rdf:datatype="xsd:date">2011-02-14</dm:shipped>
    <dm:quantity rdf:datatype="xsd:integer">4</dm:quantity>
    <dm:invoiced rdf:datatype="xsd:boolean">false</dm:invoiced>
    <dm:costPerItem rdf:datatype="xsd:decimal">3.50</dm:costPerItem>
  </rdf:Description>

</rdf:RDF>
```

Because ex033.ttl, ex034.ttl, and ex035.rdf all store the same triples, executing a given SPARQL query with any one of these files will give you the same answer.

Making RDF More Readable with Language Tags and Labels

Earlier we saw a triple saying that Tim Berners-Lee's job title at the W3C is "Director", but to W3C staff members at their European headquarters near Nice, France, his title would be "Directeur". RDF serializations each have their own way to attach a language tag to a string of text, and we'll see later how SPARQL lets you narrow your query results to literals tagged in a given language. To represent Berners-Lee's job title in both English and French, we could use these triples:

```
# filename: ex036.ttl

@prefix v:  <http://www.w3.org/2006/vcard/> .

<http://www.w3.org/People/Berners-Lee/card#i> v:title "Director"@en .
<http://www.w3.org/People/Berners-Lee/card#i> v:title "Directeur"@fr .
```

Two-letter codes such as "en" for "English" and "fr" for "French" are part of the ISO 639 standard "Codes for the Representation of Names of Languages." You can augment these with a hyphen and a tag from the ISO 3166-1 standard "Codes for the Representation of Names of Countries and Their Subdivisions" to show country-specific terms like in this example:

```
# filename: ex037.ttl

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

```
:sideDish42 rdfs:label "french fries"@en-US .
:sideDish42 rdfs:label "chips"@en-GB .

:sideDish43 rdfs:label "chips"@en-US .
:sideDish43 rdfs:label "crisps"@en-GB .
```

The `label`predicate from the RDF Schema (RDFS) namespace has always been one of RDF's most important properties. We saw in Chapter 1 that a triple's subject rarely conveys much information by itself (for example, `:sideDish42 above`), because its job is to be a unique identifier, not to describe something. It's the job of the predicates and objects used with that subject to describe it.

Because of this, it's an RDF best practice to assign `rdfs:label` values to resources so that human readers can more easily see what they represent. For example, in Tim Berners-Lee's FOAF file, he uses the URI *http://www.w3.org/People/Berners-Lee/card#i* to represent himself, but his FOAF file also includes the following triple:

```
# filename: ex038.ttl

<http://www.w3.org/People/Berners-Lee/card#i>
<http://www.w3.org/2000/01/rdf-schema#label>
"Tim Berners-Lee" .
```

Using multiple `rdfs:label` values, each with its own language tag, is a common practice. The DBpedia collection of RDF extracted from Wikipedia infoboxes has fifteen `rdfs:label` values for the resource *http://dbpedia.org/resource/Switzerland*. The following has triples assigning four of these labels to that resource; it uses the comma delimiter to show that the four values are all objects for the same subject and predicate:

```
# filename: ex039.ttl

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

<http://dbpedia.org/resource/Switzerland> rdfs:label "Switzerland"@en,
  "Suiza"@es, "Sveitsi"@fi, "Suisse"@fr .
```

When semantic web applications retrieve information in response to a query, it's very common for them to retrieve the `rdfs:label` values associated with the relevant resources, if they're available, instead of the often cryptic URIs.

The RDF-based W3C SKOS standard for defining vocabularies, taxonomies, and thesauruses offers more specialized versions of the `rdfs:label` property, including the `skos:prefLabel` property for preferred labels and the `skos:altLabel` property for alternative labels. A single concept in the UN Food and Agriculture Organization's SKOS thesaurus for agriculture, forestry, fisheries, food and related domains may have dozens of `skos:prefLabel` values and dozens of `skos:altLabel` values for a single concept, all with separate language tags ranging from English to Farsi to Thai.

Along with the rdfs:label property, the RDF Schema vocabulary provides the rdfs:comment property, which typically stores a longer description of a resource. Using this property to describe how a resource (or property) is used or any special issues associated with it can make the resource's data easier to use, just as adding comments to a program's source code can help people understand how the program works so that they can use its source code more effectively.

Blank Nodes and Why They're Useful

In "More Realistic Data and Matching on Multiple Triples" on page 7, we learned that an RDF dataset is known as a graph. You can picture it visually with the nodes of a graph representing the subject and object resources from a set of triples and the lines connecting those nodes representing the predicates. Nearly all of the nodes have a name consisting of the URI that represents that resource, or, for literals as objects, a string or other typed value.

Wait—"nearly" all of the nodes? What purpose could a node with no name serve? Let's look at an example, first of a set of triples with no blank nodes, and then how a blank node can help to organize them better.

The following shows some triples that represent an entry for someone in our fake address book:

```
# filename: ex040.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .

ab:i0432 ab:firstName     "Richard" ;
         ab:lastName      "Mutt" ;
         ab:postalCode    "49345" ;
         ab:city          "Springfield" ;
         ab:homeTel       "(229) 276-5135" ;
         ab:streetAddress "32 Main St." ;
         ab:region        "Connecticut" ;
         ab:email         "richard49@hotmail.com" .
```

Figure 2-1 has a graph image of this data. Each subject or object value is a labeled node of the graph image, and the predicates are the labeled arcs connecting the nodes to show their relationships.

We've seen that the order of triples doesn't matter in RDF, and Richard's mailing address information is a bit difficult to find scattered among the other information about him. In database modeling terms, the address book entry in ex040.ttl is very flat, being just a list of values about Richard with no structure to those values.

The following version has a new predicate, d:address, but its value has a strange namespace prefix: an underscore. That value in turn has its own values describing it: the individual components of Richard's address.

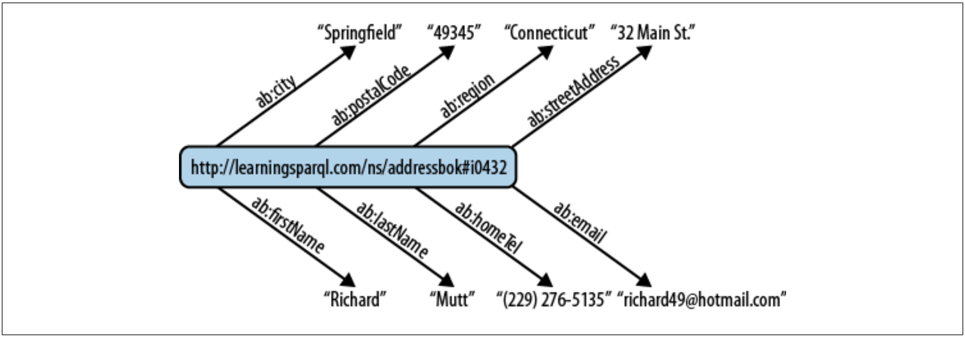


Figure 2-1. Graph of the ab:i0432 address book entry

```
# filename: ex041.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .

ab:i0432 ab:firstName     "Richard" ;
         ab:lastName      "Mutt" ;
         ab:homeTel       "(229) 276-5135" ;
         ab:email         "richard49@hotmail.com" ;
         ab:address       _:b1 .

_:b1     ab:postalCode    "49345" ;
         ab:city          "Springfield" ;
         ab:streetAddress "32 Main St." ;
         ab:region        "Connecticut" .
```

The underscore prefix means that this is a special kind of node known as a *blank node* or *bnode*. It has no permanent identity; its only purpose is to group together some other values. The `b1` it uses for a local name is just a placeholder in case other parts of this dataset need to refer to this grouping of triples. RDF software that reads this data can ignore the `b1` value, but it must remember that Richard (or, more technically, resource `ab:i0432`) has an `ab:address` value that points to those four other values.

Figure 2-2 shows a graph of the ex041.ttl data. The node representing the `ab:address` value on the image has no name, because that node has no identity—it's blank. This shows that while an RDF parser will pay attention to all the other subjects, predicates, and objects in the ex041.ttl dataset, the `b1` after the `_:` means nothing to the parser, but the parser does remember what the node is connected to. The `b1` in ex041.ttl is just a temporary local name for the node in that version of the data. This name may not make it into new versions of data when the data is copied—for example, into the Rhizomik parser that created the diagram.

Turtle and SPARQL sometimes use a pair of square braces (`[]`) instead of a prefixed name with an underscore prefix to represent a blank node.

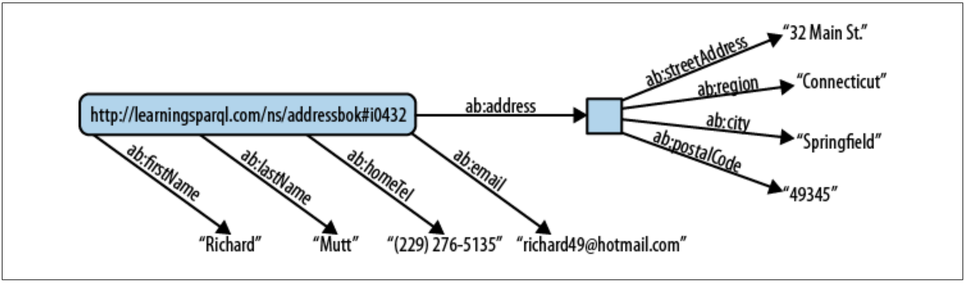


Figure 2-2. Using a blank node to group together postal address data

In the example, the `_:b1` blank node is the object of one triple and the subject of several others. This is common, because RDF and SPARQL use blank nodes to connect things up. For example, if I have address book data structured like the ex041.ttl sample above, I can use a SPARQL query to ask for the `d:streetAddress` value of the `d:address` node from the address book entry that has a `d:firstName` of "Richard" and a `d:lastName` of "Mutt". The `d:address` node doesn't have a URI that I can use to refer to it, but the query wouldn't need it, because it can just say that it wants the address values from the entry with a `d:firstName` of "Richard" and a `d:lastName` of "Mutt".

Named Graphs

Named graphs are another way to group triples together. When you assign a name to a set of triples, of course the name is a URI, so because RDF lets you assign metadata to anything that you can identify with a URI, this lets you assign metadata to that set of triples. For example, you could say that a given set of triples in a triplestore came from a certain source at a certain time, or that a particular set should be replaced by another set.

The original RDF specifications didn't cover this, but it eventually became clear that this would be valuable, so later specifications did cover it—especially the SPARQL specifications. We'll see in Chapter 3 and Chapter 6 how to query and update named subsets of a collection of triples.

Reusing and Creating Vocabularies: RDF Schema and OWL

You can make up new URIs for all the resources and properties in your triples, but when you use existing ones, it's easier to connect other data with yours, which lets you do more with it. I've already used examples of properties from several existing vocabularies such as FOAF and Dublin Core. If you want to use existing vocabularies of properties, what do these vocabularies look like? What format do they take?

They're usually stored using the RDF Schema and OWL standards. Vocabularies expressed using one of these standards provides a good reference for someone (or something) writing a SPARQL query and wondering what properties are available in the dataset being queried. As a bonus, the definitions often add metadata about the declared vocabulary terms, making it easier to learn about them so that your query can make better use of them.

Earlier, I mentioned the W3C RDF Schema specification when describing the `rdfs:label` and `rdfs:comment` properties. Its full title is "RDF Vocabulary Description Language: RDF Schema," and it gives people a way to describe vocabularies. As you may have guessed, you describe these vocabularies with RDF, so this metadata is just as accessible to your SPARQL queries as the data itself.

Here are a few of the triples from the RDF Schema vocabulary description of the Dublin Core vocabulary. They describe the term "creator" that we used to describe Tim Berners-Lee's relationship to the book represented by the URI *urn:isbn:006251587X*:

```
# filename: ex042.ttl

@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

dc:creator
      rdf:type rdf:Property ;
      # a rdf:Property ;
      rdfs:comment "An entity primarily responsible for making the resource."@en-US ;
      rdfs:label "Creator"@en-US .
```

The last two triples describe the *http://purl.org/dc/elements/1.1/creator* property using the `rdfs:comment` and `rdfs:label` properties, and they both have language tags to show that they're written in American English. (As if to prove a point about the need for proper disambiguation of terms that identify things, the Dublin Core standard was developed in Dublin, Ohio, not in Ireland's more famous Dublin.)

The first triple in ex042.ttl uses a property we haven't seen before to declare that `dc:creator` is a property: `rdf:type`. (Turtle, N3, and SPARQL offer you the shortcut of using the word "a" instead of `rdf:type` in a triple declaring that something is of a particular type, as shown in the commented-out line in ex042.ttl that would mean the same thing as the line above it.) The `rdf:type` triple actually says that `dc:creator` is a member of the `rdf:Property` class, but in plain English, it's saying that it's a property, which is why the commented-out version can be easier to read.

When you use the word "a" like this in a SPARQL query, it must be in lowercase. This is the only case-sensitive part of SPARQL.

Nothing in any RDF specification says that you have to declare properties before using them. However, declaring them offers the advantage of letting you assign metadata to the properties themselves, like the `rdfs:comment` and `rdfs:label` properties in ex042.ttl, so that people can learn more about these properties and use them as their authors intended. Declaring a new property to have specific relationships with other properties and classes lets processors such as SPARQL engines do even more with its values.

RDF Schema is itself a vocabulary with a schema whose triples declare facts (for example, that `rdfs:label` and `rdfs:comment` are properties) just like the Dublin Core schema excerpt above declares that `dc:creator` is a property.

In addition to identifying properties, RDF Schema lets you define new classes of resources. For example, the following shows how I might declare `ab:Musician` and `ab:MusicalInstrument` classes for my address book data:

```
# filename: ex043.ttl

@prefix ab:   <http://learningsparql.com/ns/addressbook#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ab:Musician
      rdf:type rdfs:Class ;
      rdfs:label "Musician" ;
      rdfs:comment "Someone who plays a musical instrument" .

ab:MusicalInstrument
      a rdfs:Class ;
      rdfs:label "musical instrument" .
```

(I didn't bother with an `rdfs:comment` for `ab:MusicalInstrument` because I thought the `rdfs:label` value was enough.)

There's a lot more metadata that we can assign when we declare a class—for example, that it's subclass of another one—but with just the metadata above we can see another very nice feature of RDFS. Below I've declared an `ab:playsInstrument` property:

```
# filename: ex044.ttl

@prefix ab:   <http://learningsparql.com/ns/addressbook#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ab:playsInstrument
      rdf:type rdf:Property ;
      rdfs:comment "Identifies the instrument that someone plays" ;
      rdfs:label "plays instrument" ;
      rdfs:domain ab:Musician ;
      rdfs:range ab:MusicalInstrument .
```

The `rdfs:domain` property means that if I use this `ab:playsInstrument` property in a triple, then the subject of the triple is an `ab:Musician`. The `rdfs:range` property means that the object of such a triple is an `ab:MusicalInstrument`.

Let's say I add one triple to the end of the ex040.ttl address book example, like this:

```
# filename: ex045.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .

ab:i0432 ab:firstName     "Richard" ;
        ab:lastName       "Mutt" ;
        ab:postalCode     "49345" ;
        ab:city           "Springfield" ;
        ab:homeTel        "(229) 276-5135" ;
        ab:streetAddress  "32 Main St." ;
        ab:region         "Connecticut" ;
        ab:email          "richard49@hotmail.com" ;
        ab:playsInstrument ab:vacuumCleaner .
```

In traditional object-oriented thinking, if we say "members of class `Musician` have a `playsInstrument` property," this means that a member of the `Musician` class must have a `playsInstrument` value. RDFS and OWL approach this from the opposite direction: the last two triples of ex044.ttl tell us that if something has a `playsInstrument` value, then it's a member of class `Musician` and the `playsInstrument` value is a member of class `ab:MusicalInstrument`.

Once I've added the new triple shown at the end of ex045.ttl, an RDFS-aware SPARQL processor knows that Richard Mutt (or, more precisely, resource `ab:i0432`) is now a member of the class `ab:Musician`, because `ab:playsInstrument` has a domain of `ab:Musician`. Because `ab:playsInstrument` has a range of `ab:MusicalInstrument`, `ab:vacuumCleaner` is now a member of the `ab:MusicalInstrument` class, even if it never was before.

When I tell an RDFS-aware SPARQL engine to give me the first and last names of all the musicians in the address book data above, it will list Richard Mutt. If I was using an object-oriented system, I'd have to declare a new musician instance and then assign it all the details about Richard. Using semantic web standards, by adding one property to the metadata about the existing resource `ab:i0432`, that resource becomes a member of a class that it wasn't a member of before.

> An "RDFS-aware" SPARQL engine is probably going to be an OWL engine. OWL builds on RDFS, and not much software is available that supports RDFS without also supporting at least some of OWL.

This ability of RDF resources to become members of classes based on their data values has made semantic web technology popular in areas such as medical research and intelligence agencies. Researchers can accumulate data with little apparent structure and then see what structure turns out to be there—that is, which resources turn out to be members of which classes, and what their relationships are.

RDFS lets you define classes as subclasses of other ones, and (unlike object-oriented systems) properties as subproperties of other ones, which broadens the possibilities for how you can use SPARQL to retrieve information. For example, if I said that `ab:Musician` was a subclass of `foaf:Person` and then queried an RDFS-aware processor for the phone numbers of all the `foaf:Person` instances in a dataset that included the ex044.ttl and ex045.ttl data, the processor would give me Richard's phone number, because by being in the class of musicians Richard is also in the class of persons.

The W3C's Web Ontology Language, abbreviated as OWL because it's easier to pronounce than "WOL," builds on RDFS to let you define ontologies. Ontologies are formal definitions of vocabularies that allow you to define complex structures as well as new relationships between your vocabulary terms and between members of the classes that you define. Ontologies often describe very specific domains such as scientific research areas so that scientists from different institutions can more easily share data.

An ontology defined with OWL is also just another collection of triples. OWL itself is an OWL ontology, declaring classes and properties that OWL-aware software will watch for so that it can make inferences from the data that use them.

Without defining a large, complex ontology, many semantic web developers use just a few classes and properties from OWL to add metadata to their triples. For example, how much information do you think the following dataset has about resource `ab:i9771`, Cndy Marshall?

```
# filename: ex046.ttl

@prefix ab:   <http://learningsparql.com/ns/addressbook#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .

ab:i0432
   ab:firstName "Richard" ;
   ab:lastName  "Mutt" ;
   ab:spouse    ab:i9771 .

ab:i8301
   ab:firstName "Craig" ;
   ab:lastName  "Ellis" ;
   ab:patient   ab:i9771 .

ab:i9771
   ab:firstName "Cindy" ;
   ab:lastName  "Marshall" .
```

```
ab:spouse
   rdf:type owl:SymmetricProperty ;
   rdfs:comment "Identifies someone's spouse" .

ab:patient
   rdf:type rdf:Property ;
   rdfs:comment "Identifies a doctor's patient" .

ab:doctor
   rdf:type rdf:Property ;
   rdfs:comment "Identifies a doctor treating the named resource" ;
   owl:inverseOf ab:patient .
```

It looks like it only includes a first name and last name, but an OWL-aware processor will see more. Because the `ab:spouse` property is defined in this dataset as being symmetric, and resource `ab:i0432` has an `ab:spouse` value of `ab:i9771` (Cindy), an OWL processor knows that resource `ab:i9771` has resource `ab:i0432` as her spouse. It also knows that if the `ab:patient` property is the inverse of the `ab:doctor` property, and resource `ab:i8301` has an `ab:patient` value of `ab:i9771`, then resource `ab:i9771` has an `ab:doctor` value of `i8301`. Now we know who Cindy's spouse and doctor are, even though these facts are not explicitly included in the dataset.

OWL offers many other ways to define property and class relationships so that a processor can infer new information from an existing set. This example shows that you don't need lots of OWL to gain some advantages from it. By adding just a little bit of metadata (for example, the information about the `ab:spouse`, `ab:patient`, and `ab:doctor` properties above) to a small set of data (the information about Richard, Craig, and Cindy) we got more out of this dataset than we originally put into it. This is one of the great payoffs of semantic web technology.

The OWL 2 upgrade to the original OWL standard introduced several profiles, or subsets of OWL, that are specialized for certain kinds of applications. These profiles are easier to implement and use than attempting to take on all of OWL at once. If you're thinking of doing some data modeling with OWL, look into OWL 2 RL, OWL 2 QL, and OWL 2 EL as possible starting points for your needs.

Of all the W3C semantic web standards, OWL is the key one for putting the "semantic" in "semantic web." The term "semantics" is sometimes defined as the meaning behind words, and those who doubt the value of semantic web technology like to question the viability of storing all the meaning of a word in a machine-readable way. As we saw above, though, we don't need to store all the meaning of a word to add value to a given set of data. For example, simply knowing that "spouse" is a symmetric term made it possible to find out the identity of Cindy's spouse, even though this fact was not part of the dataset.

Linked Data

The idea of Linked Data is newer than that of the semantic web, but sometimes it's easier to think of the semantic web as building on the ideas behind Linked Data. Linked Data is not a specification, but a set of best practices for providing a data infrastructure that makes it easier to share data across the web. You can then use semantic web technologies such as RDFS, OWL, and SPARQL to build applications around that data.

Tim Berners-Lee came up with these four principles of Linked Data in 2006 (I've bolded his wording and added my own commentary):

1. **Use URIs as names for things.** URIs are the best way available to uniquely identify things, and therefore to identify connections between things.
2. **Use HTTP URIs so that people can look up those names.** You may have seen URIs that begin with ftp:, mailto:, or prefixes made up by a particular community, but using these other ones reduces interoperability, and interoperability is what it's all about.
3. **When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL).** A URI can just be a name and not actually the address of a web page, but this principle says that you may as well put something there. It can be an HTML web page, or something else; whatever it is, it should use a recognized standard. RDFS and OWL let you spell out out a list of terms and information about those terms and their relationships in a machine-readable way—readable, for example, by SPARQL queries. Because of this, if a URI that identifies a resource leads to RDFS or OWL declarations about that resource, this is a big help to applications. (The asterisk in "RDF*" means that it refers to both RDF and RDFS as standards.)
4. **Include links to other URIs so that they can discover more things.** Imagine if none of the original HTML pages had `a` elements to link them to other pages. It wouldn't have been much of a web. In addition to the HTML linking element, various RDF vocabularies provide other properties as a way to say "this data (or this element of data) has a specific relationship to another resource on the web." When applications can follow these links, they can do interesting new things.

In a talk at the 2010 Gov 2.0 Expo in Washington, D.C., Berners-Lee gave a fairly nontechnical introduction to Linked Data in which he suggested the awarding of stars to governments for sharing data on the web. They would get at least one star for any kind of sharing at all. They would be awarded two stars for sharing it in a machine-readable format (as opposed to a scan of a fax), regardless of the format. They deserve three stars for sharing data on the web using a nonproprietary format, such as comma-separated values instead of Microsoft Excel spreadsheets. Putting it in a Linked Data format, in which concepts were identified by URLs so that we could more easily cross-reference them with other data, would earn four stars. (I'm sure that with a more technical audience, he would have used the term "URI," but the older term would have been more familiar to his audience that day.) A government should get a full five stars for connecting the data to other data—that is, by providing links to related data, especially links that make use of the URLs in the data.

He gave this talk at a time when the US and UK governments were just starting to make more data available on the web. It would be nice for SPARQL-based applications if all that public data was available in RDF, but that can be a lot to ask of government agencies with low budgets and limited technical expertise.

For some, this is not a limitation, but an opportunity: Professor Jim Hendler and his Tetherless World Constellation group at Rensselaer Polytechnic Institute converted a lot of the simpler data that they found through the US Data.gov project to RDF so that they could build semantic web applications around it. After seeing this work, US CIO Vivek Kundra appointed Hendler the "Internet Web Expert" for Data.gov.

The term "Linked Open Data" is also becoming popular. The growing amount of freely available public data is a great thing, but remember: just as web servers and web pages can be great for sharing information among employees behind a company's firewall without making those web pages accessible to the outside world, the techniques of Linked Data can benefit an organization's operations behind the firewall as well, making it easier to share data across internal silos.

SPARQL's Past, Present, and Future

RDF provides great ways to model and store data, and the Linked Data infrastructure offers tons of data to play with. As long as RDF has been around, there have been programming libraries that let you load triples into the data structures of popular programming languages so that you could build applications around that data. As the relational database and XML worlds have shown, though, a straightforward query language that requires no compiling of code to execute makes it much easier for people (including part-time developers dabbling in the technology) to quickly assemble applications.

RDF became a standard in 1999. By 2004, over a dozen query languages had been developed as commercial, academic, and personal projects. (In fact, one N3 feature that was omitted from the Turtle subset of N3 was its query language.) That year, the W3C formed the RDF Data Access Working Group.

After the RDF-DAWG gathered use cases and requirements, they released the first draft of the SPARQL Query Language specification in late 2004. In early 2008, the query language, protocol, and query results XML format became Recommendations, or official W3C specifications.

By then, SPARQL implementations already existed, and once the 1.0 standard was official, more came along as the earlier query languages adapted to support the standard. Triplestores added support for SPARQL, and more standalone tools like ARQ came along. SPARQL endpoints began appearing, accepting SPARQL queries delivered over the web and returning results in a choice of formats.

Early use of SPARQL led to new ideas about ways to improve it. The RDF-DAWG's charter expired in 2009, so the W3C created a new Working Group to replace them: the SPARQL Working Group. The SPARQL Working Group released their first Working Drafts of the SPARQL 1.1 specifications in late 2009, and implementations of the new features began appearing shortly afterward. As of this writing, the SPARQL 1.1 specifications are in Last Call status.

The SPARQL Specifications

There are three SPARQL 1.0 specification documents:

- **SPARQL Query Language for RDF** covers the syntax of the queries themselves. As the "QL" in "SPARQL," it's where you'll spend most of your time. Chapter 1 and Chapter 3 cover the use of the query language, including new features that SPARQL 1.1 adds.
- **SPARQL Protocol for RDF** specifies how a program should pass SPARQL queries to a SPARQL query processing service and how that service should return the results. This specification is more of a concern for SPARQL software implementers than people writing queries (and, in SPARQL 1.1, update requests) to run against datasets.
- **SPARQL Query Results XML Format** describes a simple XML format for query processors to use when returning results. If you send a query to a processor and request this XML format, you can then use XSLT or another XML tool to convert the results into whatever you like. We'll learn more about this in "SPARQL Query Results XML Format" on page 217 of Chapter 7.

The SPARQL 1.1 effort added several new technical documents to the SPARQL collection. Some will become official Recommendations, but others are considered background information (or, in standards-speak, they're not "normative"). Several are aimed more at SPARQL software implementers than at people writing queries and update requests to run against datasets.

- The **Federation Extensions** specification describes how a single query can retrieve data from multiple sources. This makes it much easier to build applications that take advantage of distributed environments. "Federated Queries: Searching Multiple Datasets with One Query" on page 98 in Chapter 3 describes how to do this.
- The **Update** specification is the key difference between SPARQL 1.0 and 1.1, because it takes SPARQL from just being a query language to something that can add data to a dataset and replace and delete it as well. Chapter 6 covers the key features of this spec.

- The **Service Description** describes how a client program can ask a SPARQL engine exactly what features it supports.
- The **Query Results JSON Format** describes a JSON equivalent of the Query Results XML Format.
- The **Graph Store HTTP Protocol** extends the SPARQL Protocol with a REST-like API for communication between a client and a SPARQL processor about graphs, or sets of triples. For example, it provides HTTP ways to say "here's a graph to add to the dataset" or "delete the graph *http://my/fine/graph* from the dataset."
- The **Entailment Regimes** specification describes criteria for determining what information a SPARQL processor should take into account when performing *entailment*. What is entailment? If A entails B, and A is true, then we know that B is true. If A is a complicated set of facts, it can be handy to have technology such as an OWL-aware SPARQL processor to help you discover whether B is true. Because of some confusion over exactly which extra information should be considered when performing entailment, this spec spells out which sets of information to take into account and when.
- The **SPARQL New Features and Rationale** document has a self-explanatory title. It's a great place to start for experienced SPARQL 1.0 users who want to get the most out of 1.1 quickly.

Summary

In this chapter, we learned:

- What the semantic web is.
- Why URIs are the foundation of the semantic web, their relationship to URLs and IRIs, and the role of namespaces.
- How people store RDF, and how they can identify the datatypes and the languages of string literals (for example, Spanish, German, Mexican Spanish, or Austrian German) in their data.
- What blank nodes and named graphs are and the flexibility they can add to how you arrange and track your data.
- How the RDFS and OWL specifications let you define properties and classes as well as metadata about these properties and classes to let you get more out of the data they describe.
- How Linked Data is a popular set of best practices for sharing data that semantic web applications can build on, and what kind of data is becoming available.
- SPARQL's history and the specifications that make up the SPARQL standard.

CHAPTER 3

SPARQL Queries: A Deeper Dive

Chapter 1 gave you your first taste of writing and running SPARQL queries. In this chapter, we'll dig into more powerful features of the SPARQL query language:

- "More Readable Query Results" on page 46: URIs are important for identifying and linking things, but when it's time to display query results in an application, users want to see information they can read, not something that looks like a bunch of web addresses.
- "Data That Might Not Be There" on page 53: We've seen how to request data that matches certain patterns. When you can ask for data that may or may not match certain patterns, it makes your queries more flexible, which is especially useful when exploring data you're unfamiliar with.
- "Finding Data That Doesn't Meet Certain Conditions" on page 57: Much of SPARQL is about retrieving data that fits certain patterns. What if you want the data that doesn't fit a particular pattern—for example, to clean it up?
- "Searching Further in the Data" on page 59: SPARQL offers some simple ways to ask for a set of triples and any other triples that are connected to them.
- "Eliminating Redundant Output" on page 67: If you're looking for triples that fit some pattern and a SPARQL query engine finds multiple instances of certain values, it will show you all of them. Unless you tell it not to.
- "Combining Different Search Conditions" on page 70: SPARQL lets you ask, in one query, for data that fits certain patterns and other data that fits other patterns; you can also ask for data that meets either of two sets of patterns.
- "FILTERing Data Based on Conditions" on page 73: Specifying boolean conditions, which may include regular expressions, lets you focus your result set even more.
- "Retrieving a Specific Number of Results" on page 76: If your query may retrieve more results than you want, you can limit the returned results to a specific amount.

- "Querying Named Graphs" on page 78: When triples in your dataset are organized into named graphs, you use their membership in one or more graphs as part of your search conditions.
- "Queries in Your Queries" on page 85: Subqueries let you put queries within queries so that you can break down a complex query into more easily manageable parts.
- "Combining Values and Assigning Values to Variables" on page 86: SPARQL 1.1 gives you greater control over how you use variables so that your queries and applications have even greater flexibility for what they do with retrieved data.
- "Sorting, Aggregating, Finding the Biggest and Smallest and..." on page 88: You don't have to just list the data you found; you can sort it and find totals, subtotals, averages, and other aggregate values for the whole dataset or for sorted groupings of data.
- "Querying a Remote SPARQL Service" on page 95: You can run your query against a single file of data sitting on your hard disk, but you can also run it against other datasets accessible to your computer around the world.
- "Federated Queries: Searching Multiple Datasets with One Query" on page 98: A single query can ask for data from several sources, both local and remote, and then put the results together.

More Readable Query Results

In "More Realistic Data and Matching on Multiple Triples" on page 7 of Chapter 1, we saw this query, which asks who in the address book data has the phone number (245) 646-5488:

```
# filename: ex008.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?person
WHERE
{ ?person ab:homeTel "(229) 276-5135" . }
```

When run against this data,

```
# filename: ex012.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName "Richard" .
d:i0432 ab:lastName  "Mutt" .
d:i0432 ab:homeTel   "(229) 276-5135" .
d:i0432 ab:email     "richard49@hotmail.com" .

d:i9771 ab:firstName "Cindy" .
d:i9771 ab:lastName  "Marshall" .
```

```
d:i9771 ab:homeTel   "(245) 646-5488" .
d:i9771 ab:email     "cindym@gmail.com" .

d:i8301 ab:firstName "Craig" .
d:i8301 ab:lastName  "Ellis" .
d:i8301 ab:email     "craigellis@yahoo.com" .
d:i8301 ab:email     "c.ellis@usairwaysgroup.com" .
```

it produced this result:

```
---------------------------------------------
| person                                    |
=============================================
| <http://learningsparql.com/ns/data#i0432> |
---------------------------------------------
```

This is not a very helpful answer, but by asking for the first and last names of the person with that phone number, like this,

```
# filename: ex017.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last
WHERE
{
  ?person ab:homeTel "(229) 276-5135" .
  ?person ab:firstName ?first .
  ?person ab:lastName  ?last .
}
```

we get a much more readable answer:

```
----------------------
| first     | last   |
======================
| "Richard" | "Mutt" |
----------------------
```

As a side note, a semicolon means the same thing in SPARQL that it means in Turtle and N3: “here comes another predicate and object to go with this triple’s subject.” Using this abbreviation, the following query will work exactly the same as the previous one:

```
# filename: ex047.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last
WHERE
{
  ?person ab:homeTel "(229) 276-5135" ;
          ab:firstName ?first ;
          ab:lastName  ?last .
}
```

Many if not most SPARQL queries include multiple triple patterns that reference the same subject, so the semicolon abbreviation for listing triples about the same subject is very common.

SPARQL queries often look up some data and the human-readable information associated with that data and then return only the human-readable data. For the address book sample data, this human-readable data is the `ab:firstName` and `ab:lastName` values associated with each entry. Different datasets may have different properties associated with their URIs as readable alternatives, but one property in particular has been popular for this since the first RDF specs came out: the `rdfs:label` property.

When your query retrieves data in the form of URIs, it's a good idea to also retrieve any `rdfs:label` values associated with those URIs.

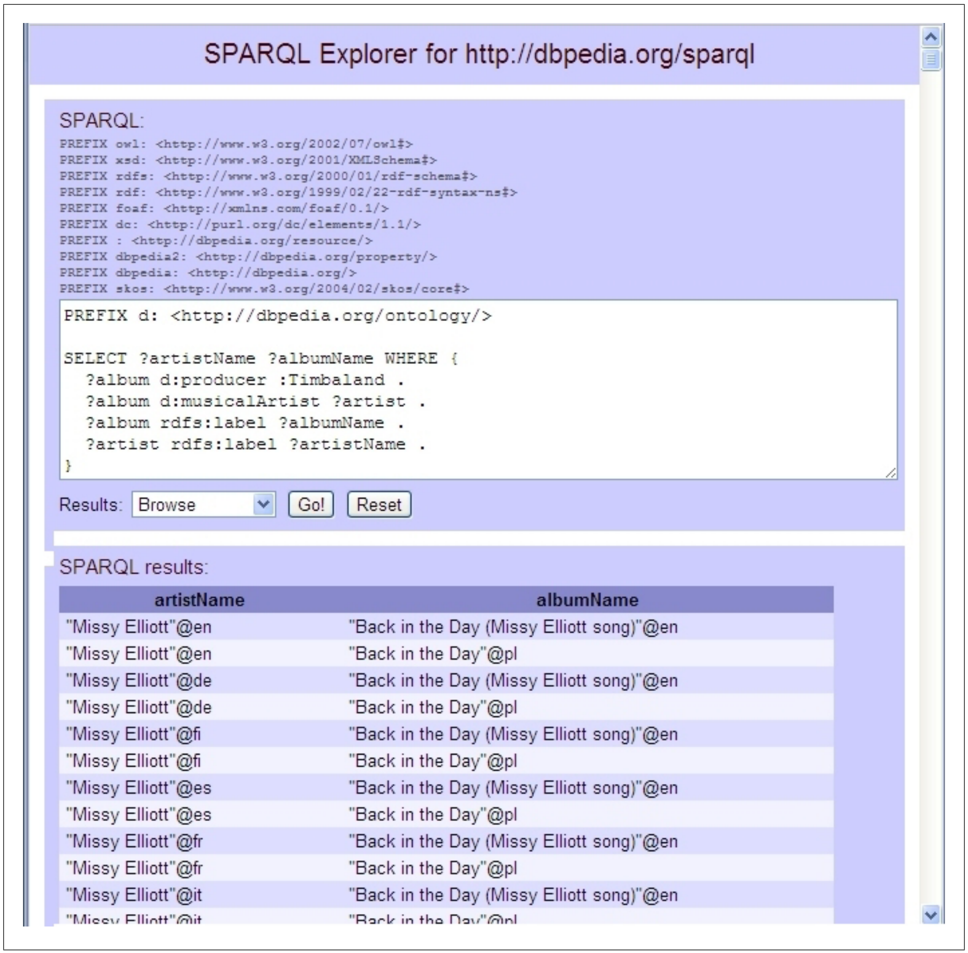
Using the Labels Provided by DBpedia

In another example in Chapter 1, we saw a query that listed albums produced by Timbaland and the artists associated with those albums. The query results actually listed the URIs that represented those albums and artists, and while they were somewhat readable, we can do better by retrieving the `rdfs:label` values associated with those URIs and SELECTing those instead:

```
# filename: ex048.rq

PREFIX d: <http://dbpedia.org/ontology/>

SELECT ?artistName ?albumName
WHERE
{
  ?album d:producer :Timbaland .
  ?album d:musicalArtist ?artist .
  ?album rdfs:label ?albumName .
  ?artist rdfs:label ?artistName .
}
```

Like several queries and data files that we've already seen, this query uses the prefix `d:`, but note that here it stands in for a different namespace URI. Never take it for granted that a given prefix stands for a particular URI; always check its declaration, because RDF (and XML) software is required to check. Also, all prefixes must be declared first; in the query above, it looks as though the `:` in `:Timbaland` hasn't been declared, but on the DBpedia *http://dbpedia.org/snorql/* form where I entered this query, you can see that the declaration for `:` is already there.

As it turns out, each artist and album name have multiple `rdfs:label` values with different language tags assigned to each—for example "en" for English and "de" (Deutsche) for German. (The album title is still shown in English because it was released under that title in those countries.) When we enter this query into DBpedia's SNORQL interface, it gives us back every combination of them, starting with several for Missy Elliot's "Back in the Day," as shown in Figure 3-1.



SPARQL Explorer for http://dbpedia.org/sparql

SPARQL:

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX d: <http://dbpedia.org/ontology/>

SELECT ?artistName ?albumName WHERE {
  ?album d:producer :Timbaland .
  ?album d:musicalArtist ?artist .
  ?album rdfs:label ?albumName .
  ?artist rdfs:label ?artistName .
}
```

Results: Browse Go! Reset

SPARQL results:

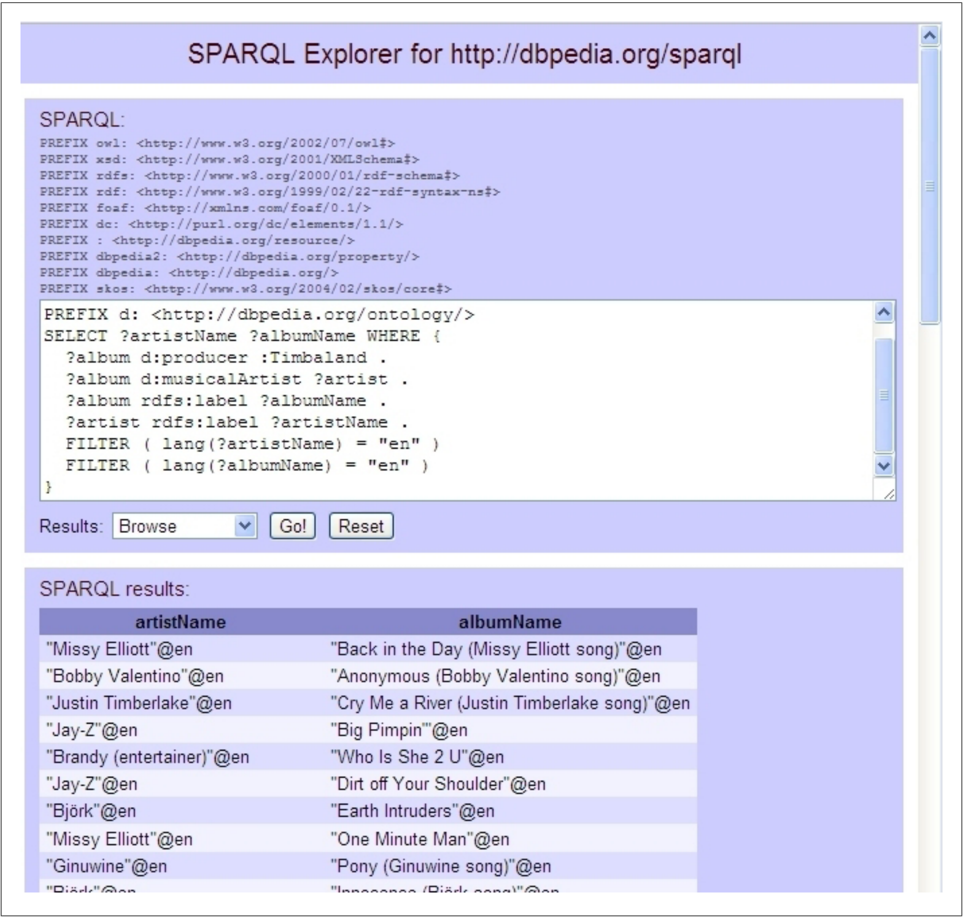
artistName	albumName
"Missy Elliott"@en	"Back in the Day (Missy Elliott song)"@en
"Missy Elliott"@en	"Back in the Day"@pl
"Missy Elliott"@de	"Back in the Day (Missy Elliott song)"@en
"Missy Elliott"@de	"Back in the Day"@pl
"Missy Elliott"@fi	"Back in the Day (Missy Elliott song)"@en
"Missy Elliott"@fi	"Back in the Day"@pl
"Missy Elliott"@es	"Back in the Day (Missy Elliott song)"@en
"Missy Elliott"@es	"Back in the Day"@pl
"Missy Elliott"@fr	"Back in the Day (Missy Elliott song)"@en
"Missy Elliott"@fr	"Back in the Day"@pl
"Missy Elliott"@it	"Back in the Day (Missy Elliott song)"@en
"Missy Elliott"@it	"Back in the Day"@pl

Figure 3-1. Beginning of results for query about albums produced by Timbaland

FILTERs, which we'll learn more about in "FILTERing Data Based on Conditions" on page 73, let us specify that we only want English language artist labels and album names:

```
# filename: ex049.rq
```

```
PREFIX d: <http://dbpedia.org/ontology/>

SELECT ?artistName ?albumName
WHERE
{
  ?album d:producer :Timbaland .
  ?album d:musicalArtist ?artist .
  ?album rdfs:label ?albumName .
  ?artist rdfs:label ?artistName .
  FILTER ( lang(?artistName) = "en" )
  FILTER ( lang(?albumName) = "en" )

}
```



SPARQL Explorer for http://dbpedia.org/sparql

SPARQL:

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
```

```
PREFIX d: <http://dbpedia.org/ontology/>
SELECT ?artistName ?albumName WHERE {
  ?album d:producer :Timbaland .
  ?album d:musicalArtist ?artist .
  ?album rdfs:label ?albumName .
  ?artist rdfs:label ?artistName .
  FILTER ( lang(?artistName) = "en" )
  FILTER ( lang(?albumName) = "en" )
}
```

SPARQL results:

artistName	albumName
"Missy Elliott"@en	"Back in the Day (Missy Elliott song)"@en
"Bobby Valentino"@en	"Anonymous (Bobby Valentino song)"@en
"Justin Timberlake"@en	"Cry Me a River (Justin Timberlake song)"@en
"Jay-Z"@en	"Big Pimpin'"@en
"Brandy (entertainer)"@en	"Who Is She 2 U"@en
"Jay-Z"@en	"Dirt off Your Shoulder"@en
"Björk"@en	"Earth Intruders"@en
"Missy Elliott"@en	"One Minute Man"@en
"Ginuwine"@en	"Pony (Ginuwine song)"@en

Figure 3-2. Albums produced by Timbaland, restricted to English-language data

Getting Labels from Schemas and Ontologies

RDF Schemas and OWL ontologies can provide all kinds of metadata about the terms and relationships they define, and sometimes the simplest and most useful set of information is the `rdfs:label` properties associated with the terms that those ontologies define.

Because RDF Schemas and OWL ontologies are themselves collections of triples, if a dataset has an ontology associated with it, querying the dataset and ontology together can help you get more out of that data—which is what metadata is for.

Resources used as the subjects or objects of triples often have metadata associated with them, but remember: predicates are resources too, and often have valuable metadata such as `rdfs:label` values associated with them in an RDF Schema or OWL ontology. As with any other resources, `rdfs:label` values are one of the first things to check for.

Here's some data about Richard Mutt expressed using the FOAF vocabulary. The property names aren't too cryptic, but they're not too user-friendly either:

```
# filename: ex050.ttl

@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<http://www.learningsparql.com/ns/demo#i93234>
        foaf:nick "Dick" ;
        foaf:givenname "Richard" ;
        foaf:mbox "richard49@hotmail.com" ;
        foaf:surname "Mutt" ;
        foaf:workplaceHomepage <http://www.philamuseum.org/> ;
        foaf:aimChatID "bridesbachelor" .
```

By querying this data and the FOAF ontology together, we can ask for the labels associated with the properties, which makes the data easier to read. ARQ can accept multiple `--data` arguments, and with the FOAF ontology (available at *http://xmlns.com/foaf/spec/index.rdf*) stored in a file called index.rdf, the following query runs the ex052.rq query against the combined triples of index.rdf and ex050.ttl:

```
arq --data ex050.ttl --data index.rdf --query ex052.rq
```

Here's the ex052.rq query. The first triple pattern asks for all triples, because all three parts of the triple pattern are variables. The second triple pattern binds `rdfs:label` values associated with the `?property` values from the first triple pattern to the `?propertyLabel` variable. The SELECT list asks for these `?propertyLabel` values, not the URIs that represent the properties, and the `?value` values:

```
#filename: ex052.rq

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?propertyLabel ?value
WHERE
{
  ?s ?property ?value .
  ?property rdfs:label ?propertyLabel .
}
```

Note how the `?property` variable is in the predicate position of the first triple pattern and the subject position of the second one. Putting the same variable in different parts of different triple patterns is a common way to find interesting connections between data that you know about and data that you don't know as well.

It's also how your query can take advantage of the fact that a graph of triples is really a graph of connected nodes and not just a list of triples. A given resource doesn't always have to be either a subject or a predicate or an object; it can play two or three of these roles in different triples. Putting the same variable in different positions in your query's triple patterns, like ex050.ttl does with `?property` above, lets you find the resources that make these connections.

The result of running this query against the combination of the Richard Mutt data shown above and the FOAF ontology is much more readable than a straight dump of the data itself:

```
---------------------------------------------------------
| propertyLabel        | value                          |
=========================================================
| "personal mailbox"   | "richard49@hotmail.com"        |
| "nickname"           | "Dick"                         |
| "Surname"            | "Mutt"                         |
| "Given name"         | "Richard"                      |
| "workplace homepage" | <http://www.philamuseum.org/>  |
| "AIM chat ID"        | "bridesbachelor"               |
---------------------------------------------------------
```

`rdfs:comment` is another popular property, especially in standards that use RDFS or OWL to define terms, so checking if a resource has an `rdfs:comment` value can often help you learn more about it.

There are other kinds of labels besides the `rdfs:label` property. The W3C SKOS (Simple Knowledge Organization System) standard is an OWL ontology designed to represent taxonomies and thesauri. Its `skos:prefLabel` property names a preferred label for a particular concept, and the `skos:altLabel` property names an alternative label. These are both declared in the SKOS ontology as subproperties (that is, more specialized versions) of `rdfs:label`.

It's common for parts of a domain-specific standard (like the `skos:prefLabel` and `skos:altLabel` properties) to be based on something from a more general standard like `rdfs:label`. The connection to the generalized standard makes the data more usable for programs that may not know about the specialized standard.

Data That Might Not Be There

Let's augment our address book database by adding a nickname for Richard and a work telephone number for Craig:

```
# filename: ex054.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName "Richard" .
d:i0432 ab:lastName  "Mutt" .
d:i0432 ab:homeTel   "(229) 276-5135" .
d:i0432 ab:nick      "Dick" .
d:i0432 ab:email     "richard49@hotmail.com" .

d:i9771 ab:firstName "Cindy" .
d:i9771 ab:lastName  "Marshall" .
d:i9771 ab:homeTel   "(245) 646-5488" .
d:i9771 ab:email     "cindym@gmail.com" .

d:i8301 ab:firstName "Craig" .
d:i8301 ab:lastName  "Ellis" .
d:i8301 ab:workTel   "(245) 315-5486" .
d:i8301 ab:email     "craigellis@yahoo.com" .
d:i8301 ab:email     "c.ellis@usairwaysgroup.com" .
```

When I run a query that asks for first names, last names, and work phone numbers,

```
# filename: ex055.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last ?workTel
WHERE
{
  ?s ab:firstName ?first ;
     ab:lastName ?last ;
     ab:workTel ?workTel .
}
```

I get this information for Craig, but for no one else:

```
----------------------------------------
| first   | last    | workTel          |
========================================
| "Craig" | "Ellis" | "(245) 315-5486" |
----------------------------------------
```

Why? Because the triples in the pattern work together as a unit, or, as the SPARQL specification would put it, as a graph pattern. The graph pattern asks for someone who has an `ab:firstName` value, an `ab:lastName` value, and an `ab:workTel` value, and Craig is the only one who does.

Putting the triple pattern about the work phone number in an OPTIONAL graph pattern (remember, a graph pattern is one or more triple patterns inside of curly braces) lets your query say "show me this value, if it's there":

```
# filename: ex057.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last ?workTel
WHERE
{
  ?s ab:firstName ?first ;
     ab:lastName ?last .
  OPTIONAL
  { ?s ab:workTel ?workTel . }
}
```

When we run this query with the same ex054.ttl data, the result includes everyone's first and last names, and Craig's work phone number:

```
---------------------------------------------
| first     | last       | workTel          |
=============================================
| "Craig"   | "Ellis"    | "(245) 315-5486" |
| "Cindy"   | "Marshall" |                  |
| "Richard" | "Mutt"     |                  |
---------------------------------------------
```

Relational database developers may find this similar to the concept of the outer join, in which an SQL query lists the connected data from two or more tables and still includes data from one table that doesn't have corresponding data in another table.

Richard has a nickname value stored with the `ab:nick` property, and no one else does. What happens if we ask for that and put the triple inside the OPTIONAL graph pattern that we just added?

```
# filename: ex059.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last ?workTel ?nick
```

```
WHERE
{
  ?s ab:firstName ?first ;
     ab:lastName ?last .
  OPTIONAL
  {
     ?s ab:workTel ?workTel ;
        ab:nick ?nick .
  }
}
```

We get everyone's first and last names, but no one's nickname or work phone number, even though we have Richard's nickname and Craig's work phone number:

```
-------------------------------------------
| first     | last       | workTel | nick |
===========================================
| "Craig"   | "Ellis"    |         |      |
| "Cindy"   | "Marshall" |         |      |
| "Richard" | "Mutt"     |         |      |
-------------------------------------------
```

Why? Because the OPTIONAL graph pattern is just that: a pattern, and no subjects in our data fit that pattern—that is, no subjects have both a nickname and a work phone number. If we made the optional nickname and the optional work phone number separate OPTIONAL graph patterns, like this, then the query processor will look at them separately:

```
# filename: ex061.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last ?workTel ?nick
WHERE
{
  ?s ab:firstName ?first ;
     ab:lastName ?last .
  OPTIONAL { ?s ab:workTel ?workTel . }
  OPTIONAL { ?s ab:nick ?nick .  }

}
```

The processor then retrieves this data for people who have one or the other of those values:

```
------------------------------------------------------
| first     | last       | workTel          | nick   |
======================================================
| "Craig"   | "Ellis"    | "(245) 315-5486" |        |
| "Cindy"   | "Marshall" |                  |        |
| "Richard" | "Mutt"     |                  | "Dick" |
------------------------------------------------------
```

A query processor tries to match the triple patterns in OPTIONAL graph patterns in the order that it sees them, which lets us perform some neat tricks. For example, let's say we want everyone's first name and last name, but we prefer to use the nickname if it's there, in which case we don't want the first name. In the case of our sample data, we want "Dick" to show up as Mr. Mutt's first name and not "Richard".

This next query does this by first looking for subjects with an `ab:lastName` value and then checking whether there is an optional `ab:nick` value. If it finds it, it's going to bind it to the `?first` variable. If not, it will bind the `ab:firstName` value to that variable.

```
# filename: ex063.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last
WHERE
{
  ?s ab:lastName ?last .
  OPTIONAL { ?s ab:nick ?first . }
  OPTIONAL { ?s ab:firstName ?first . }
}
```

The order of OPTIONAL graph patterns matters.

For example, let's say that the query processor found the fourth triple in our ex054.ttl sample data above, so that `?s` holds `d:i0432` and `?first` holds "Dick". When it moves on to the next OPTIONAL graph pattern, it's going to look for a triple with a subject of `d:i0432` (because that's what `?s` has been bound to), a predicate of `ab:firstName`, and an object of `"Dick"`, because that's what `?first` has been bound to. It's not going to find that triple, but because it already has a `?first` and `?last` value and that last triple pattern is optional, it's going to output the third line of data below.

```
------------------------
| first   | last       |
========================
| "Craig" | "Ellis"    |
| "Cindy" | "Marshall" |
| "Dick"  | "Mutt"     |
------------------------
```

For the other two people in the address book, nothing happens with that first OPTIONAL graph pattern, because they don't have `ab:nick` values, so it binds their `ab:firstName` values to the `?first` variable. When the query completes, we have the first name values that we want for everyone.

The OPTIONAL keyword is especially helpful for exploring a new dataset that you're unfamiliar with. For example, if you see that it assigns certain property values to certain resources, remember that not all resources of that type may have those properties assigned to them. Putting triple patterns inside of OPTIONAL sections lets you retrieve values if they're there without interfering with the retrieval of related data if those values aren't there.

Finding Data That Doesn't Meet Certain Conditions

By now, it should be pretty clear how to list the first name, last name, and work number of everyone in the ex054.ttl address book dataset who has a work number; just ask for `ab:firstName`, `ab:lastName`, and `ab:workTel` values that have the same subject, like we did with the ex055.rq query in "Data That Might Not Be There" on page 53:

```
# filename: ex055.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last ?workTel
WHERE
{
  ?s ab:firstName ?first ;
     ab:lastName ?last ;
     ab:workTel ?workTel .
}
```

What if we want to list everyone whose work number is missing? We just saw how to list everyone's names and their work number if they have one. The SPARQL 1.0 way to list everyone whose work number is missing builds on that; SPARQL 1.1 provides two options, both of which are simpler.

In the first example, which is the only way to do this in SPARQL 1.0, our query asks for each person's first and last names and work phone number if they have it, but it has a filter to pass along a subset of the retrieved triples:

```
# filename: ex065.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last
WHERE
{
  ?s ab:firstName ?first ;
     ab:lastName ?last .

  OPTIONAL { ?s ab:workTel ?workNum . }
  FILTER (!bound(?workNum))
}
```

In "Using the Labels Provided by DBpedia" on page 48 we learned about using a FILTER to only retrieve labels with specific language tags assigned to them. The query above uses the boolean `bound()` function to decide what the FILTER statement should pass along. This function returns `true` if the variable passed as a parameter is bound (that is, if it's been assigned a value) and `false` otherwise.

As with several other programming languages, the exclamation point is a "not" operator, so `!bound(?workNum)` will be true if the `?workNum` variable is *not* bound. Using this, running the ex065.rq query above with the ex054.ttl dataset will pass along the first and last names of everyone who didn't have an `ab:workTel` value to assign to the `?workNum` variable:

```
--------------------------
| first     | last       |
==========================
| "Cindy"   | "Marshall" |
| "Richard" | "Mutt"     |
--------------------------
```

1.1 Alert

There are two SPARQL 1.1 options for filtering out data you don't want, and both have a simpler syntax than the SPARQL 1.0 approach, although the 1.0 approach works just fine with a SPARQL 1.1 processor.

Both SPARQL 1.1 alternatives to the `!bound()` trick are more intuitive. The first, the NOT EXISTS test, returns a boolean `true` if the specified graph pattern does not exist. In the following, if it finds a subject with `ab:firstName` and `ab:lastName` values, it will only pass them along if a triple with that same subject and a predicate of `ab:workTel` does not exist:

```
# filename: ex067.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last

WHERE
{
  ?s ab:firstName ?first ;
     ab:lastName ?last .
  NOT EXISTS { ?s ab:workTel ?workNum }
}
```

The other SPARQL 1.1 way to find the first and last names of people with no `ab:workTel` value uses the MINUS keyword. The following query finds all the subjects with an `ab:firstName` and an `ab:lastName` value but uses the MINUS keyword to subtract those that have an `ab:workTel` value:

```
# filename: ex068.rq
```

```
PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last

WHERE
{
  ?s ab:firstName ?first ;
     ab:lastName ?last .
  MINUS { ?s ab:workTel ?workNum }
}
```

For our purposes, these two SPARQL 1.1 approaches have the same results as the SPARQL 1.0 `!bound()` trick, and in most cases you'll find them behaving identically.

There are some edge cases where NOT EXISTS and MINUS may return different results. See the SPARQL 1.1 Query Recommendation's "Relationship and difference between NOT EXISTS and MINUS" section for details.

Searching Further in the Data

All the data we've queried so far would fit easily into a nice, simple table. This next version of our sample data adds new data that, if we stored this all in a relational database, would go into two other tables: one about courses being offered and another about who took which courses.

```
# filename: ex069.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

# People

d:i0432 ab:firstName "Richard" ;
        ab:lastName  "Mutt" ;
        ab:email     "richard49@hotmail.com" .

d:i9771 ab:firstName "Cindy" ;
        ab:lastName  "Marshall" ;
        ab:email     "cindym@gmail.com" .

d:i8301 ab:firstName "Craig" ;
        ab:lastName  "Ellis" ;
        ab:email     "c.ellis@usairwaysgroup.com" .

# Courses

d:course34 ab:courseTitle "Modeling Data with OWL" .
d:course71 ab:courseTitle "Enhancing Websites with RDFa" .
d:course59 ab:courseTitle "Using SPARQL with non-RDF Data" .
d:course85 ab:courseTitle "Updating Data with SPARQL" .

# Who's taking which courses
```

```
d:i8301 ab:takingCourse d:course59 .
d:i9771 ab:takingCourse d:course34 .
d:i0432 ab:takingCourse d:course85 .
d:i0432 ab:takingCourse d:course59 .
d:i9771 ab:takingCourse d:course59 .
```

In a relational database, each row of each table would need a unique identifier within that table so that you could cross-reference between the tables to find out, for example, who took which courses and the names of those courses. RDF data has this built in: each triple's subject is a unique identifier for that resource. Because of this, in the RDF version of the people and courses data, the subject of each triple about a person (for example, `d:i0432`) is the unique identifier for that person, and the subject of each triple about a course (for example, `d:course34`) is the unique identifier for that course. A SPARQL version of what relational databases developers call a "join," or a combination of data from multiple "tables," is very simple:

```
# filename: ex070.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?last ?first ?courseName
WHERE
{
  ?s ab:firstName ?first ;
     ab:lastName ?last ;
     ab:takingCourse ?course .

  ?course ab:courseTitle ?courseName .
}
```

This query uses no new bits of SPARQL that we haven't seen before. One technique that we first saw in ex048.rq, when we were looking for the names of albums produced by Timbaland, is the use of the same variable in the object position of one triple pattern and the subject position of another. In ex070.rq above, when the query processor looks for the course that a student is taking, it assigns the course's identifying URI to the `?course` variable, and then it looks for an `ab:courseTitle` value for that course and assigns it to the `?courseName` variable. This is a very common way to link up different sets of data with SPARQL.

The object of an RDF triple can be a literal string value or a URI, and a string value can be easier to read, but a URI makes it easier to link that data with other data. The last few triples in the ex069.ttl data above reference courses using their URIs, and the ex070.rq query uses these to look up their more readable `ab:courseTitle` values for the output. When creating RDF data, you can always give the URI an `rdfs:label` value (or something more specialized, like the `ab:courseTitle` values used above) for use by queries that want a more readable representation of that resource.

As with a relational database, the information about who took what course links person IDs with course IDs, and the ex070.rq query links those IDs to the more readable labels, showing us that Cindy and Richard are taking two courses and Craig is taking one:

```
-------------------------------------------------------------
| last       | first     | courseName                       |
=============================================================
| "Ellis"    | "Craig"   | "Using SPARQL with non-RDF Data" |
| "Marshall" | "Cindy"   | "Using SPARQL with non-RDF Data" |
| "Marshall" | "Cindy"   | "Modeling Data with OWL"         |
| "Mutt"     | "Richard" | "Using SPARQL with non-RDF Data" |
| "Mutt"     | "Richard" | "Updating Data with SPARQL"      |
-------------------------------------------------------------
```

If you split the ex069.ttl dataset's triples about people, available courses, and who took which courses into three different files instead of one, what would be different about the ex070.rq query? *Absolutely nothing.* If this data was in files named ex072.ttl, ex073.ttl, and ex368.ttl, the command line that called ARQ would be a little different, because it would have to name the three data files, but the query itself would need no changes at all:

```
arq --query ex070.rq --data ex072.ttl --data ex073.ttl --data ex368.ttl
```

The data queried in this example is an artificially small example of different datasets to link. A typical relational database would have two or more tables of hundreds or even thousands of rows that you could link up with an SQL query, but these tables must be defined together as part of the same database. A great strength of RDF and SPARQL is that you can do this with datasets from completely different places assembled by people unaware of each others' work, as long as there are resource URIs in one dataset that can line up with resource URIs in another. This is why it's so valuable to use existing URIs to represent people, places and things in your data: because it makes it easier for people to hook that data up with other data. (If they don't line up nicely, a few function calls may be all you need to transform some values to line them up better.)

Another way to tell a SPARQL query to look further in the data is with property paths, which let you express more extensive patterns to look for by just adding a little to the predicate part of a triple pattern.

1.1 Alert

Property paths are completely new for SPARQL 1.1.

It's easiest to see the power of property paths by looking at some examples. For sample data, imagine that Richard Mutt published a paper that we'll call Paper A in a prestigious academic journal. Cindy Marshall later published Paper B, which cited Richard's Paper A. Craig Ellis also cited Paper A in his classic work "Paper C." Over time, others wrote papers that cited Richard, Cindy, and Craig, forming the citation pattern shown in Figure 3-3.

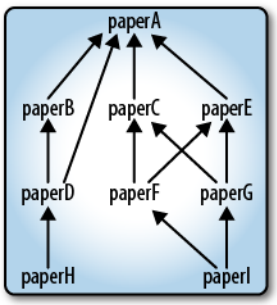


Figure 3-3. Which papers cite which papers in ex074.ttl data

The ex074.ttl dataset has data about which papers cited which. We'll see how property paths let us find out interesting things about the citation patterns with some very brief queries. Authors' names are omitted to keep the example short:

```
# filename: ex074.ttl

@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix c: <http://learningsparql.com/ns/citations#> .
@prefix : <http://learningsparql.com/ns/papers#> .

:paperA dc:title "Paper A" .

:paperB rdfs:label "Paper B" ;
        c:cites :paperA .

:paperC c:cites :paperA .

:paperD c:cites :paperA , :paperB .

:paperE c:cites :paperA .

:paperF c:cites :paperC , :paperE .

:paperG c:cites :paperC , :paperE .

:paperH c:cites :paperD .

:paperI c:cites :paperF , :paperG .
```

Remember, in N3 and Turtle and SPARQL, a comma means "the next triple has the same subject and predicate as the last one and the following object," so in ex074.ttl below, the first comma means that the triple after `:paperD c:cites :paperA` is `:paperD c:cites :paperB`.

For a start, note that the title of `:paperA` is expressed as a `dc:title` property and the title of `:paperB` is expressed as an `rdfs:label` property; perhaps this data is the result of merging data from two sources that used different conventions to identify paper titles. This isn't a problem—a simple query can bind either to the `?title` variable with one triple pattern:

```
# filename: ex075.rq

PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX : <http://learningsparql.com/ns/papers#>

SELECT ?s ?title
WHERE { ?s (dc:title | rdfs:label) ?title . }
```

The parentheses in query ex075.rq above don't affect the behavior of the query but make it a little easier to read.

The result of running this query on the citation data isn't especially exciting, but it demonstrates how this very brief query can do something that would have required a few OPTIONAL statements in SPARQL 1.0. It also demonstrates a nice strategy for working with data whose property names don't line up as clearly as you might wish.

```
-----------------------
| s       | title     |
=======================
| :paperB | "Paper B" |
| :paperA | "Paper A" |
-----------------------
```

It's easy enough with what we already know about SPARQL to ask which papers cited paper A. The following does this, and would tell us that that papers B, C, D, and E did so.

```
# filename: ex077.rq

PREFIX : <http://learningsparql.com/ns/papers#>
PREFIX c: <http://learningsparql.com/ns/citations#>

SELECT ?s
WHERE { ?s c:cites :paperA . }
```

Using several symbols that may be familiar from regular expression syntax in programming languages and utilities such as Perl and grep (for example, the pipe symbol in the ex075.rq query above) property paths let you say things like "and keep looking for more." Simply adding a plus sign to the most recent query tells the query processor to look for papers that cite paper A, and papers that cite those, and papers that cite those, until it runs out of papers citing this tree of papers:

```
# filename: ex078.rq

PREFIX : <http://learningsparql.com/ns/papers#>
PREFIX c: <http://learningsparql.com/ns/citations#>

SELECT ?s
WHERE { ?s c:cites+ :paperA . }
```

The result of running this query shows us how influential paper A was:

```
-----------
| s       |
===========
| :paperE |
| :paperG |
| :paperI |
| :paperF |
| :paperI |
| :paperD |
| :paperH |
| :paperC |
| :paperG |
| :paperI |
| :paperF |
| :paperI |
| :paperB |
| :paperD |
| :paperH |
-----------
```

Because this query asked for both direct and indirect links, papers with multiple pathways to paper A appear multiple times in the result. We'll see how to remove duplicate answers in the next section.

As with regular expressions, the plus sign means "one or more." You could use an asterisk instead, which in this case would mean "zero or more links." You can also be much more specific, asking for papers that are exactly three links away (that is, asking for papers that cited papers that cited papers that cited paper A):

```
# filename: ex080.rq

PREFIX : <http://learningsparql.com/ns/papers#>
PREFIX c: <http://learningsparql.com/ns/citations#>

SELECT ?s
WHERE { ?s c:cites{3} :paperA . }
```

The result has some repeats, because there are four ways to find a three-link `c:cites` path from paper I to paper A:

```
-----------
| s       |
===========
| :paperI |
| :paperI |
| :paperI |
| :paperI |
| :paperH |
-----------
```

Here's another way to do the same thing. It's not quite as terse, but it demonstrates why property paths are called paths: you can use them to lay out a series of steps separated by slashes, similarly to the way an XML XPath expression or a Linux or Windows directory pathname does.

```
# filename: ex082.rq

PREFIX : <http://learningsparql.com/ns/papers#>
PREFIX c: <http://learningsparql.com/ns/citations#>

SELECT ?s
WHERE { ?s c:cites/c:cites/c:cites :paperA . }
```

Inverse property paths let you flip the relationship described by a triple's predicate. For example, the following query does the same thing as the ex077.rq query earlier: it lists the papers that cite paper A.

```
# filename: ex083.rq

PREFIX : <http://learningsparql.com/ns/papers#>
PREFIX c: <http://learningsparql.com/ns/citations#>

SELECT ?s
WHERE { :paperA ^c:cites ?s }
```

Unlike ex077.rq, this query puts `:paperA` in the triple pattern's subject position and the `?s` variable in the object position. It's still looking for papers that cite paper A, and not the other way around, because the ^ operator at the beginning of the `c:cites` property tells the query processor that we want the inverse of this property.

So far, the inverse property path operator only lets us do something that we could do before but with the details specified in a different order. You can also use SPARQL property path operators on the individual steps of a path, and that's when the ^ operator lets you do some interesting new things.

For example, the following asks for all the papers that cite papers cited by paper F. In other words, it asks the question "which papers also cite the papers that paper F cites?"

```
# filename: ex084.rq

PREFIX : <http://learningsparql.com/ns/papers#>
PREFIX c: <http://learningsparql.com/ns/citations#>

SELECT ?s
WHERE
{
  ?s c:cites/^c:cites :paperF .
  FILTER(?s != :paperF)
}
```

The answer is paper G, which makes sense when you look at the diagram in Figure 3-3:

```
-----------
| s       |
===========
| :paperG |
| :paperG |
-----------
```

A network of paper citations is one way to use property paths. When you think about social networking, or computer networking, or any kind of networking, you'll see that property paths give you some nice new ways to ask about patterns in all kinds of datasets out there.

Searching with Blank Nodes

In the last chapter, we learned that although blank nodes have no permanent identity, we can use them to group together other values. For example, in the following, the person represented by the prefixed name `ab:i0432` has an address value of `_:b1`. The underscore tells us that the part after the colon is insignificant; the important thing is that the same resource has an `ab:postalCode` value of "49345", an `ab:city` value of "Springfield", an `ab:streetAddress` value of "32 Main St.", and an `ab:region` value of "Connecticut". In other words, Richard's address has these values.

```
# filename: ex041.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .

ab:i0432 ab:firstName    "Richard" ;
         ab:lastName     "Mutt" ;
         ab:homeTel      "(229) 276-5135" ;
         ab:email        "richard49@hotmail.com" ;
         ab:address      _:b1 .

_:b1     ab:postalCode   "49345" ;
         ab:city         "Springfield" ;
         ab:streetAddress "32 Main St." ;
         ab:region       "Connecticut" .
```

This simple query asks about the `ab:address` value:

```
# filename: ex086.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?addressVal
WHERE { ?s ab:address ?addressVal }
```

The result highlights the key thing to remember about blank nodes: that any name assigned to one is temporary, and that a processor doesn't have to worry about the temporary name as long as it remembers which nodes it connects to. In this case, it has a different name in the output (`_:b0`) from what it has in the input (`_:b1`) .

```
--------------
| addressVal |
==============
| _:b0       |
--------------
```

In a real query that references blank nodes, you'd use variables to reference them as connections between triples (just like you have with so many other triple subjects before) and just not ask for the values of those variables in the final query results—that is, you wouldn't include the names of the variables that represent the blank nodes in the SELECT list. For example:

```
# filename: ex088.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?firstName ?lastName ?streetAddress ?city ?region ?postalCode
WHERE
{
  ?s ab:firstName ?firstName ;
     ab:lastName ?lastName ;
     ab:address ?address .

  ?address ab:postalCode ?postalCode ;
           ab:city ?city ;
           ab:streetAddress ?streetAddress ;
           ab:region ?region .
}
```

Compared with queries that don't reference blank nodes, the query has nothing unusual, and neither does the result when run with the same ex041.ttl dataset:

```
-------------------------------------------------------------------------------------
| firstName | lastName | streetAddress | city         | region        | postalCode |
=====================================================================================
| "Richard" | "Mutt"   | "32 Main St." | "Springfield" | "Connecticut" | "49345"    |
-------------------------------------------------------------------------------------
```

To summarize, you can use a variable to reference blank nodes in RDF data just like you can use one to reference any other nodes, and doing so is useful for finding connections between other nodes, but there's no reason to ask for the values of any variables standing in for blank nodes.

Eliminating Redundant Output

Like it does in SQL, the DISTINCT keyword lets you tell the SPARQL processor "don't show me duplicate answers." For example, the following query, without the DISTINCT keyword, will show you the predicate of every triple in a dataset:

```
# filename: ex090.rq

SELECT ?p
WHERE
{ ?s ?p ?o . }
```

Here's the result when running it with the ex069.ttl dataset we saw earlier, which lists people and the courses they're taking:

```
-----------------------------------------------------------
| p                                                       |
===========================================================
| <http://learningsparql.com/ns/addressbook#courseTitle>  |
| <http://learningsparql.com/ns/addressbook#courseTitle>  |
| <http://learningsparql.com/ns/addressbook#courseTitle>  |
| <http://learningsparql.com/ns/addressbook#takingCourse> |
| <http://learningsparql.com/ns/addressbook#email>        |
| <http://learningsparql.com/ns/addressbook#lastName>     |
| <http://learningsparql.com/ns/addressbook#firstName>    |
| <http://learningsparql.com/ns/addressbook#takingCourse> |
| <http://learningsparql.com/ns/addressbook#takingCourse> |
| <http://learningsparql.com/ns/addressbook#email>        |
| <http://learningsparql.com/ns/addressbook#lastName>     |
| <http://learningsparql.com/ns/addressbook#firstName>    |
| <http://learningsparql.com/ns/addressbook#takingCourse> |
| <http://learningsparql.com/ns/addressbook#takingCourse> |
| <http://learningsparql.com/ns/addressbook#email>        |
| <http://learningsparql.com/ns/addressbook#lastName>     |
| <http://learningsparql.com/ns/addressbook#firstName>    |
| <http://learningsparql.com/ns/addressbook#courseTitle>  |
-----------------------------------------------------------
```

It's not a very useful query, especially for a dataset that's bigger than a few triples, because there's a lot of clutter in the output. With the DISTINCT keyword, however, it's a very useful query—in fact, it's often the first query I execute against a new dataset, because it tells me what kind of data I will find there:

```
# filename: ex092.rq

SELECT DISTINCT ?p
WHERE
{ ?s ?p ?o . }
```

Running ex092.rq with the same dataset gives us this output:

```
----------------------------------------------------------
| p                                                      |
==========================================================
| <http://www.w3.org/2000/01/rdf-schema#label>           |
| <http://learningsparql.com/ns/addressbook#takingClass> |
| <http://learningsparql.com/ns/addressbook#email>       |
| <http://learningsparql.com/ns/addressbook#lastName>    |
| <http://learningsparql.com/ns/addressbook#firstName>   |
----------------------------------------------------------
```

The result of this query is like a simplistic little schema, because it tells you what data is being tracked in this dataset. It can guide your subsequent queries as you explore a dataset, whether someone carefully designed a schema for the data in advance or the data is an ad hoc accumulation of data with no unifying schema. And, ex069.ttl wasn't very big—the value of the DISTINCT keyword will be even clearer with more realistically sized datasets.

How could we ask the same dataset which employees are taking courses? Without the DISTINCT keyword, the following query would list the first and last name of the student for every triple with `ab:takingCourse` as its predicate.

```
# filename: ex094.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT DISTINCT ?first ?last
WHERE
{
  ?s ab:takingCourse ?class ;
     ab:firstName ?first ;
     ab:lastName ?last .
}
```

Two of these triples have `d:i9771` as a subject, and two have `d:i0432`, so without the DISTINCT keyword in the query, Cindy Marshall and Richard Mutt would each be listed twice:

```
--------------------------
| first     | last       |
==========================
| "Craig"   | "Ellis"    |
| "Cindy"   | "Marshall" |
| "Cindy"   | "Marshall" |
| "Richard" | "Mutt"     |
| "Richard" | "Mutt"     |
--------------------------
```

With the DISTINCT keyword, the query lists the name of each person taking a course with no repeats:

```
--------------------------
| first     | last       |
==========================
| "Craig"   | "Ellis"    |
| "Cindy"   | "Marshall" |
| "Richard" | "Mutt"     |
--------------------------
```

The DISTINCT keyword adds no complexity to the structure of your query. You'll often find yourself writing a query without even thinking about this keyword, then seeing repeated values in the result, and then adding DISTINCT after the SELECT keyword to get more manageable results.

Combining Different Search Conditions

SPARQL's UNION keyword lets you specify multiple different graph patterns and then ask for a combination of all the data that fits any of those patterns. Compare Figure 3-4 diagram with Figure 1-1 near the beginning of Chapter 1.

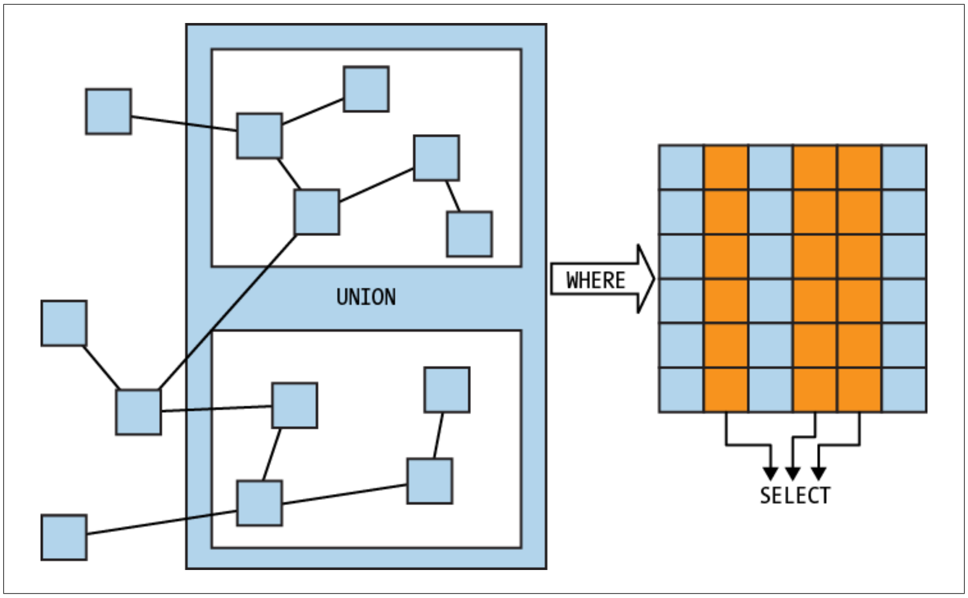


Figure 3-4. The UNION keyword lets WHERE clause grab two sets of data

For example, with our ex069.ttl data that listed people, courses, and who took which course, we could list the people and the courses with this query:

```
# filename: ex098.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d:  <http://learningsparql.com/ns/data#>

SELECT *
WHERE
{
       { ?person ab:firstName ?first ; ab:lastName ?last . }

       UNION

       { ?course ab:courseTitle ?courseName . }

}
```

The result has columns for each variable filled out for the names and the courses:

```
------------------------------------------------------------------------------------
| person  | first      | last        | course      | courseName                     |
==================================================================================
| d:i8301 | "Craig"   | "Ellis"    |            |                                 |
| d:i9771 | "Cindy"   | "Marshall" |            |                                 |
| d:i0432 | "Richard" | "Mutt"     |            |                                 |
|         |           |            | d:course85 | "Updating Data with SPARQL"     |
|         |           |            | d:course59 | "Using SPARQL with non-RDF Data" |
|         |           |            | d:course71 | "Enhancing Websites with RDFa"  |
|         |           |            | d:course34 | "Modeling Data with OWL"        |
----------------------------------------------------------------------------------
```

It's not a particularly useful example, but it demonstrates how UNION can let a query pull two sets of triples without specifying any connection between the sets.

Why does query ex098.rq declare a `d:` prefix that it doesn't use? As the query results show, some SPARQL processors use declared prefixes in their results instead of writing out the entire URI for every resource, which can make the data easier to read and easier to fit on a page.

Our next example is a bit more useful, using UNION to retrieve two overlapping sets of data. Imagine that our sample address book data stored information about musical instruments that each person plays, and we want to retrieve the first names, last names, and instrument names of the horn players.

```
# filename: ex100.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName  "Richard" ;
        ab:lastName   "Mutt" ;
        ab:instrument "sax" ;
        ab:instrument "clarinet" .

d:i9771 ab:firstName  "Cindy" ;
        ab:lastName   "Marshall" ;
        ab:instrument "drums" .

d:i8301 ab:firstName  "Craig" ;
        ab:lastName   "Ellis" ;
        ab:instrument "trumpet" .
```

The query in ex101.rq has a graph pattern to retrieve the first name, last name, and instrument names of any trumpet players and the same information about any sax players.

```
# filename: ex101.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last ?instrument
WHERE
{
  { ?person ab:firstName ?first ;
            ab:lastName ?last ;
            ab:instrument "trumpet" ;
            ab:instrument ?instrument .
  }

  UNION

  { ?person ab:firstName ?first ;
            ab:lastName ?last ;
            ab:instrument "sax" ;
            ab:instrument ?instrument .
  }

}
```

Why use the `?instrument` variable to ask for the instrument name if we already know it in each graph pattern? Because, being bound to a variable, we can then use it to include the specific instrument name next to each horn player's name.

The results show who play these instruments. For each person who plays the sax or trumpet, it also lists any other instruments they play, so it also shows that sax player Richard plays the clarinet, because that also matched the last triple pattern:

```
------------------------------------
| first     | last    | instrument |
====================================
| "Craig"   | "Ellis" | "trumpet"  |
| "Richard" | "Mutt"  | "clarinet" |
| "Richard" | "Mutt"  | "sax"      |
------------------------------------
```

That query had some unnecessary redundancy in it. We can fix this by using the UNION keyword to unite smaller graph patterns and add them to the part that the last query's two patterns had in common:

```
# filename: ex103.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?first ?last ?instrument
WHERE
{
    ?person ab:firstName ?first ;
            ab:lastName ?last ;
            ab:instrument ?instrument .

    { ?person ab:instrument "sax" . }

    UNION

    { ?person ab:instrument "trumpet" . }
}
```

These example used the UNION keyword to unite only two graph patterns, but you can link as many as you like by repeating the keyword before each graph pattern that you want to connect with the others.

FILTERing Data Based on Conditions

In Chapter 1 we saw the following query. It has the most flexible possible triple pattern, because with variables in all three positions, all the triples that could possibly be in any dataset's default graph (that is, all the dataset triples that aren't in named graphs; we'll learn about these in "Querying Named Graphs" on page 78) will match against it. It only retrieved one triple, though, because the FILTER expression specified that we only wanted triples with the string "yahoo" in their object:

```
# filename: ex021.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT *
WHERE
{
  ?s ?p ?o .
  FILTER (regex(?o, "yahoo","i"))
}
```

FILTER takes a single argument. The expression you put there can be as complex as you want, as long as it returns a boolean value. The expression above is a call to the `regex()` function, which we'll learn about in Chapter 5. In the ex021.rq query, it checks whether the `?o` value has "yahoo" as a substring, without worrying whether it's in upper- or lowercase.

The FILTER argument can also be very simple. For example, let's say we say we have some data tracking the cost and location of a few items.

```
# filename: ex104.ttl

@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:item432 dm:cost 8 ;
          dm:location <http://dbpedia.org/resource/Boston> .
d:item857 dm:cost 12 ;
          dm:location <http://dbpedia.org/resource/Montreal> .
d:item693 dm:cost 10 ;
          dm:location "Heidelberg" .
d:item126 dm:cost 5 ;
          dm:location <http://dbpedia.org/resource/Lisbon> .
```

The FILTER argument doesn't need to be a function call. It can be a simple comparison, which also returns a boolean value:

```
# filename: ex105.rq

PREFIX dm: <http://learningsparql.com/ns/demo#>

SELECT ?s ?cost
WHERE
{
  ?s dm:cost ?cost .
  FILTER (?cost < 10)
}
```

This query pulls the items that cost less than 10:

```
------------------------------------------------------
| s                                           | cost |
======================================================
| <http://learningsparql.com/ns/data#item126> | 5    |
| <http://learningsparql.com/ns/data#item432> | 8    |
------------------------------------------------------
```

The two `?s` values have full URIs instead of being prefixed names with prefixes because the query processor didn't know about the `d:` prefixes in the input data. When the RDF parser read in the input data, it mapped those prefixes to the appropriate namespace URIs before handing off the data to the query processor, and the query didn't define any prefix for that URI.

FILTER is also helpful for data cleanup. For example, while a triple's object can be a string or a URI, a URI is better, because you can use it to link the triple with other triples. The item cost and location data above has URIs for most of its location values, but not all of them.

The next query lists the ones that aren't URIs. SPARQL's `isURI()` function returns a boolean `true` if its argument is a proper URI. The exclamation point functions as a "not" operator, so that the expression `!(isURI(?city))` will return a boolean `true` if the value of `?city` is *not* a proper URI:

```
# filename: ex107.rq

PREFIX dm: <http://learningsparql.com/ns/demo#>

SELECT ?s ?city
WHERE
{
  ?s dm:location ?city .
  FILTER (!(isURI(?city)))
}
```

The result shows us where our data needs some cleanup:

```
---------------------------------------------------------------
| s                                          | city          |
===============================================================
| <http://learningsparql.com/ns/data#item693> | "Heidelberg" |
---------------------------------------------------------------
```

As we saw in "Using the Labels Provided by DBpedia" on page 48, FILTER is also great for pulling values in a particular language from data that stores values in multiple languages.

1.1 Alert

The more you learn about SPARQL functions, the more you'll be able to do with the FILTER keyword, so the new functions available in SPARQL 1.1 extend FILTER's power even more.

SPARQL 1.1's new IN keyword lets you use an enumerated list as part of a query. For example, the following query asks for data where the location value is either `db:Montreal` or `db:Lisbon`:

```
# filename: ex109.rq

PREFIX dm:  <http://learningsparql.com/ns/demo#>
PREFIX db: <http://dbpedia.org/resource/>

SELECT ?s ?cost ?location
WHERE
{
  ?s dm:location ?location ;
     dm:cost ?cost .
  FILTER (?location IN (db:Montreal, db:Lisbon)) .
}
```

This list has only two values inside the parentheses after the IN keyword, but can have as many as you like. Here's the result of running ex109.rq with the ex104.ttl dataset:

```
 ---------------------------------------------------------------------
| s                                           | cost | location    |
=====================================================================
| <http://learningsparql.com/ns/data#item857> | 12   | db:Montreal |
| <http://learningsparql.com/ns/data#item126> | 5    | db:Lisbon   |
---------------------------------------------------------------------
```

The list doesn't have to be prefixed names. It can be quoted strings or any other kind of data. For example, you could query the data using cost values like this:

```
# filename: ex111.rq

PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX db: <http://dbpedia.org/resource/>

SELECT ?s ?cost ?location
WHERE
```

```
{
  ?s dm:location ?location ;
     dm:cost ?cost .
  FILTER (?cost IN (8, 12, 10)) .
}
```

You can also add the keyword NOT before IN,

```
# filename: ex112.rq

PREFIX dm:  <http://learningsparql.com/ns/demo#>
PREFIX db: <http://dbpedia.org/resource/>

SELECT ?s ?cost ?location
WHERE
{
  ?s dm:location ?location ;
     dm:cost ?cost .
  FILTER (?location NOT IN (db:Montreal, db:Lisbon)) .
}
```

and get the data for everything where `?location` is not in the list after the IN keyword:

```
 ---------------------------------------------------------------------
| s                                         | cost | location      |
=====================================================================
| <http://learningsparql.com/ns/data#item693> | 10   | "Heidelberg" |
| <http://learningsparql.com/ns/data#item432> | 8    | db:Boston    |
---------------------------------------------------------------------
```

Retrieving a Specific Number of Results

If you sent the following query to DBpedia, you'd be asking too much of it:

```
# filename: ex114.rq

SELECT ?label
WHERE
{ ?s rdfs:label ?label . }
```

It's a simple little query. (The problem is not that the `rdfs:` prefix is not declared in the query; for DBpedia queries entered using the SNORQL query form, it's predeclared.) However, DBpedia has quite a few `rdfs:label` values—maybe millions—and this query asks for all of them. It will either time out and not give you any values, or it will give a limited number.

You can set your own limit with the LIMIT keyword. We'll try it out with the following little dataset:

```
# filename: ex115.ttl

@prefix d:    <http://learningsparql.com/ns/data#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

d:one   rdfs:label "one" .
```

```
d:two   rdfs:label "two" .
d:three rdfs:label "three" .
d:four  rdfs:label "four" .
d:five  rdfs:label "five" .
d:six   rdfs:label "six" .
```

This next query tells the SPARQL processor that no matter how many triples match the pattern shown, we want no more than two results:

```
# filename: ex116.rq

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

SELECT ?label
WHERE
{ ?s rdfs:label ?label . }
LIMIT 2
```

Note that the LIMIT keyword is outside of the curly braces, not inside of them.

A set of stored triples has no order, so this query won't necessarily retrieve the first two results that you see in the sample data above. It might for you, but for me it got values from the last two, with their order reversed:

```
----------
| label  |
==========
| "six"  |
| "five" |
----------
```

As we'll see in "Sorting, Aggregating, Finding the Biggest and Smallest and..." on page 88, when you tell the SPARQL processor to sort your data, LIMIT will retrieve the first results of the sorted data.

The OFFSET keyword tells the processor to skip a given number of results before picking some to return. This next query tells it to skip 3 of the 6 results that we know would appear without the OFFSET keyword:

```
# filename: ex118.rq

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

SELECT ?label
WHERE
{ ?s rdfs:label ?label . }
OFFSET 3
```

This gives us three results:

```
-----------
| label   |
===========
| "three" |
| "two"   |
| "one"   |
-----------
```

As with the LIMIT keyword, if you don't tell the query engine to sort the results, the ones it chooses to return may seem fairly random. OFFSET is actually used with LIMIT quite often to pull different handfuls of triple from out of a larger collection. For example, the following adds a LIMIT keyword to the previous query:

```
# filename: ex120.rq

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

SELECT ?label
WHERE
{ ?s rdfs:label ?label . }
OFFSET 3
LIMIT 1
```

When run with the same output, we get only the one value that the query asks for:

```
-----------
| label   |
===========
| "three" |
-----------
```

Querying Named Graphs

A dataset can include sets of triples that have names assigned to them to make it easier to manage those sets. (For more information on this, see "Named Graphs" on page 35 of Chapter 2.) For example, perhaps a given set of triples came from a specific source on a particular date and should be replaced by the next batch to come from that same source; the ability to refer to that set with a single name makes this possible. We'll learn more about making such replacements in Chapter 6. Here, we'll learn how to use graph names in your queries.

First, though, let's review something we learned in Chapter 1. We saw there that instead of telling the query processor what data we want to query separately from the query itself (for example, by including the name of a data file on the ARQ command line) we can specify the data to query right in the query itself with the FROM keyword. Using this, your query can specify as many graphs of triples as you'd like for the query.

For example, let's say we have a data file of new students to add to the ex069.ttl file we used earlier. That one had data about three people, four courses that were being offered, and who took which course. The new file has data about two more students:

```
# filename: ex122.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:i5433 ab:firstName "Katherine" ;
        ab:lastName  "Duncan" ;
        ab:email     "katherine.duncan@elpaso.com" .

d:i2194 ab:firstName "Bradley" ;
        ab:lastName  "Perry" ;
        ab:email     "bradley.perry@corning.com" .
```

The following query uses the FROM keyword to specify that it wants to look for data in both the old ex069.ttl file and in the new ex122.ttl file:

```
# filename: ex123.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?email
FROM <ex069.ttl>
FROM <ex122.ttl>
WHERE
{ ?s ab:email ?email . }
```

The ex069.ttl dataset has three email addresses and ex122.ttl has two, so this query returns a total of five:

```
---------------------------------
| email                         |
=================================
| "bradley.perry@corning.com"   |
| "katherine.duncan@elpaso.com" |
| "c.ellis@usairwaysgroup.com"  |
| "cindym@gmail.com"            |
| "richard49@hotmail.com"       |
---------------------------------
```

All of the datasets that you specify this way (or specify outside of the query, like on ARQ's command line) are added together to form what's called the *default graph*. This refers to all the triples accessible to the query that aren't part of any named graphs. Default graphs are the only kind we've seen so far in this book.

If FROM is a way to say "add the triples from the following graph to the default dataset that I'm going to query," then FROM NAMED is a way to say "I'll be querying data from this particular graph, but don't add its triples to the default graph—when I want data from this graph, I'll mention it by name." The SPARQL processor will remember the pairing of that graph name and that graph. As we'll see, we can even query the list of pairings.

Of course, because we're talking about RDF and SPARQL, a named graph's name is a URI.

The ability to assign these names is an important feature of triplestores, because as you add, update, and remove datasets, you must be able to identify the sets that you're adding, updating, and removing. When you have an RDF file sitting on a disk in any of the popular RDF serializations, there's no standard way to identify a subset of that file's triples as belonging to a particular named graph, so to demonstrate the querying of named graphs in this chapter, I used the ARQ convention of using a URI for a file's location as the name of each named graph. In Chapter 6, we'll see the standardized SPARQL 1.1 way to create and delete named graphs in a triplestore and how to add and replace triples in them.

Above we saw that the file ex122.ttl had data about new students. Here is ex125.ttl, which has data about new courses being offered:

```
# filename: ex125.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

ab:course42 ab:courseTitle "Combining Public and Private RDF Data" .
ab:course24 ab:courseTitle "Using Named Graphs" .
```

In the following examples, the named graphs' names are relative URIs, so when ARQ sees <*ex125.ttl*>, it will look for the ex125.ttl file in the directory where the query file is stored. If I stored a copy of that file in the examples directory of www.learningsparql.com and referenced <*http://www.learningsparql.com/examples/ex125.ttl*> as a named graph in the query, ARQ would retrieve the file's triples from that location and use them.

While ARQ treats graph names as pointers to actual files, a triplestore that maps user-specified graph names to triple sets will use those mappings to find the triple sets.

The following query loads our original ex069.ttl data about students, courses, and who took what courses into the default graph and then specifies the ex125.ttl file of data about new courses and the ex122.ttl file of data about new students as named graphs to reference in the query.

```
# filename: ex126.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?lname ?courseName
```

```
FROM <ex069.ttl>
FROM NAMED <ex125.ttl>
FROM NAMED <ex122.ttl> # unnecessary

WHERE
{
  { ?student ab:lastName ?lname }
  UNION
  { GRAPH <ex125.ttl> { ?course ab:courseTitle ?courseName } }
}
```

In addition to the phrase FROM NAMED, our other new keyword is GRAPH, which a query uses to reference data from a specific named graph. Here's the result of running this query:

```
--------------------------------------------------------
| lname      | courseName                              |
========================================================
| "Ellis"    |                                         |
| "Marshall" |                                         |
| "Mutt"     |                                         |
|            | "Using Named Graphs"                    |
|            | "Combining Public and Private RDF Data" |
--------------------------------------------------------
```

The query pulls a union of two sets of triples out of the specified data—the triples from the default graph with `ab:lastName` as a predicate and the triples from ex125.ttl with `ab:courseTitle` as a predicate—and then selects the `?lname` and `?courseName` values from the combined set of triples. There are two important things to note about the results:

- Even though the ex069.ttl file that provided the triples for the default graph has triples that would match the `?course ab:courseTitle ?courseName` pattern, they're not in the output, because the query only asked for triples that matched that pattern from the `<ex125.ttl>` named graph, and it only asked for triples that matched the pattern `?student ab:lastName ?lname` from the default graph.
- The ex122.ttl named graph had triples that would match the `?student ab:lastName ?lname` pattern, but none of its data showed up in the output because its triples were never added to the default graph with a `FROM <ex122.ttl>` expression. The `FROM NAMED <ex122.ttl>` expression essentially said "I'll be using a named graph with the name <ex122.ttl>," but the query never got around to actually using it—there's a `GRAPH <ex125.ttl>` part, but no `GRAPH <ex122.ttl>` part. I just added it to the query to show that FROM NAMED doesn't contribute triples to the query unless you actually use the named graph.

In the ex126.rq query, the GRAPH keyword points at a specific named graph, but we can use a variable instead and let the SPARQL processor look for graphs that fit the pattern. The following query checks all the named graphs it knows about for any triples that match the pattern `?s ?p ?o` (which will be all of them) and asks for the names of those graphs:

```
# filename: ex128.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?g
FROM NAMED <ex125.ttl>
FROM NAMED <ex122.ttl>
WHERE
{
  GRAPH ?g { ?s ?p ?o }
}
```

The answer lists ex122.ttl six times and ex125.ttl twice, because there's a result set row for each triple that the SPARQL processor found in those files to match the `?s ?p ?o` pattern:

```
---------------
| g           |
===============
| <ex122.ttl> |
| <ex122.ttl> |
| <ex122.ttl> |
| <ex122.ttl> |
| <ex122.ttl> |
| <ex122.ttl> |
| <ex125.ttl> |
| <ex125.ttl> |
---------------
```

If you add the DISTINCT keyword after SELECT in that query, it will only list each graph name once.

The ex128.rq query only specifies named graphs to query, with no default graph. If it did have a default graph, none of its triples would appear in the result, because the GRAPH keyword means that the graph pattern is only looking for triples from named graphs.

In a more realistic example of using a variable after the GRAPH keyword, we can use it to tell the SPARQL processor to retrieve a UNION of all `ab:courseTitle` values from the default graph and from all the named graphs without actually specifying any graph names after the GRAPH keyword:

```
# filename: ex130.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?courseName
FROM <ex069.ttl>
FROM NAMED <ex125.ttl>
FROM NAMED <ex122.ttl>
WHERE
{
  { ?course ab:courseTitle ?courseName }

  UNION

  {GRAPH ?g { ?course ab:courseTitle ?courseName } }

}
```

It finds the four `ab:classTitle` values from ex069.ttl and the two from ex125.ttl:

```
-------------------------------------------
| courseName                              |
===========================================
| "Updating Data with SPARQL"             |
| "Using SPARQL with non-RDF Data"        |
| "Enhancing Websites with RDFa"          |
| "Modeling Data with OWL"                |
| "Using Named Graphs"                    |
| "Combining Public and Private RDF Data" |
-------------------------------------------
```

Because graph names are URIs, you can use them as either the subject or the object of triples, which makes things more interesting. This means that a graph name can be a metadata value for something. For example, you can say that the image at *http://www.somepublisher.com/img/f43240.jpg* has metadata in its owner's metadata repository triplestore in the named graph*http://www.somepublisher.com/ns/m43240* with a triple like this:

```
# filename: ex132.ttl

@prefix images: <http://www.somepublisher.com/img/> .
@prefix isi: <http://www.isi.edu/ikcap/Wingse/fileOntology.owl#> .

images:f43240.jpg isi:hasMetadata <http://www.somepublisher.com/ns/m43240> .
```

The *http://www.somepublisher.com/ns/m43240* named graph might have triples like this:

```
# filename: ex133.ttl

@prefix images: <http://www.somepublisher.com/img/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

images:f43240.jpg dc:creator "Richard Mutt" ;
                  dc:title "Fountain" ;
                  dc:format "jpeg" .
```

To find an appropriate predicate for this example, I searched the Swoogle vocabulary directory, and it led me to the `hasMetadata` property in the *http://www.isi.edu/ikcap/Wingse/fileOntology.owl* vocabulary. For a production application, I would have searched a little more for other candidate vocabularies and then evaluated their popularity before I picked one whose property or properties I was going to use.

A graph name can also have its own metadata. The following dataset includes Dublin Core date and creator values for the two named graphs used in the above examples:

```
# filename: ex134.ttl

@prefix dc: <http://purl.org/dc/elements/1.1/> .

<ex125.ttl> dc:date "2011-09-23" ;
            dc:creator "Richard Mutt" .

<ex122.ttl> dc:date "2011-09-24" ;
            dc:creator "Richard Mutt" .
```

This next query asks for all the email addresses from the named graph that has a `dc:date` value of "2011-09-24" assigned to it:

```
# filename: ex135.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?graph ?email
FROM <ex134.ttl>
FROM NAMED <ex125.ttl>
FROM NAMED <ex122.ttl>
WHERE
{
  ?graph dc:date "2011-09-24" .
  { GRAPH ?graph { ?s ab:email ?email } }
}
```

Above I said that a query uses the GRAPH keyword to reference data from a specific named graph. I also said that FROM NAMED identifies a named graph that may be referenced by a GRAPH keyword. You may see SPARQL queries out there, however, that use the GRAPH keyword to reference named graphs that were not identified in a FROM NAMED clause first. How can they do this?

Some SPARQL processors—in particular, those that are part of a triplestore or a SPARQL endpoint—have some predefined named graphs that you don't need to identify in a FROM NAMED clause before referencing them with the GRAPH keyword.

The open source Sesame triplestore stores a default graph and any accompanying named graphs in what Sesame calls a repository. If you reference a named graph in that repository with the GRAPH keyword, Sesame will know what you're talking about without requiring that this graph be identified first with a FROM NAMED clause. Other triplestores and SPARQL endpoints may do this differently, because the W3C SPARQL Recommendations don't spell out the scope of what named graphs a SPARQL processor must know about outside of the FROM NAMED ones.

You can ask which ones a SPARQL processor knows about with a simple query:

```
# filename: ex136.rq

SELECT DISTINCT ?g
WHERE
{
  GRAPH ?g {?s ?p ?o }
}
```

I've seen processors that didn't provide any response to this query but still seemed to know about certain named graphs mentioned in other queries, so don't be too disappointed if you see no results. Just treat this as one more nice generic query that's helpful when exploring a new SPARQL processor or data source.

Queries in Your Queries

The ability to put queries inside of queries is known as the subqueries feature. This lets you break down a complex query into more easily manageable parts, and it also lets you combine information from different queries into a single answer set.

> **1.1 Alert**
>
> Subqueries are new for SPARQL 1.1.

Each subquery must be enclosed in its own set of curly braces. The following query, which you can test with the ex069.ttl data file, has one subquery to retrieve last name values and another to retrieve course title values, and the SELECT statement of the enclosing main query asks for `?lastName` and `?courseName` values retrieved by the subqueries:

```
# filename: ex137.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?lastName ?courseName
WHERE
{
  {
    SELECT ?lastName
    WHERE { ?student ab:lastName ?lastName . }
  }

  {
    SELECT ?courseName
    WHERE { ?course ab:courseTitle ?courseName . }
  }

}
```

It's a fairly artificial example, because the output that this query creates (every possible ?lastName-?courseName pairing) isn't very useful. We'll see some more productive examples of subqueries, though, in the next section.

Combining Values and Assigning Values to Variables

Once your SPARQL query pulls values out of a dataset, it can use these values in expressions that perform math and function calls. This lets your queries do even more with the data.

1.1 Alert

Using expressions to calculate new values and the variations on this described in this chapter are all SPARQL 1.1 features.

To experiment with the creation of expressions, we'll use some fake expense report data:

```
# filename: ex138.ttl

@prefix e: <http://learningsparql.com/ns/expenses#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:m40392 e:description "breakfast" ;
         e:date "2011-10-14T08:53" ;
         e:amount 6.53 .

d:m40393 e:description "lunch" ;
         e:date "2011-10-14T13:19" ;
         e:amount 11.13 .

d:m40394 e:description "dinner" ;
         e:date "2011-10-14T19:04" ;
         e:amount 28.30 .
```

Leaving the quotation marks off of numeric values tells the SPARQL processor to treat them as numbers and not strings. Chapter 5 describes this in greater detail.

The following query's WHERE clause plugs values into the ?meal, ?description, and ?amount variables as it goes through the ex138.ttl dataset, but the query's SELECT clause does a bit more with the ?amount value than just display it:

```
# filename: ex139.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>
```

```
SELECT ?description ?amount ((?amount * .2) AS ?tip)
       ((?amount + ?tip) AS ?total)
WHERE
{
  ?meal e:description ?description ;
        e:amount ?amount .
}
```

It will be easier to understand what it does if we look at the output first:

```
-----------------------------------------
| description | amount | tip   | total  |
=========================================
| "dinner"    | 28.30  | 5.660 | 33.960 |
| "lunch"     | 11.13  | 2.226 | 13.356 |
| "breakfast" | 6.53   | 1.306 | 7.836  |
-----------------------------------------
```

In addition to displaying the `?description` and `?amount` values, the SELECT clause multiplies the `?amount` value by .2 and uses the AS keyword to store the result in the variable `?tip`, which shows up as the third column of the search results. Then, the SELECT clause adds the `?amount` and `?tip` values together and stores the results in the variable `?total`, whose values appear in the fourth column of the query results.

As we'll see in Chapter 5, SPARQL 1.0 supports a few functions to manipulate data values, and SPARQL 1.1 adds many more. The following query of the same ex138.ttl data uses two string manipulation functions to create an uppercase version of the first three letters of the expense descriptions:

```
# filename: ex141.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT (UCASE(SUBSTR(?description,1,3))
       as ?mealCode) ?amount

WHERE
{
  ?meal e:description ?description ;
        e:amount ?amount .
}
```

The result has a column for the calculated `?mealCode` value and another for the `?amount` value:

```
---------------------
| mealCode | amount |
=====================
| "DIN"    | 28.30  |
| "LUN"    | 11.13  |
| "BRE"    | 6.53   |
---------------------
```

Performing complex calculations on multiple SELECT values can make for a pretty long SELECT statement. That's why I had to move the calculation of the `?total` value in ex139.rq to a new line. The following revision of this query gets the same result with the expression calculation moved to a subquery. This allows the main SELECT statement at the query's beginning to be much simpler:

```
# filename: ex143.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT ?mealCode ?amount
WHERE
{
  {
    SELECT ?meal (UCASE(SUBSTR(?description,1,3)) as ?mealCode)
    WHERE { ?meal e:description ?description . }
  }

  {
    SELECT ?meal ?amount
    WHERE { ?meal e:amount ?amount . }
  }
}
```

Another way to move the expression calculation away from the main SELECT clause is with the BIND keyword. This is the most concise alternative supported by the SPARQL 1.1 standard, because it lets you assign the expression value to the new variable in one line of code. The following example produces the same query results as the last two queries:

```
# filename: ex144.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT ?mealCode ?amount
WHERE
{
  ?meal e:description ?description ;
        e:amount ?amount .
  BIND (UCASE(SUBSTR(?description,1,3)) as ?mealCode)
}
```

Sorting, Aggregating, Finding the Biggest and Smallest and...

SPARQL lets you sort your data and use built-in functions to get more out of that data. To experiment with these features, we'll use an expanded version of the expense report data that we saw in the last section, but this time covering three days of meals:

```
# filename: ex145.ttl

@prefix e: <http://learningsparql.com/ns/expenses#> .
@prefix d: <http://learningsparql.com/ns/data#> .

d:m40392 e:description "breakfast" ;
         e:date "2011-10-14T08:53" ;
         e:amount 6.53 .

d:m40393 e:description "lunch" ;
         e:date "2011-10-14T13:19" ;
         e:amount 11.13 .

d:m40394 e:description "dinner" ;
         e:date "2011-10-14T19:04" ;
         e:amount 28.30 .

d:m40395 e:description "breakfast" ;
         e:date "2011-10-15T08:32" ;
         e:amount 4.32 .

d:m40396 e:description "lunch" ;
         e:date "2011-10-15T12:55" ;
         e:amount 9.45 .

d:m40397 e:description "dinner" ;
         e:date "2011-10-15T18:54" ;
         e:amount 31.45 .

d:m40398 e:description "breakfast" ;
         e:date "2011-10-16T09:05" ;
         e:amount 6.65 .

d:m40399 e:description "lunch" ;
         e:date "2011-10-16T13:24" ;
         e:amount 10.00 .

d:m40400 e:description "dinner" ;
         e:date "2011-10-16T19:44" ;
         e:amount 25.05 .
```

Sorting Data

SPARQL uses the phrase ORDER BY to sort data. This should look familiar to SQL users. The following query sorts the data using the values bound to the `?amount` variable:

```
# filename: ex146.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT ?description ?date ?amount
WHERE
{
  ?meal e:description ?description ;
        e:date ?date ;
        e:amount ?amount .
}

ORDER BY ?amount
```

Using the expense report data above, the result shows the expense data sorted from the smallest ?amount value to the largest:

```
---------------------------------------------
| description | date               | amount |
=============================================
| "breakfast" | "2011-10-15T08:32" | 4.32   |
| "breakfast" | "2011-10-14T08:53" | 6.53   |
| "breakfast" | "2011-10-16T09:05" | 6.65   |
| "lunch"     | "2011-10-15T12:55" | 9.45   |
| "lunch"     | "2011-10-16T13:24" | 10.00  |
| "lunch"     | "2011-10-14T13:19" | 11.13  |
| "dinner"    | "2011-10-16T19:44" | 25.05  |
| "dinner"    | "2011-10-14T19:04" | 28.30  |
| "dinner"    | "2011-10-15T18:54" | 31.45  |
---------------------------------------------
```

To sort the expense report data in descending order, wrap the sort key in the DESC() function:

```
# filename: ex148.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT ?description ?date ?amount
WHERE
{
  ?meal e:description ?description ;
        e:date ?date ;
        e:amount ?amount .
}

ORDER BY DESC(?amount)
```

We saw in "Data Typing" on page 30 that when the object of a Turtle triple is a number without quotation marks, the processor will treat it as an integer or a decimal number, depending on whether the figure has a decimal point. When the SORT keyword knows that it's sorting numbers, it treats them as amounts and not as strings, as we'll see in the next example.

To sort on multiple keys, separate the key value names by spaces. The following sorts the expense report data alphabetically by description and then by amount, with amounts shown from largest to smallest:

```
# filename: ex149.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT ?description ?date ?amount
WHERE
{
  ?meal e:description ?description ;
```

```
        e:date ?date ;
        e:amount ?amount .
}

ORDER BY ?description DESC(?amount)
```

Here is the result, with "breakfast," "dinner," and "lunch" listed alphabetically, and the amounts sorted in descending order within each meal category:

```
---------------------------------------------
| description | date                | amount |
=============================================
| "breakfast" | "2011-10-16T09:05" | 6.65   |
| "breakfast" | "2011-10-14T08:53" | 6.53   |
| "breakfast" | "2011-10-15T08:32" | 4.32   |
| "dinner"    | "2011-10-15T18:54" | 31.45  |
| "dinner"    | "2011-10-14T19:04" | 28.30  |
| "dinner"    | "2011-10-16T19:44" | 25.05  |
| "lunch"     | "2011-10-14T13:19" | 11.13  |
| "lunch"     | "2011-10-16T13:24" | 10.00  |
| "lunch"     | "2011-10-15T12:55" | 9.45   |
---------------------------------------------
```

Finding the Smallest, the Biggest, the Count, the Average...

The SPARQL 1.0 way to find the smallest, the biggest, or the alphabetically first or last value in the data retrieved from a dataset is to sort the data and then use the LIMIT keyword (described in "Retrieving a Specific Number of Results" on page 76) to only retrieve the first value. For example, the ex148.rq query above asked for expense report data sorted in descending order from the largest amount value to the smallest; adding LIMIT 1 to this query tells the SPARQL processor to only return the first of those values:

```
# filename: ex151.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT ?description ?date ?amount
WHERE
{
  ?meal e:description ?description ;
        e:date ?date ;
        e:amount ?amount .
}

ORDER BY DESC(?amount)
LIMIT 1
```

Running the query shows us the most expensive meal:

```
 ---------------------------------------------
| description | date                | amount |
=============================================
| "dinner"    | "2011-10-15T18:54" | 31.45  |
---------------------------------------------
```

Without the `DESC()` function on the sort key, the query would have displayed data for the least expensive meal, because with the data sorted in ascending order, that would have been first.

The `MAX()` function lets you find the largest amount with a much simpler query.

1.1 Alert

`MAX()` and the remaining functions described in this section are new in SPARQL 1.1.

Here's a simple example:

```
# filename: ex153.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT (MAX(?amount) as ?maxAmount)
WHERE { ?meal e:amount ?amount . }
```

This query's SELECT clause stores the maximum value bound to the `?amount` variable in the `?maxAmount` variable, and that's what the query engine returns:

```
-------------
| maxAmount |
=============
| 31.45     |
-------------
```

To find the description and date values associated with the maximum amount identified with the `MAX()` function, you'd have to find the maximum value in a subquery and separately ask for the data that goes with it, like this:

```
# filename: ex155.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT ?description ?date ?maxAmount
WHERE
{
  {
    SELECT (MAX(?amount) as ?maxAmount)
    WHERE { ?meal e:amount ?amount . }
  }
  {
    ?meal e:description ?description ;
          e:date ?date ;
          e:amount ?maxAmount .
  }
}
```

The SPARQL 1.0 ORDER BY with LIMIT 1 trick is actually simpler, but as we'll see, the `MAX()` function (and its partner the `MIN()` function, which finds the smallest value) is handy in other situations.

Substituting other new functions for `MAX()` in ex153.rq lets you do interesting things that have no SPARQL 1.0 equivalent. For example, the following finds the average cost of all the meals in the expense report data:

```
# filename: ex156.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT (AVG(?amount) as ?avgAmount)
WHERE { ?meal e:amount ?amount . }
```

ARQ calculates the average to quite a few decimal places, which you may not find with other SPARQL processors:

```
------------------------------
| avgAmount                  |
==============================
| 14.764444444444444444444444 |
------------------------------
```

Using the `SUM()` function in the same place would add up the values, and the `COUNT()` function would count how many values got bound to that variable—in other words, how many `e:amount` values were retrieved.

Another interesting function is `GROUP_CONCAT()`, which concatenates all the values bound to the variable, separated by a space or the delimiter that you specify in the optional second argument. The following stores all the expense values in the `?amountList` variable separated by commas:

```
# filename: ex158.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT (GROUP_CONCAT(?amount; SEPARATOR = ",") AS ?amountList)
WHERE { ?meal e:amount ?amount . }
```

The result (minus the hyphen, equals, and pipe characters that ARQ adds here for display) will be easy to import into a spreadsheet:

```
-------------------------------------------------------
| amountList                                          |
=======================================================
| "25.05,10.00,6.65,31.45,9.45,4.32,28.30,11.13,6.53" |
-------------------------------------------------------
```

Grouping Data and Finding Aggregate Values within Groups

Another feature that SPARQL inherited from SQL is the GROUP BY keyword phrase. This lets you group sets of data together to perform aggregate functions such as subtotal calculation on each group. For example, the following query tells the SPARQL processor to group the results together by `?description` values (that is, for the expense report data, to group breakfast values together, lunch values together, and dinner values together) and to then sum the `?amount` values for each group:

```
# filename: ex160.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT ?description (SUM(?amount) AS ?mealTotal)
WHERE
{
  ?meal e:description ?description ;
        e:amount ?amount .
}
GROUP BY ?description
```

Running this query with the ex145.ttl data gives us this result:

```
---------------------------
| description | mealTotal |
===========================
| "dinner"    | 84.80     |
| "lunch"     | 30.58     |
| "breakfast" | 17.50     |
---------------------------
```

Substituting the `AVG()`, `MIN()`, `MAX()`, or `COUNT()` functions for `SUM()` in that query would give you the average, minimum, maximum, or number of values in each group.

The next query demonstrates a nice use of `COUNT()` to explore a new dataset, because it tells us how many times each predicate was used:

```
# filename: ex162.rq

SELECT ?p (COUNT(?p) AS ?pTotal)
WHERE
{ ?s ?p ?o . }
GROUP BY ?p
```

Note that ex162.rq doesn't even declare any namespaces. It's a very general-purpose query.

The expense report data has a very regular format, with the same number of triples describing each meal, so the result of running this query on the expense report data isn't especially interesting, but it does show that the query does its job properly:

```
-----------------------------------------------------------------
| p                                                  | pTotal |
=================================================================
| <http://learningsparql.com/ns/expenses#date>        | 9      |
| <http://learningsparql.com/ns/expenses#description> | 9      |
| <http://learningsparql.com/ns/expenses#amount>      | 9      |
-----------------------------------------------------------------
```

When you explore more unevenly shaped data, this kind of query can tell you a lot about it.

The HAVING keyword does for aggregate values what FILTER does for individual values: it specifies a condition that lets you restrict which values you want to appear in the results. In the following version of the the query that totals up expenses by meal, the HAVING clause tells the SPARQL processor that we're only interested in subtotals greater than 20:

```
# filename: ex164.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT ?description (SUM(?amount) AS ?mealTotal)
WHERE
{
  ?meal e:description ?description ;
        e:amount ?amount .
}
GROUP BY ?description
HAVING (SUM(?amount) > 20)
```

And that's what we get:

```
---------------------------
| description | mealTotal |
===========================
| "dinner"    | 84.80     |
| "lunch"     | 30.58     |
---------------------------
```

Querying a Remote SPARQL Service

We've seen how the FROM keyword can name a dataset to query that may be a local or remote file to query. For example, this next query asks for any Dublin Core title values in Tim Berners-Lee's FOAF file, which is stored on an MIT server:

```
# filename: ex166.rq

PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?title
FROM <http://dig.csail.mit.edu/2008/webdav/timbl/foaf.rdf>
WHERE { ?s dc:title ?title .}
```

The SERVICE keyword gives you another way to query remote data, but instead of pointing at an RDF file somewhere (or at a service delivering the equivalent of an RDF file), you point it at a SPARQL endpoint. While a typical SPARQL query or subquery retrieves data from somewhere local or remote and applies a query to it, the SERVICE keyword lets you say "send this query off to the specified SPARQL endpoint service, which will run the query, and then retrieve the result." It's a great keyword to know, because so many SPARQL endpoint services are available, and their data can add a lot to your applications.

1.1 Alert

The SERVICE keyword is a new feature in SPARQL 1.1.

The SERVICE keyword must go inside of a graph pattern, so you use it to introduce a subquery. Your entire query may just be the one subquery, like in this example:

```
# filename: ex167.rq

PREFIX cat:    <http://dbpedia.org/resource/Category:>
PREFIX skos:   <http://www.w3.org/2004/02/skos/core#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl:    <http://www.w3.org/2002/07/owl#>
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>

SELECT ?p ?o
WHERE
{
  SERVICE <http://DBpedia.org/sparql>
  { SELECT ?p ?o
    WHERE { <http://dbpedia.org/resource/Joseph_Hocking> ?p ?o . }
  }
}
```

ARQ expects you to specify some data to query, either on the command line or with a FROM statement. The query above may appear to specify some data to query, but not really—it's actually naming an endpoint to send the query to. Either way, to run this with ARQ from your command line, you must name a data file with the `--data` argument, even though this query won't do anything with that file's data. This may not be the case with other SPARQL processors.

The subquery above is fairly simple. It just asks for the predicates and objects of all the DBpedia triples that have *http://dbpedia.org/resource/Joseph_Hocking* as their subject. The outer query asks for the same variables, creating this output:

```
-----------------------------------------------------------------|
| p            | o                                               |
=================================================================|
| owl:sameAs   | <http://rdf.freebase.com/ns/guid.9202a8c...>    |
| rdfs:comment | "Joseph Hocking (November 7, 1860-March ..."@en |
| skos:subject | cat:Cornish_writers                             |
| skos:subject | cat:English_Methodist_clergy                    |
| skos:subject | cat:19th-century_Methodist_clergy               |
| skos:subject | cat:People_from_St_Stephen-in-Brannel           |
| skos:subject | cat:1860_births                                 |
| skos:subject | cat:1937_deaths                                 |
| skos:subject | cat:English_novelists                           |
| rdfs:label   | "Joseph Hocking"@en                             |
| foaf:page    | <http://en.wikipedia.org/wiki/Joseph_Hocking>   |
------------------------------------------------------------------
```

This result doesn't have a ton of data, but only because I deliberately picked an obscure person to ask about. I also trimmed the data in the two places where you see `...` above to make it easier to fit on the page; the `rdfs:comment` value describing the British novelist/minister is actually an entire paragraph.

This next query sends a similar request about the same British novelist to the Free University of Berlin's collection of metadata about the Project Gutenberg free eBook collection:

```
# filename: ex170.rq

PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX gp:   <http://www4.wiwiss.fu-berlin.de/gutendata/resource/people/>

SELECT ?p ?o
WHERE
{
  SERVICE <http://www4.wiwiss.fu-berlin.de/gutendata/sparql>
  { SELECT ?p ?o
    WHERE { gp:Hocking_Joseph ?p ?o . }
  }
}
```

Here is the result:

```
 -------------------------------------------------------------------------
| p                                  | o                                  |
=========================================================================
| rdfs:label                         | "Hocking, Joseph"                  |
| <http://xmlns.com/foaf/0.1/name>   | "Hocking, Joseph"                  |
| rdf:type                           | <http://xmlns.com/foaf/0.1/Person> |
-------------------------------------------------------------------------
```

It's not much data, but if you rearrange that query to ask for the subjects and predicates of all the triples where `gp:Hocking_Joseph` is the object, you'll find that one of the subjects is a URI that represents one of his novels. Along with `gp:Hocking_Joseph` as the creator value of this novel, you'll see a link to the novel itself and to other metadata about it. This gets even more interesting when you do the same thing with more well-known authors whose works are now in the public domain—if you're interested in literature, it's a fascinating corner of the Linked Data cloud.

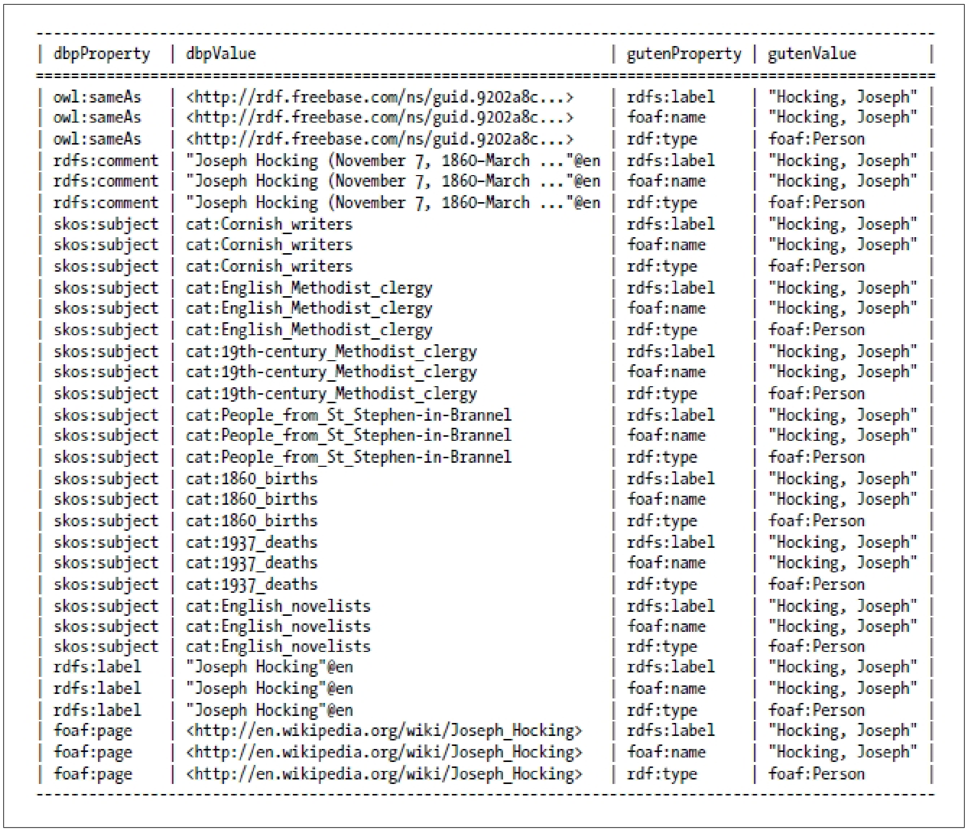
There are many data sources out there offering interesting sets of triples for you to query, but remember that a remote service doesn't have to be very remote, and it doesn't even have to store triples. The open source D2RQ interface lets you use SPARQL to query relational databases, and it's fairly easy to set up, which puts a lot of power in your hands—it means that you can set up your own relational databases to be available for SPARQL queries by the public or by authorized users behind your firewall. The latter option is making semantic technology increasingly popular for giving easier access within an organization to data that would otherwise be hidden within silos, and the SERVICE keyword lets you get at data stored this way.

Federated Queries: Searching Multiple Datasets with One Query

To write and execute a single query that retrieves data from multiple datasets, you already know everything you need to know: just create a subquery for each one. (See "Queries in Your Queries" on page 85 for more on subqueries.) The following combines the two examples that demonstrated the SERVICE keyword in "Querying a Remote SPARQL Service" on page 95, but changes the variable names to keep them unique within each subquery:

```
# filename: ex172.rq

PREFIX cat:  <http://dbpedia.org/resource/Category:>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX gp:   <http://www4.wiwiss.fu-berlin.de/gutendata/resource/people/>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

SELECT ?dbpProperty ?dbpValue ?gutenProperty ?gutenValue
WHERE
{
  SERVICE <http://DBpedia.org/sparql>
  { SELECT ?dbpProperty ?dbpValue
    WHERE
    {
      <http://dbpedia.org/resource/Joseph_Hocking> ?dbpProperty ?dbpValue .
    }
  }

  SERVICE <http://www4.wiwiss.fu-berlin.de/gutendata/sparql>
  { SELECT ?gutenProperty ?gutenValue
    WHERE
    {
      gp:Hocking_Joseph ?gutenProperty ?gutenValue .
    }
  }

}
```

In a federated query like this, once the variables in the first subquery are bound, the SPARQL processor uses the same values for those variables in subsequent subqueries. This can be useful when you want to cross-reference data from multiple sources, but in a case like this it may lead to no results if the results from the first query don't correspond well enough to the results in the second query. That's why this query has different variable names for each query.



```
-------------------------------------------------------------------------------------------------------------
| dbpProperty  | dbpValue                                          | gutenProperty | gutenValue        |
=============================================================================================================
| owl:sameAs   | <http://rdf.freebase.com/ns/guid.9202a8c...>      | rdfs:label    | "Hocking, Joseph" |
| owl:sameAs   | <http://rdf.freebase.com/ns/guid.9202a8c...>      | foaf:name     | "Hocking, Joseph" |
| owl:sameAs   | <http://rdf.freebase.com/ns/guid.9202a8c...>      | rdf:type      | foaf:Person       |
| rdfs:comment | "Joseph Hocking (November 7, 1860-March ..."@en   | rdfs:label    | "Hocking, Joseph" |
| rdfs:comment | "Joseph Hocking (November 7, 1860-March ..."@en   | foaf:name     | "Hocking, Joseph" |
| rdfs:comment | "Joseph Hocking (November 7, 1860-March ..."@en   | rdf:type      | foaf:Person       |
| skos:subject | cat:Cornish_writers                               | rdfs:label    | "Hocking, Joseph" |
| skos:subject | cat:Cornish_writers                               | foaf:name     | "Hocking, Joseph" |
| skos:subject | cat:Cornish_writers                               | rdf:type      | foaf:Person       |
| skos:subject | cat:English_Methodist_clergy                      | rdfs:label    | "Hocking, Joseph" |
| skos:subject | cat:English_Methodist_clergy                      | foaf:name     | "Hocking, Joseph" |
| skos:subject | cat:English_Methodist_clergy                      | rdf:type      | foaf:Person       |
| skos:subject | cat:19th-century_Methodist_clergy                 | rdfs:label    | "Hocking, Joseph" |
| skos:subject | cat:19th-century_Methodist_clergy                 | foaf:name     | "Hocking, Joseph" |
| skos:subject | cat:19th-century_Methodist_clergy                 | rdf:type      | foaf:Person       |
| skos:subject | cat:People_from_St_Stephen-in-Brannel             | rdfs:label    | "Hocking, Joseph" |
| skos:subject | cat:People_from_St_Stephen-in-Brannel             | foaf:name     | "Hocking, Joseph" |
| skos:subject | cat:People_from_St_Stephen-in-Brannel             | rdf:type      | foaf:Person       |
| skos:subject | cat:1860_births                                   | rdfs:label    | "Hocking, Joseph" |
| skos:subject | cat:1860_births                                   | foaf:name     | "Hocking, Joseph" |
| skos:subject | cat:1860_births                                   | rdf:type      | foaf:Person       |
| skos:subject | cat:1937_deaths                                   | rdfs:label    | "Hocking, Joseph" |
| skos:subject | cat:1937_deaths                                   | foaf:name     | "Hocking, Joseph" |
| skos:subject | cat:1937_deaths                                   | rdf:type      | foaf:Person       |
| skos:subject | cat:English_novelists                             | rdfs:label    | "Hocking, Joseph" |
| skos:subject | cat:English_novelists                             | foaf:name     | "Hocking, Joseph" |
| skos:subject | cat:English_novelists                             | rdf:type      | foaf:Person       |
| rdfs:label   | "Joseph Hocking"@en                               | rdfs:label    | "Hocking, Joseph" |
| rdfs:label   | "Joseph Hocking"@en                               | foaf:name     | "Hocking, Joseph" |
| rdfs:label   | "Joseph Hocking"@en                               | rdf:type      | foaf:Person       |
| foaf:page    | <http://en.wikipedia.org/wiki/Joseph_Hocking>     | rdfs:label    | "Hocking, Joseph" |
| foaf:page    | <http://en.wikipedia.org/wiki/Joseph_Hocking>     | foaf:name     | "Hocking, Joseph" |
| foaf:page    | <http://en.wikipedia.org/wiki/Joseph_Hocking>     | rdf:type      | foaf:Person       |
-------------------------------------------------------------------------------------------------------------
```

Figure 3-5. Results of query ex172.rq

The results, shown in Figure 3-5 (again, with data trimmed at the `...` parts), might not be quite what you expected.

When run individually in the preceding section, this query's two subqueries gave us 11 and then 3 rows of results. The combination here gives us 33 rows, so you can guess what's going on: the result is a cross product, or every combination of the two values from each row of the first query with the two values from each row of the second query.

A cross-product of these two sets of triples is not particularly useful, but we'll see in Chapter 4 how multiple sets of data from different sources can be more easily tied together for future reuse.

A federated query can have more than two subqueries, but remember that you're telling the query processor to query one source, then another, and then perhaps more, so it may take awhile.

1.1 Alert

The use of subqueries and the SERVICE keyword means that you need SPARQL 1.1 support to execute federated queries.

Most well-known free public SPARQL endpoints are the result of volunteer labor, so they're not necessarily up and responding to all queries 24 hours a day. Querying more than one free service at once increases your chances that your query may not execute successfully.

Don't write a federated query that goes through an entire remote dataset and then looks for corresponding values for everything it finds in another remote dataset unless you're absolutely sure that the two (or more) datasets are a size that makes such a request reasonable. DBpedia, the Linked Movie Database, the Project Gutenberg metadata, and most other public SPARQL endpoints are too large for this kind of request to be reasonable, and will time out before completely following through on such on a request. Adding the LIMIT and OFFSET keywords to these queries can help you cut back on how much these queries ask of these SPARQL endpoints.

Summary

In this chapter, we learned about:

- The `rdfs:label` property, which can make query results more readable by showing natural language representations of resources instead of URIs.
- The OPTIONAL keyword, which gives your queries the flexibility to retrieve data that may or may not be there without causing a whole graph pattern to fail if there is no match for a particular variable.
- How the same variable in the object position of one triple pattern and the subject position of another can help you to find connected triples, and how property paths can let a short query request a network of data.
- How blank nodes let you group triples together.
- The DISTINCT keyword, which lets you eliminate duplicate data in query results.
- The UNION keyword, which lets you specify multiple graph patterns to retrieve multiple different sets of data and then combine them in the query result.
- The FILTER keyword, which can trim your results down based on boolean conditions that the data must meet; the LIMIT keyword, which lets you cut off the results after the query processor has retrieved a certain number of results; OFFSET, which skips a certain number of results; and, how to retrieve data only from certain named graphs.

- How to use expressions to calculate new values, and how the BIND keyword can simplify your use of expressions.
- How ORDER BY lets you sort data in ascending or descending order, and, when combined with the LIMIT keyword, lets you ask for the biggest and smallest values; how `MAX()`, `AVG()`, `SUM()`, and other aggregate functions let you massage your data even more; and, how to group data by specific values to find aggregate values within the groups.
- How to create subqueries.
- How to assign values to variables.
- The SERVICE keyword, which lets you specify a remote SPARQL endpoint to query.
- How to combine multiple subqueries that each use the SERVICE keyword to perform federated queries.

CHAPTER 4

Copying, Creating, and Converting Data (and Finding Bad Data)

Chapter 3 described many ways to pull triples out of a dataset and to display values from those triples. In this chapter, we'll learn how you can do a lot more than just display those values. We'll learn about:

- "Query Forms: SELECT, DESCRIBE, ASK, and CONSTRUCT" on page 104: Pulling triples out of a dataset with a graph pattern is pretty much the same throughout SPARQL, and you already know several ways to do that. Besides SELECT, there are three more keywords that you can use to indicate what you want to do with those extracted triples.
- "Copying Data" on page 105: Sometimes you just want to pull some triples out of one collection to store in a different one. Maybe you're aggregating data about a particular topic from several sources, or maybe you just want to store data locally so that your applications can work with that data more quickly and reliably.
- "Creating New Data" on page 109: After executing the kind of graph pattern logic that we learned about in the previous chapter, you sometimes have new facts that you can store. Creating new data from existing data is one of the most exciting aspects of SPARQL and the semantic web.
- "Converting Data" on page 114: If your application expects data to fit a certain model, and you have data that almost but not quite fits that model, converting it to triples that fit properly can be easy. If the target model is an established standard, this gives you new opportunities for integrating your data with other data and applications.
- "Finding Bad Data" on page 117: If you can describe the kind of data that you don't want to see, you can find it. When gathering data from multiple sources, this (and the ability to convert data) can be invaluable for massaging data into shape to better serve your applications. Along with the checking of constraints such as the use of appropriate datatypes, these techniques can also let you check a dataset for conformance to business rules.

- "Asking for a Description of a Resource" on page 127: SPARQL's DESCRIBE operation lets you ask for information about the resource represented by a particular URI.

1.1 Alert

The general ideas described in this chapter work with SPARQL 1.0 as well as 1.1, but several examples take advantage of the BIND keyword and functions that are only available in SPARQL 1.1.

Query Forms: SELECT, DESCRIBE, ASK, and CONSTRUCT

As with SQL, SPARQL's most popular verb is SELECT. It lets you request data from a collection whether you want a single phone number or a list of first names, last names, and phone numbers of employees hired after January 1st sorted by last name. SPARQL processors such as ARQ typically show the result of a SELECT query as a table of rows and columns, with a column for each variable name that the query listed after the SELECT keyword, and SPARQL APIs will load the values into a suitable data structure for the programming language that forms the basis of that API.

In SPARQL, SELECT is known as a query form, and there are three more:

- CONSTRUCT returns triples. You can pull triples directly out of a data source without changing them, or you can pull values out and use those values to create new triples. This lets you copy, create, and convert RDF data, and it makes it easier to identify data that doesn't conform to specific business rules.
- ASK asks a query processor whether a given graph pattern describes a set of triples in a particular dataset or not, and the processor returns a boolean true or false. This is great for expressing business rules about conditions that should or should not hold true in your data. You can use sets of these rules to automate quality control in your data processing pipeline.
- DESCRIBE asks for triples that describe a particular resource. The SPARQL specification leaves it up to the query processor to decide which triples to send back as a description of the named resource. This has led to inconsistent implementations of DESCRIBE queries, so this query form isn't very popular, but it's worth playing with.

Most of this chapter covers the broad range of uses that people find for the CONSTRUCT query form. We'll also see some examples of how to put ASK to use, and we'll try out DESCRIBE.

Copying Data

The CONSTRUCT keyword lets you create triples, and those triples can be exact copies of the triples from your input. As a review, imagine that we want to query the following dataset from Chapter 1 for all the information about Craig Ellis.

```
# filename: ex012.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName "Richard" .
d:i0432 ab:lastName  "Mutt" .
d:i0432 ab:homeTel   "(229) 276-5135" .
d:i0432 ab:email     "richard49@hotmail.com" .

d:i9771 ab:firstName "Cindy" .
d:i9771 ab:lastName  "Marshall" .
d:i9771 ab:homeTel   "(245) 646-5488" .
d:i9771 ab:email     "cindym@gmail.com" .

d:i8301 ab:firstName "Craig" .
d:i8301 ab:lastName  "Ellis" .
d:i8301 ab:email     "craigellis@yahoo.com" .
d:i8301 ab:email     "c.ellis@usairwaysgroup.com" .
```

The SELECT query would be simple. We want the subject, predicate, and object of all triples where that same subject has an `ab:firstName` value of “Craig” and an `ab:lastName` value of Ellis:

```
# filename: ex174.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d:  <http://learningsparql.com/ns/data#>

SELECT ?person ?p ?o
WHERE
{
  ?person ab:firstName "Craig" ;
          ab:lastName  "Ellis" ;
          ?p ?o .
}
```

The subjects, predicates, and objects get stored in the `?person`, `?p`, and `?o` variables, and ARQ returns these values with a column for each variable:

```
---------------------------------------------------------
| person  | p            | o                            |
=========================================================
| d:i8301 | ab:email     | "c.ellis@usairwaysgroup.com" |
| d:i8301 | ab:email     | "craigellis@yahoo.com"       |
| d:i8301 | ab:lastName  | "Ellis"                      |
| d:i8301 | ab:firstName | "Craig"                      |
---------------------------------------------------------
```

A CONSTRUCT version of the same query has the same graph pattern following the WHERE keyword, but specifies a triple to create with each set of values that got bound to the three variables:

```
# filename: ex176.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d:  <http://learningsparql.com/ns/data#>

CONSTRUCT
{ ?person ?p ?o . }

WHERE
{
  ?person ab:firstName "Craig" ;
          ab:lastName  "Ellis" ;
          ?p ?o .
}
```

The set of triple patterns (just one in ex176.rq) that describe what to create is itself a graph pattern, so don't forget to enclose it in curly braces.

A SPARQL query processor returns the data for a CONSTRUCT query as actual triples, not as a formatted report with a column for each named variable. The format of these triples depends on the processor you use. ARQ returns them as a Turtle text file, which should look familiar; here is what ARQ returns after running query ex176.rq on the data in ex012.ttl:

```
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix ab:      <http://learningsparql.com/ns/addressbook#> .

d:i8301
      ab:email      "c.ellis@usairwaysgroup.com" ;
      ab:email      "craigellis@yahoo.com" ;
      ab:firstName  "Craig" ;
      ab:lastName   "Ellis" .
```

This may not seem especially exciting, but when you use this technique to gather data from one or more remote sources, it gets more interesting. The following shows a variation on the ex172.rq query from the last chapter, this time pulling triples about Joseph Hocking from the two SPARQL endpoints:

```
# filename: ex178.rq

PREFIX cat:  <http://dbpedia.org/resource/Category:>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX gp:   <http://www4.wiwiss.fu-berlin.de/gutendata/resource/people/>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
```

```
CONSTRUCT
{
  <http://dbpedia.org/resource/Joseph_Hocking> ?dbpProperty ?dbpValue .
  gp:Hocking_Joseph ?gutenProperty ?gutenValue .
}


WHERE
{
  SERVICE <http://DBpedia.org/sparql>
  { SELECT ?dbpProperty ?dbpValue
    WHERE
    {
      <http://dbpedia.org/resource/Joseph_Hocking> ?dbpProperty ?dbpValue .
    }
  }

  SERVICE <http://www4.wiwiss.fu-berlin.de/gutendata/sparql>
  { SELECT ?gutenProperty ?gutenValue
    WHERE
    {
      gp:Hocking_Joseph ?gutenProperty ?gutenValue .
    }
  }

}
```

The CONSTRUCT graph pattern in this query has two triple patterns. It can have as many as you like.

The result (with the paragraph of description about Hocking trimmed at "...") has the 14 triples about him pulled from the two sources:

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix cat:     <http://dbpedia.org/resource/Category:> .
@prefix foaf:    <http://xmlns.com/foaf/0.1/> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix skos:    <http://www.w3.org/2004/02/skos/core#> .
@prefix gp:      <http://www4.wiwiss.fu-berlin.de/gutendata/resource/people/> .

<http://dbpedia.org/resource/Joseph_Hocking>
  rdfs:comment  "Joseph Hocking (November 7, 1860-March 4, 1937) was ..."@en ;
  rdfs:label    "Joseph Hocking"@en ;
  owl:sameAs    <http://rdf.freebase.com/ns/guid.9202a8c04000641f800000000ab14b75> ;
  skos:subject  <http://dbpedia.org/resource/Category:People_from_St_Stephen-in-Brannel> ;
  skos:subject  <http://dbpedia.org/resource/Category:1860_births> ;
  skos:subject  <http://dbpedia.org/resource/Category:English_novelists> ;
  skos:subject  <http://dbpedia.org/resource/Category:Cornish_writers> ;
  skos:subject  <http://dbpedia.org/resource/Category:19th-century_Methodist_clergy> ;
  skos:subject  <http://dbpedia.org/resource/Category:1937_deaths> ;
  skos:subject  <http://dbpedia.org/resource/Category:English_Methodist_clergy> ;
  foaf:page     <http://en.wikipedia.org/wiki/Joseph_Hocking> .

gp:Hocking_Joseph
  rdf:type      foaf:Person ;
  rdfs:label    "Hocking, Joseph" ;
  foaf:name     "Hocking, Joseph" .
```

You also can use the GRAPH keyword to ask for all the triples from a particular named graph:

```
# filename: ex180.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

CONSTRUCT
{ ?course ab:courseTitle ?courseName . }
FROM NAMED <ex125.ttl>
FROM NAMED <ex122.ttl>
WHERE
{
  GRAPH <ex125.ttl> { ?course ab:courseTitle ?courseName }
}
```

The result of this query is essentially a copy of the data in the ex125.ttl graph, because all it had were triples with predicates of `ab:courseTitle`:

```
@prefix ab:      <http://learningsparql.com/ns/addressbook#> .

ab:course24
      ab:courseTitle  "Using Named Graphs" .

ab:course42
      ab:courseTitle  "Combining Public and Private RDF Data" .
```

It's a pretty artificial example, because there's not much point in naming two graphs and then asking for all the triples from one of them—especially with the ARQ command line utility, where a named graph corresponds to an existing disk file, because then I'm creating a copy of something I already have. However, when you work with triplestores that hold far more triples than you would ever store in a file on your hard disk, you'll better appreciate the ability to grab all the triples from a specific named graph.

In Chapter 3, we saw that using the FROM keyword without following it with the NAMED keyword lets you name the dataset to query right in your query. This works for CONSTRUCT queries as well. The following query retrieves and outputs all the triples (as of this writing, about 22 of them) from the Freebase community database about Joseph Hocking:

```
# filename: ex182.rq

CONSTRUCT
{ ?s ?p ?o }
FROM <http://rdf.freebase.com/rdf/en.joseph_hocking>
```

```
WHERE
{ ?s ?p ?o }
```

The important overall lesson so far is that in a CONSTRUCT query the graph pattern after the WHERE keyword can use all the techniques you learned about in the chapters before this one, but that after the CONSTRUCT keyword, instead of a list of variable names, you put a graph pattern showing the triples you want CONSTRUCTed. In the simplest case, these triples are straight copies of the ones extracted from the source dataset or datasets.

Creating New Data

As the ex178.rq query above showed, the triples you create in a CONSTRUCT query need not be composed entirely of variables. If you want, you can create one or more triples entirely from hard-coded values, with an empty GRAPH pattern following the WHERE keyword:

```
# filename: ex184.rq

PREFIX dc: <http://purl.org/dc/elements/1.1/>
CONSTRUCT
{
  <http://learningsparql.com/ns/data/book312> dc:title "Jabez Easterbrook" .
}
WHERE
{}
```

When you rearrange and combine the values retrieved from the dataset, though, you see more of the real power of CONSTRUCT queries. For example, while copying the data for everyone in ex012.ttl who has a phone number, if you can be sure that the second through fourth characters of the phone number are the area code, you can create and populate a new `areaCode` property with a query like this:

```
# filename: ex185.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

CONSTRUCT
{
  ?person ?p ?o ;
          ab:areaCode ?areaCode .
}
WHERE
{
  ?person ab:homeTel ?phone ;
          ?p ?o .
  BIND (SUBSTR(?phone,2,3) as ?areaCode)
}
```

The ?person ?p ?o triple pattern after the WHERE keyword would have copied all the triples, including the ab:homeTel value, even if the ?person ab:homeTel ?phone triple pattern wasn't there. The query included the ab:homeTel triple pattern to allow the storing of the phone number value in the ?phone variable so that the BIND statement could use it to calculate the area code.

The result of running this query with the data in ex012.ttl shows all the triples associated with the two people from the dataset who have phone numbers, and now they each have a new triple showing their area code:

```
@prefix d:      <http://learningsparql.com/ns/data#> .
@prefix ab:     <http://learningsparql.com/ns/addressbook#> .

d:i9771
      ab:areaCode   "245" ;
      ab:email      "cindym@gmail.com" ;
      ab:firstName  "Cindy" ;
      ab:homeTel    "(245) 646-5488" ;
      ab:lastName   "Marshall" .

d:i0432
      ab:areaCode   "229" ;
      ab:email      "richard49@hotmail.com" ;
      ab:firstName  "Richard" ;
      ab:homeTel    "(229) 276-5135" ;
      ab:lastName   "Mutt" .
```

We'll learn more about functions like SUBSTR() in Chapter 5. As you develop CONSTRUCT queries, remember that the more functions you know how to use in your queries, the more kinds of data you can create.

We used the SUBSTR() function above to calculate the area code values, but you don't need to use function calls to infer new data from existing data. It's very common in SPARQL queries to look for relationships among the data and to then create new triples that make those relationships explicit. For a few examples of this, we'll use this data about the gender and parental relationships of several people:

```
# filename: ex187.ttl

@prefix d:  <http://learningsparql.com/ns/data#> .
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

d:jane ab:hasParent d:gene .
d:gene ab:hasParent d:pat ;
       ab:gender    d:female .
d:joan ab:hasParent d:pat ;
       ab:gender    d:female .
```

```
d:pat  ab:gender    d:male .
d:mike ab:hasParent d:joan .
```

Our first query with this data looks for people who have a parent who themselves have a male parent. It then outputs a fact about the parent of the parent being the grandfather of the person. Or, in SPARQL terms, it looks for a person `?p` with an `ab:hasParent` relationship to someone whose identifier will be stored in the variable `?parent`, and then it looks for someone who that `?parent` has an `ab:hasParent` relationship with who has an `ab:gender` value of `d:male`. If it finds such a person, it outputs a triple saying that the person `?p` has the relationship `ab:Grandfather` to `?g`.

```
# filename: ex188.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d:  <http://learningsparql.com/ns/data#>

CONSTRUCT
{ ?p ab:hasGrandfather ?g . }
WHERE
{
  ?p ab:hasParent ?parent .
  ?parent ab:hasParent ?g .
  ?g ab:gender d:male .
}
```

The query finds that two people have an `ab:grandParent` relationship to someone else in the ex187.ttl dataset:

```
@prefix d:      <http://learningsparql.com/ns/data#> .
@prefix ab:     <http://learningsparql.com/ns/addressbook#> .

d:mike
      ab:hasGrandfather  d:pat .

d:jane
      ab:hasGrandfather  d:pat .
```

A different query with the same data creates triples about who is the aunt of who:

```
# filename: ex190.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d:  <http://learningsparql.com/ns/data#>

CONSTRUCT
{ ?p ab:hasAunt ?aunt . }
WHERE
{
  ?p ab:hasParent ?parent .
     ?parent ab:hasParent ?g .
     ?aunt ab:hasParent ?g ;
           ab:gender d:female .

  FILTER (?parent != ?aunt)
}
```

The query can't just ask about someone's parents' sisters, because there is no explicit data about sisters in the dataset, so:

1. It looks for a grandparent of `?p`, as before.
2. It also looks for someone different from the parent of `?p` (with the difference ensured by the FILTER statement) who has that same grandparent (stored in `?g`) as a parent.
3. If that person has an `ab:gender` value of `d:female`, the query outputs a triple about that person being the aunt of `?p`.

```
@prefix d:      <http://learningsparql.com/ns/data#> .
@prefix ab:     <http://learningsparql.com/ns/addressbook#> .

d:mike
      ab:hasAunt    d:gene .

d:jane
      ab:hasAunt    d:joan .
```

Are these queries really creating new information? A relational database developer would be quick to point out that they're not—that they're actually taking information that is implicit and making it explicit. In relational database design, much of the process known as normalization involves looking for redundancies in the data, including the storage of data that could instead be calculated dynamically as necessary—for example, the grandfather and aunt relationships output by the last two queries.

A relational database, though, is a closed world with very fixed boundaries. The data that's there is the data that's there, and combining two relational databases so that you can search for new relationships between table rows from the different databases is much easier said than done. In semantic web and Linked Data applications, the combination of two datasets like this is very common; easy data aggregation is one of RDF's greatest benefits. Combining data, finding patterns, and then storing data about what was found is popular in many of the fields that use this technology, such as pharmaceutical and intelligence research.

In "Reusing and Creating Vocabularies: RDF Schema and OWL" on page 35, we saw how declaring a resource to be a member of a particular class can tell people more about it, because there may be metadata associated with that class. Let's see how a small revision to that last query can make it even more explicit that a resource matching the `?aunt` variable is an aunt. We'll add a triple saying that she's a member of that specific class:

```
# filename: ex192.rq

PREFIX ab:  <http://learningsparql.com/ns/addressbook#>
PREFIX d:   <http://learningsparql.com/ns/data#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT
{
  ?aunt rdf:type ab:Aunt .
  ?p ab:hasAunt ?aunt .
}

WHERE
{
  ?p ab:hasParent ?parent .
  ?parent ab:hasParent ?g .
  ?aunt ab:hasParent ?g ;
        ab:gender d:female .

FILTER (?parent != ?aunt)
}
```

Identifying resources as members of classes is a very good practice because it makes it much easier to infer information about your data.

Making a resource a member of a class that hasn't been declared is not an error, but there's not much point to it. The triples created by the query above should be used with additional triples from an ontology that declares that an aunt is a class and adds at least a bit of metadata about it, like this:

```
# filename: ex193.ttl

@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ab:   <http://learningsparql.com/ns/addressbook#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .

ab:Aunt rdf:type owl:Class ;
        rdfs:comment "The sister of one of the resource's parents." .
```

Classes are also members of a class—the class `rdfs:Class`, or its subclass `rdfs:Class`. Note the similarity of the triple saying "`ab:Aunt` is a member of the class `owl:Class`" to the triple saying "`?aunt` is a member of class `ab:Aunt`."

There's nothing to prevent you from putting the two ex193.ttl triples in the graph pattern after the ex192.rq query's CONSTRUCT keyword, as long as you remember to include the declarations for the `rdf:`, `rdfs:`, and `owl:` prefixes. The query would then create those triples when it creates the triple saying that `?aunt` is a member of the class `ab:Aunt`. In practice, though, when you say that a resource is a member of a particular class, you're probably doing it because that class is already declared somewhere else.

Converting Data

Because CONSTRUCT queries can create new triples based on information extracted from a dataset, they're a a great way to convert data that uses properties from one namespace into data that uses properties from another. This lets you take data from just about anywhere and turn it into something that you can use in your system.

Typically, this means converting data that uses one schema or ontology into data that uses another, but sometimes (especially with the input data) there is no specific schema in use, and you're just replacing one set of predicates with another. Ideally, though, an ontology exists for the target format, which is often why you're doing the conversion —so that your new version of the data conforms to a known schema and is therefore easier to combine with other data.

Let's look at an example. We've been using the ex012.ttl data file, shown here, since Chapter 1:

```
# filename: ex012.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName "Richard" .
d:i0432 ab:lastName  "Mutt" .
d:i0432 ab:homeTel   "(229) 276-5135" .
d:i0432 ab:email     "richard49@hotmail.com" .

d:i9771 ab:firstName "Cindy" .
d:i9771 ab:lastName  "Marshall" .
d:i9771 ab:homeTel   "(245) 646-5488" .
d:i9771 ab:email     "cindym@gmail.com" .

d:i8301 ab:firstName "Craig" .
d:i8301 ab:lastName  "Ellis" .
d:i8301 ab:email     "craigellis@yahoo.com" .
d:i8301 ab:email     "c.ellis@usairwaysgroup.com" .
```

A serious address book application would be better off storing this data using the FOAF ontology or the W3C ontology that models vCard, a standard file format for modeling business card information. The following query converts the data above to vCard RDF:

```
# filename: ex194.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX v:  <http://www.w3.org/2006/vcard/ns#>

CONSTRUCT
{
 ?s v:given-name  ?firstName ;
    v:family-name ?lastName ;
    v:email       ?email ;
    v:homeTel     ?homeTel .
}
WHERE
{
 ?s ab:firstName ?firstName ;
    ab:lastName  ?lastName ;
    ab:email     ?email .
    OPTIONAL
    { ?s ab:homeTel ?homeTel . }
}
```

We first learned about the OPTIONAL keyword in "Data That Might Not Be There" on page 53 of Chapter 3. It serves the same purpose here that it serves in a SELECT query: to indicate that an unmatched part of the graph pattern should not prevent the matching of the rest of the pattern. In this case, if an input resource has no `ab:homeTel` value but does have `ab:firstName`, `ab:lastName`, and `ab:email` values, we still want those last three.

ARQ outputs this when applying the ex194.rq query to the ex012.ttl data:

```
@prefix v:       <http://www.w3.org/2006/vcard/ns#> .
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix ab:      <http://learningsparql.com/ns/addressbook#> .

d:i9771
      v:email       "cindym@gmail.com" ;
      v:family-name  "Marshall" ;
      v:given-name  "Cindy" ;
      v:homeTel     "(245) 646-5488" .

d:i0432
      v:email       "richard49@hotmail.com" ;
      v:family-name  "Mutt" ;
      v:given-name  "Richard" ;
      v:homeTel     "(229) 276-5135" .

d:i8301
      v:email       "c.ellis@usairwaysgroup.com" ;
      v:email       "craigellis@yahoo.com" ;
      v:family-name  "Ellis" ;
      v:given-name  "Craig" .
```

Converting `ab:email` to `v:email` or `ab:homeTel` to `v:homeTel` may not seem like much of a change, but remember the URIs that those prefixes stand for. Lots of software in the semantic web world will recognize the predicate *http://www.w3.org/2006/vcard/ns#email*, but nothing outside of what I've written for this book will recognize *http://learningsparql.com/ns/addressbook#email*, so there's a big difference.

Converting data may also mean normalizing resource URIs to more easily combine data. For example, let's say I have a set of data about British novelists, and I'm using the URI *http://learningsparql.com/ns/data#HockingJoseph* to represent Joseph Hocking. The following variation on the ex178.rq CONSTRUCT query, which pulled triples about this novelist both from DBpedia and from the Project Gutenberg metadata, doesn't copy the triples exactly; instead, it uses my URI for him as the subject of all the constructed triples:

```
# filename: ex196.rq

PREFIX cat:  <http://dbpedia.org/resource/Category:>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX gp:   <http://www4.wiwiss.fu-berlin.de/gutendata/resource/people/>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX d:    <http://learningsparql.com/ns/data#>
CONSTRUCT
{
  d:HockingJoseph ?dbpProperty ?dbpValue ;
                  ?gutenProperty ?gutenValue .

}
WHERE
{
  SERVICE <http://DBpedia.org/sparql>
  { SELECT ?dbpProperty ?dbpValue
    WHERE
    {
      <http://dbpedia.org/resource/Joseph_Hocking> ?dbpProperty ?dbpValue .
    }
  }

  SERVICE <http://www4.wiwiss.fu-berlin.de/gutendata/sparql>
  { SELECT ?gutenProperty ?gutenValue
    WHERE
    {
      gp:Hocking_Joseph ?gutenProperty ?gutenValue .
    }
  }

}
```

Like the triple patterns in a WHERE graph pattern and in Turtle data, the triples in a CONSTRUCT graph pattern can use semicolons and commas to be more concise.

The result of running the query has 14 triples about *http://learningsparql.com/ns/data#HockingJoseph*:

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix cat:     <http://dbpedia.org/resource/Category:> .
@prefix d:       <http://learningsparql.com/ns/data#> .
```

```
@prefix foaf:    <http://xmlns.com/foaf/0.1/> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix skos:    <http://www.w3.org/2004/02/skos/core#> .
@prefix gp:      <http://www4.wiwiss.fu-berlin.de/gutendata/resource/people/> .

d:HockingJoseph
  rdf:type       foaf:Person ;
  rdfs:comment   "Joseph Hocking (November 7, 1860-March 4, 1937) was..."@en ;
  rdfs:label     "Hocking, Joseph" ;
  rdfs:label     "Joseph Hocking"@en ;
  owl:sameAs     <http://rdf.freebase.com/ns/guid.9202a...> ;
  skos:subject   <http://dbpedia.org/resource/Category:People_from_St_Stephen-in-Brannel> ;
  skos:subject   <http://dbpedia.org/resource/Category:1860_births> ;
  skos:subject   <http://dbpedia.org/resource/Category:English_novelists> ;
  skos:subject   <http://dbpedia.org/resource/Category:Cornish_writers> ;
  skos:subject   <http://dbpedia.org/resource/Category:19th-century_Methodist_clergy> ;
  skos:subject   <http://dbpedia.org/resource/Category:1937_deaths> ;
  skos:subject   <http://dbpedia.org/resource/Category:English_Methodist_clergy> ;
  foaf:name      "Hocking, Joseph" ;
  foaf:page      <http://en.wikipedia.org/wiki/Joseph_Hocking> .
```

If different URIs are used to represent the same resource in different datasets (such as *http://dbpedia.org/resource/Joseph_Hocking* and *http://www4.wiwiss.fu-berlin.de/gutendata/resource/people/Hocking_Joseph* in the data retrieved by ex196.rq) and you want to aggregate the data and record the fact that they're referring to the same thing, there are better ways to do it than changing all of their URIs. The `owl:sameAs` predicate you see in one of the triples that this query retrieved from DBpedia is one approach. (Also, when collecting triples from multiple sources, you might want to record when and where you got them, which is where named graphs become useful—you can assign this information as metadata about a graph.) In this particular case, the changing of the URI is just another example of how you can use CONSTRUCT to massage some data.

Finding Bad Data

In relational database development, XML, and other areas of information technology, a schema is a set of rules about structure and datatypes to ensure data quality and more efficient systems. If one of these schemas says that `quantity` values must be integers, you know that one can never be 3.5 or "hello". This way, developers writing applications to process the data need not worry about strange data that will break the processing—if a program subtracts 1 from the `quantity` amount and a quantity might be "hello", this could lead to trouble. If the data conforms to a proper schema, the developer using the data doesn't have to write code to account for that possibility.

Semantic web applications take a different approach. Instead of providing a template that data must fit into so that processing applications can make assumptions about the data, RDF Schema and OWL ontologies add additional metadata. For example, when we know that resource `d:id432` is a member of the class `d:product3973`, which has an `rdfs:label` of "strawberries" and is a subclass of the class with an `rdfs:label` of "fruit", then we know that `d:product3973` is a member of the class "fruit" as well.

This is great, but what if you do want to define rules for your triples and check whether a set of data conforms to them so that an application doesn't have to worry about unexpected data values breaking its logic? OWL provides some ways to do this, but these can get quite complex, and you'll need an OWL-aware processor. The use of SPARQL to define such constraints is becoming more popular, both for its simplicity and the broader range of software (that is, all SPARQL processors) that let you implement these rules.

As a bonus, the same techniques let you define business rules, which are completely beyond the scope of SQL in relational database development. They're also beyond the scope of traditional XML schemas, although the Schematron language has made contributions there.

Defining Rules with SPARQL

For some sample data with errors that we'll track down, the following variation on last chapter's ex104.ttl data file adds a few things. Let's say I have an application that uses a large amount of similar data, but I want to make sure that the data conforms to a few rules before I feed it to that application.

```
# filename: ex198.ttl

@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:item432 dm:cost 8.50 ;
          dm:amount 14 ;
          dm:approval d:emp079 ;
          dm:location <http://dbpedia.org/resource/Boston> .

d:item201 dm:cost 9.25 ;
          dm:amount 12 ;
          dm:approval d:emp092 ;
          dm:location <http://dbpedia.org/resource/Ghent> .

d:item857 dm:cost 12 ;
          dm:amount 10 ;
          dm:location <http://dbpedia.org/resource/Montreal> .

d:item693 dm:cost 10.25 ;
          dm:amount 1.5 ;
          dm:location "Heidelberg" .

d:item126 dm:cost 5.05 ;
          dm:amount 4 ;
          dm:location <http://dbpedia.org/resource/Lisbon> .
```

```
d:emp092  dm:jobGrade 1 .
d:emp041  dm:jobGrade 3 .
d:emp079  dm:jobGrade 5 .
```

Here are the rules, and here is how this dataset breaks them:

- All the `dm:location` values must be URIs, because I want to connect this data with other related data. Item `d:item693` has a `dm:location` value of "Heidelberg", which is a string, not a URI.
- All the `dm:amount` values must be integers. Above, `d:item693` has an `dm:amount` value of 1.5, which I don't want to send to my application.
- As more of a business rule than a data checking rule, I consider a `dm:approval` value to be optional if the total cost of a purchase is less than or equal to 100. If it's greater than 100, the purchase must be approved by an employee with a job grade greater than 4. The purchase of 14 `d:item432` items at 8.50 each costs more than 100, but it's approved by someone with a job grade of 5, so it's OK. `d:item126` has no approval listed, but at a total cost of 20.20, it needs no approval. However, `d:item201` costs over 100 and the approving employee has a job grade of 1, and `d:item857` also costs over 100 and has no approval at all, so I want to catch those.

Because the ASK query form asks whether a given graph pattern can be matched in a given dataset, by defining a graph pattern for something that breaks a rule, we can create a query that asks "Does this data contain violations of this rule?" In "FILTERing Data Based on Conditions" on page 73 of the last chapter, we saw that the ex107.rq query listed all the `dm:location` values that were not valid URIs. A slight change turns it into an ASK query that checks whether this problem exists in the input dataset:

```
# filename: ex199.rq

PREFIX dm: <http://learningsparql.com/ns/demo#>

ASK WHERE
{
  ?s dm:location ?city .
  FILTER (!(isURI(?city)))
}
```

ARQ responds with the following:

```
Ask => Yes
```

Other SPARQL engines might return an `xsd:boolean` true value. If you're using an interface to a SPARQL processor that is built around a particular programming language, it would probably return that language's representation of a boolean true value.

Using the `datatype()` function that we'll learn more about in Chapter 5, a similar query asks whether there are any resources in the input dataset with a `dm:amount` value that does not have a type of `xsd:integer`:

```
# filename: ex201.rq

PREFIX dm:  <http://learningsparql.com/ns/demo#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

ASK WHERE
{
  ?item dm:amount ?amount .
  FILTER ((datatype(?amount)) != xsd:integer)
}
```

The `d:item693` resource's 1.5 value for `dm:amount` matches this pattern, so ARQ responds to this query with `Ask => Yes`.

A slightly more complex query is needed to check for conformance to the business rule about necessary purchase approvals, but it combines techniques you already know about: it uses an OPTIONAL graph pattern, because purchase approval is not required in all conditions, and it uses the BIND keyword to calculate a `?totalCost` for each purchase that can be compared with the boundary value of 100. It also uses parentheses and the boolean `&&` and `||` operators to indicate that a resource violating this constraint must have a `?totalCost` value over 100 and either no value bound to `?grade` (which would happen if no employee who had been assigned a job grade had approved the purchase) or if the `?grade` value was less than 5. Still, it's not a very long query!

```
# filename: ex202.rq

PREFIX dm:  <http://learningsparql.com/ns/demo#>

ASK WHERE
{
  ?item dm:cost ?cost ;
        dm:amount ?amount .
  OPTIONAL
  {
    ?item dm:approval ?approvingEmployee .
    ?approvingEmployee dm:jobGrade ?grade .
  }

  BIND (?cost * ?amount AS ?totalCost) .
  FILTER ((?totalCost > 100) &&
          ( (!(bound(?grade)) || (?grade < 5 ) )))
}
```

ARQ also responds to this query with `Ask => Yes`.

If you were checking a dataset against 40 SPARQL rules like this, you wouldn't want to repeat the process of reading the file from disk, having ARQ run a query on it, and checking the result 40 times. When you use a SPARQL processor API such as the Jena API behind ARQ, or when you use a development framework product, you'll find other options for efficiently checking a dataset against a large batch of rules expressed as queries.

Generating Data About Broken Rules

Sometimes it's handy to set up something that tells you whether a dataset conforms to a set of SPARQL rules or not. More often, though, if a resource's data breaks any rules, you'll want to know which resources broke which rules.

If a semantic web application checked for data that broke certain rules and then let you know which problems it found and where, how would it represent this information? With triples, of course. The following revision of ex199.rq is identical to the original except that it includes a new namespace declaration and replaces the ASK keyword with a CONSTRUCT clause. The CONSTRUCT clause has a graph pattern of two triples to create when the query finds a problem:

```
# filename: ex203.rq

PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

CONSTRUCT
{
  ?s dm:problem dm:prob29 .
  dm:prob29 rdfs:label "Location value must be a URI." .
}

WHERE
{
  ?s dm:location ?city .
  FILTER (!(isURI(?city)))
}
```

When you describe something (in this case, a problem found in the input data) with RDF, you need to have an identifier for the thing you're describing, so I assigned the identifier `dm:prob29` to the problem of a `dm:location` value not being a URI. You can name these problems anything you like, but instead of trying to include a description of the problem right in the URI, I used the classic RDF approach: I assigned a short description of the problem to it with an `rdfs:label` value in the second triple being created by the CONSTRUCT statement above. (See "More Readable Query Results" on page 46 for more on this.)

Running this query, we're not just asking whether there's a bad `dm:location` value somewhere in the ex198.ttl dataset. We're asking which resources have a problem and what that problems is, and running the ex203.rq query gives us this information:

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix dm:      <http://learningsparql.com/ns/demo#> .

dm:prob29
      rdfs:label    "Location value must be a URI." .

d:item693
      dm:problem    dm:prob29 .
```

As we'll see below, a properly modeled vocabulary for problem identification declares a class for the problems and various related properties. Each time a CONSTRUCT query that searches for these problems finds one, it declares a new instance of the problem class and sets the relevant properties. Cooperating applications can use the model to find out what to look for when using the data.

The following revision of ex201.rq is similar to the ex203.rq revision of ex199.rq: it replaces the ASK keyword with a CONSTRUCT clause that has a graph pattern of two triples to create whenever a problem of this type is found:

```
# filename: ex205.rq

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dm:  <http://learningsparql.com/ns/demo#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT
{
  ?item dm:problem dm:prob32 .
  dm:prob32 rdfs:label "Amount must be an integer." .
}

WHERE
{
  ?item dm:amount ?amount .
  FILTER ((datatype(?amount)) != xsd:integer)
}
```

Running this query shows the resource with this problem and a description of the problem:

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix dm:      <http://learningsparql.com/ns/demo#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .

dm:prob32
      rdfs:label    "Amount must be an integer." .

d:item693
      dm:problem    dm:prob32 .
```

Finally, here's our last ASK constraint-checking query, revised to tell us which resources broke the rule about approval of expenditures over 100:

```
# filename: ex207.rq

PREFIX dm:  <http://learningsparql.com/ns/demo#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

CONSTRUCT
{
  ?item dm:problem dm:prob44 .
  dm:prob44 rdfs:label "Expenditures over 100 require grade 5 approval." .
}

WHERE
{
  ?item dm:cost ?cost ;
        dm:amount ?amount .
  OPTIONAL
  {
    ?item dm:approval ?approvingEmployee .
    ?approvingEmployee dm:jobGrade ?grade .
  }

  BIND (?cost * ?amount AS ?totalCost) .
  FILTER ((?totalCost > 100) &&
          ( (!(bound(?grade)) || (?grade < 5 ) )))
}
```

Here is the result:

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix dm:      <http://learningsparql.com/ns/demo#> .

dm:prob44
      rdfs:label    "Expenditures over 100 require grade 5 approval." .

d:item857
      dm:problem    dm:prob44 .

d:item201
      dm:problem    dm:prob44 .
```

To check all three problems at once, I combined the last three queries into the following single one using the UNION keyword. I used different variable names to store the URIs of the potentially problematic resources to make the connection between the constructed queries and the matched patterns clearer. I also added a label about a `dm:probXX` problem just to show that all the triples about problem labels will appear in the output whether the problems were found or not, because they're hardcoded triples with no dependencies on any matched patterns. The constructed triples about the problems, however, only appear when the problems are found (that is, when triples are found that meet the rule-breaking conditions so that the appropriate variables get bound):

```
# filename: ex209.rq

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dm:  <http://learningsparql.com/ns/demo#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT
{
  ?prob32item dm:problem dm:prob32 .
  dm:prob32 rdfs:label "Amount must be an integer." .
  ?prob29item dm:problem dm:prob29 .
  dm:prob29 rdfs:label "Location value must be a URI." .

  ?prob44item dm:problem dm:prob44 .
  dm:prob44 rdfs:label "Expenditures over 100 require grade 5 approval." .

  dm:probXX rdfs:label "This is a dummy problem." .
}

WHERE
{
  {
    ?prob32item dm:amount ?amount .
    FILTER ((datatype(?amount)) != xsd:integer)
  }

  UNION

  {
    ?prob29item dm:location ?city .
    FILTER (!(isURI(?city)))
  }

  UNION

  {
    ?prob44item dm:cost ?cost ;
                dm:amount ?amount .
    OPTIONAL
    {
      ?item dm:approval ?approvingEmployee .
      ?approvingEmployee dm:jobGrade ?grade .
    }

    BIND (?cost * ?amount AS ?totalCost) .
    FILTER ((?totalCost > 100) &&
            ( (!(bound(?grade)) || (?grade < 5 ) )))
  }

}
```

Here is our result:

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix dm:      <http://learningsparql.com/ns/demo#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .

dm:prob44
      rdfs:label    "Expenditures over 100 require grade 5 approval." .

d:item432
      dm:problem    dm:prob44 .

dm:probXX
```

```
        rdfs:label    "This is a dummy problem." .

dm:prob29
        rdfs:label    "Location value must be a URI." .

dm:prob32
        rdfs:label    "Amount must be an integer." .

d:item857
        dm:problem    dm:prob44 .

d:item201
        dm:problem    dm:prob44 .

d:item693
        dm:problem    dm:prob29 ;
        dm:problem    dm:prob32 .
```

Combining multiple SPARQL rules into one query won't scale very well, because there'd be greater and greater room for error in keeping the rules' variables out of each other's way. A proper rule-checking framework provides a way to store the rules separately and then pipeline them, perhaps in different combinations for different datasets.

Using Existing SPARQL Rules Vocabularies

To keep things simple in this book's explanations, I made up minimal versions of the vocabularies I needed as I went along. For a serious application, I'd look for existing vocabularies to use, just as I use vCard properties in my real address book. For generating triple-based error messages about constraint violations in a set of data, there are two vocabularies that I could use: Schemarama and SPIN. These two separate efforts were each designed to enable the easy development of software for managing SPARQL rules and constraint violations. They each include free software to do more with these generated error messages.

Using the Schemarama vocabulary, my ex203.rq query that checks for non-URI `dm:location` values might look like this:

```
# filename: ex211.rq

PREFIX sch: <http://purl.org/net/schemarama#>
PREFIX dm:  <http://learningsparql.com/ns/demo#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT
{
  [] rdf:type sch:Error;
       sch:message "location value should be a URI";
       sch:implicated ?s.

}
WHERE
```

```
{
  ?s dm:location ?city .
  FILTER (!(isURI(?city)))
}
```

This query uses a pair of square braces to represent a blank node instead of an underscore prefix. We learned about blank nodes in Chapter 2; in this case, the blank node groups together the information about the error found in the data.

The CONSTRUCT part creates a new member of the Schemarama `Error` class with two properties: a message about the error and a triple indicating which resource had the problem. The `Error` class and its properties are part of the Schemarama ontology, and the open source sparql-check utility that checks data against these rules will look for terms from this ontology in your SPARQL rules for instructions about the rules to execute. (The utility's default action is to output a nicely formatted report about problems that it found.)

I can express the same rule using the SPIN vocabulary with this query:

```
# filename: ex212.rq

PREFIX spin: <http://spinrdf.org/spin#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dm:   <http://learningsparql.com/ns/demo#>

CONSTRUCT
{
    _:b0 a spin:ConstraintViolation .
    _:b0 rdfs:comment "Location value must be a URI" .
    _:b0 spin:violationRoot ?this .

}
WHERE
{
    ?this dm:location ?city .
    FILTER (!isURI(?city)) .
}
```

Like the version that uses the Schemarama ontology, it creates a member of a class that represents violations. This new member of the `spin:ConstraintViolation` class is represented with a blank node as the subject and properties that describe the problem and point to the resource that has the problem.

SPIN stands for SPARQL Inferencing Notation, and its specification has been submitted to the W3C for potential development into a standard. Free and commercial software is currently available to provide a framework for the use of SPIN rules. (SPIN was developed at TopQuadrant, whose application development platform gives you a variety of ways to use triples generated about constraint violations.)

We saw earlier that SPARQL isn't only for querying data stored as RDF. This means that you can write CONSTRUCT queries to check other kinds of data for rule compliance, such as relational data made available to a SPARQL engine through the appropriate interface. This could be pretty valuable; there's a lot of relational data out there!

Asking for a Description of a Resource

The DESCRIBE keyword asks for a description of a particular resource, and according to the SPARQL 1.1 specification, "The description is determined by the query service." In other words, the SPARQL query processor gets to decide what information it wants to return when you send it a DESCRIBE query, so you may get different kinds of results from different processors.

For example, the following query asks about the resource *http://learningsparql.com/ns/data#course59*:

```
# filename: ex213.rq

DESCRIBE  <http://learningsparql.com/ns/data#course59>
```

The dataset in the ex069.ttl file includes one triple where this resource is the subject and three where it's the object. When we ask ARQ to run the query above against ex069.ttl, it gives us this response:

```
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix ab:      <http://learningsparql.com/ns/addressbook#> .

d:course59
      ab:courseTitle  "Using SPARQL with non-RDF Data" .
```

In other words, it returns the triple where that resource is a subject. (According to the program's documentation, "ARQ allows domain-specific description handlers to be written.")

On the other hand, when we send the following query to DBpedia, it returns all the triples that have the named resource as either a subject or object:

```
# filename: ex215.rq

DESCRIBE <http://dbpedia.org/resource/Joseph_Hocking>
```

A DESCRIBE query need not be so simple. You can pass it more than one resource URI by writing a query that binds multiple values to a variable and then asks the query processor to describe those values. For example, when you run the following query against the ex069.ttl data with ARQ, it describes `d:course59` and `d:course85`, which in ARQ's case, means that it returns all the triples that have these resources as subjects. These are the two courses that were taken by the person represented as `d:i0432`, Richard Mutt, because that's what the query asks for.

```
# filename: ex216.rq

PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX ab: <http://learningsparql.com/ns/addressbook#>

DESCRIBE ?course WHERE
{ d:i0432 ab:takingCourse ?course . }
```

For anything that I've seen a DESCRIBE query do, you could do the same thing and have greater control with a CONSTRUCT query, so I've never used DESCRIBE in serious application development. When checking out a SPARQL engine, though, it's worth trying out a DESCRIBE query or two to get a better feel for that query engine's capabilities.

Summary

In this chapter, we learned:

- How the first keyword after a SPARQL query's prefix declarations is called a query form, and how there are three besides SELECT: DESCRIBE, ASK, and CONSTRUCT.
- How a CONSTRUCT query can copy existing triples from a dataset.
- How you can create new triples with CONSTRUCT.
- How CONSTRUCT lets you convert data using one vocabulary into data that uses another.
- How ASK and CONSTRUCT queries can help to identify data that does not conform to rules that you specify.
- How the DESCRIBE query can ask a SPARQL processor for a description of a resource, and how different processors may return different things for the same resource in the same dataset.

CHAPTER 5

Datatypes and Functions

In earlier chapters we've already seen some use of datatypes and functions in SPARQL queries. These are overlapping topics, because queries often use functions to get the most value out of datatypes. In this chapter, we'll look at the big picture of what roles these topics play in SPARQL and the range of things they let you do:

- "Datatypes and Queries" on page 129: RDF supports a defined set of types as well as customized types, and your SPARQL queries can work with both.
- "Functions" on page 139: Functions let your queries find out about your input data, create new values from it, and gain greater control over typed data. In this section, we'll look at the functions defined by the SPARQL specification.
- "Extension Functions" on page 175: SPARQL implementations often add new functions to make your development easier. In this section, we'll see how to take advantage of these and what kinds of things they offer.

Datatypes and Queries

Does "20022" represent a quantity, a Washington DC postal code, or an ISO standard for financial services messaging? If we know that it's an integer, we know that it's more likely to represent a quantity. On the other hand, if we know that it's a string, it's more likely to be an identifier such as a postal code or a part number.

Decades before the semantic web, the storing of datatype metadata was one of the earliest ways to record semantic information. Knowing this extra bit of information about a piece of data gives you a better idea of what you can do with it, and this lets you build more efficient systems.

Different programming languages, markup languages, and query languages offer different sets of datatypes to choose from. When the W3C developed the XML Schema specification, they split the part about specifying datatypes for elements and attributes off from the part about specifying element structures, in case people wanted to use the datatypes specification separately from the structures part. The datatypes part—known as “XML Schema Part 2: Datatypes”—has become more popular than Part 1 of the spec, “Structures”. RDF uses Part 2. Or, in the more abstruse wording of the W3C’s RDF Concepts and Abstract Syntax Recommendation, “The datatype abstraction used in RDF is compatible with the abstraction used in XML Schema Part 2: Datatypes.”

A node in an RDF graph can be one of three things: a URI, a blank node, or a literal. If you assign a datatype to a literal value, we call it a typed literal; otherwise, it’s a plain literal.

Discussions are currently underway at the W3C about potentially doing away with the concept of the plain literal and just making `xsd:string` the default datatype, so that `"this"` and `"this"^^xsd:string` would mean the same thing.

According to the SPARQL specification, the following are the basic datatypes that SPARQL supports for typed literals:

- `xsd:integer`
- `xsd:decimal`
- `xsd:float`
- `xsd:double`
- `xsd:string`
- `xsd:boolean`
- `xsd:dateTime`

Other types, derived from these, are also available. The most important is `xsd:date`. (In the XML Schema Part 2: Datatypes specification, this is also a primitive type.) As its name implies, it’s like `xsd:dateTime` above, but you use it for values that do not include a time value—for example, “2011-10-13” instead of “2011-10-13T12:15:00”.

Here’s an example of some typed data that we saw in Chapter 2:

```
# filename: ex033.ttl

@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix d:   <http://learningsparql.com/ns/data#> .
@prefix dm:  <http://learningsparql.com/ns/demo#> .

d:item342 dm:shipped     "2011-02-14"^^<http://www.w3.org/2001/XMLSchema#date> .
d:item342 dm:quantity    "4"^^xsd:integer .
d:item342 dm:invoiced    "false"^^xsd:boolean .
d:item342 dm:costPerItem "3.50"^^xsd:decimal .
```

Here’s that chapter’s example of the same data in RDF/XML:

```
<!-- filename: ex035.rdf -->

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dm="http://learningsparql.com/ns/demo#"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema#">

  <rdf:Description rdf:about="http://learningsparql.com/ns/demo#item342">
    <dm:shipped rdf:datatype="xsd:date">2011-02-14</dm:shipped>
    <dm:quantity rdf:datatype="xsd:integer">4</dm:quantity>
    <dm:invoiced rdf:datatype="xsd:boolean">false</dm:invoiced>
    <dm:costPerItem rdf:datatype="xsd:decimal">3.50</dm:costPerItem>
  </rdf:Description>

</rdf:RDF>
```

To review a few things we learned about data typing in that chapter:

- In Turtle, the type identifier is shown after two carat (^) symbols.
- In RDF/XML, it's stored in an `rdf:type` attribute.
- The name of the datatype can be a full URI, as shown in the `dm:shipped` value in the ex033.ttl example above.
- It can also be a prefixed name, or a name with a prefix standing in for the URI that represents the name's namespace, like the types specified for the `dm:quantity`, `dm:invoiced`, and `dm:costPerItem` values.
- Just like the prefixes on the subjects, predicates, and objects of RDF triples, the `xsd:` prefix on a datatype must also be declared.

When you leave the quotation marks off of a Turtle literal, a processor makes certain assumptions about the datatype if the value is the word "true" or "false" or if it's a number. Because of this, the following would be interpreted the same way as the two examples above:

```
# filename: ex034.ttl

@prefix d:   <http://learningsparql.com/ns/data#> .
@prefix dm: <http://learningsparql.com/ns/demo#> .

d:item342 dm:shipped     "2011-02-14"^^<http://www.w3.org/2001/XMLSchema#date> .
d:item342 dm:quantity    4 .
d:item342 dm:invoiced    true .
d:item342 dm:costPerItem 3.50 .
```

Because the `:` in ex034.ttl and `d:` prefix in ex033.ttl both stand for the same URI, `:item342` in ex034.ttl and `d:item342` in ex033.ttl still represent the same resource.

The ex201.rq query in Chapter 4, reproduced below, was an interesting example of how these abbreviated ways to represent types can be used. Its FILTER line was looking for values that didn't have a type of `xsd:integer`, and none of the values in the ex198.ttl data file had their types explicitly identified as an `xsd:integer`.

```
# filename: ex201.rq
```

```
PREFIX dm:  <http://learningsparql.com/ns/demo#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

ASK WHERE
{
  ?item dm:amount ?amount .
  FILTER ((datatype(?amount)) != xsd:integer)
}
```

The query engine still knew which `?amount` values were integers and which were not, because any unquoted series of digits with no period is treated as an integer.

Most of your work with datatypes in SPARQL will involve the use of functions that are covered in more detail in the next section. Before we look at any of those, it's a good idea to know how representations of typed literals in your queries interact with different kinds of literals in your dataset. Let's look at what a few queries do with this set of data:

```
# filename: ex217.ttl

@prefix d:   <http://learningsparql.com/ns/data#> .
@prefix dm:  <http://learningsparql.com/ns/demo#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix mt:  <http://learningsparql.com/ns/mytypesystem#> .

d:item1a dm:prop "1" .
d:item1b dm:prop "1"^^xsd:integer .
d:item1c dm:prop 1 .
d:item1d dm:prop 1.0e5 .
d:item2a dm:prop "two" .
d:item2b dm:prop "two"^^xsd:string .
d:item2c dm:prop "two"^^mt:potrzebies .
d:item2d dm:prop "two"@en .
```

Which items of ex217.ttl do you think the following query will retrieve? (Hint: look over ex033.ttl and ex034.ttl again, keeping in mind that they represent the same triples.)

```
# filename: ex218.rq

PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

SELECT ?s
WHERE { ?s ?p 1 . }
```

If you guessed `d:item1b` and `d:item1c`, you were right:

```
------------
| s        |
============
| d:item1c |
| d:item1b |
------------
```

The `d:item1a` value is the string "1", and because the object of the triple pattern in the query isn't quoted, it represents the integer 1. The `d:item1b` value is enclosed in quotes, but it has the prefixed name xsd:integer after two carat symbols to show that it should be treated as an integer.

The following query returns d:item2a and d:item2b, even though the object of one of those triples includes the xsd:string type designation and the other doesn't:

```
# filename: ex220.rq

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX d:   <http://learningsparql.com/ns/data#>
PREFIX dm:  <http://learningsparql.com/ns/demo#>

SELECT ?s
WHERE { ?s ?p "two" . }
```

This next query returns the same two results:

```
# filename: ex221.rq

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

SELECT ?s
WHERE { ?s ?p "two"^^xsd:string . }
```

In other words, a quoted value is a string whether you specifically designate it as an xsd:string or not.

A number written with a decimal point and the letter "e" to express an exponent (for example, 1.0e5 in ex217.ttl, which represents the value 100,000) is treated as a double precision floating point number (xsd:double).

The remaining values in ex217.ttl would have to be retrieved with the types explicitly specified using either the full URI of the datatype name or a prefixed name version of the URI. For example, the following would retrieve only one item, d:item2c:

```
# filename: ex222.rq

PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX mt: <http://learningsparql.com/ns/mytypesystem#>

SELECT ?s
WHERE { ?s ?p "two"^^mt:potrzebies . }
```

It's an interesting case, because it has the ^^ in it to indicate that the value is of a specific type, but it's not an xsd type. RDF lets you define custom datatypes for your own needs, and as this query demonstrates, SPARQL lets you query for them. (We'll learn how to query for d:item2d, which has the @en tag to show that it's in English, in "Checking, Adding, and Removing Spoken Language Tags" on page 157.)

The Potrzebie System of Weights and Measures was developed by noted computer scientist Donald Knuth. He published it as a teenager in Mad Magazine in 1957, so it is not considered normative. A single potrzebie is the thickness of Mad magazine issue number 26.

The use of non-XSD types in RDF is currently most common in data using the SKOS standard for controlled vocabularies. In SKOS, the `skos:notation` property names an identifier for a concept that is often a legacy value from a different thesaurus expressed as a cryptic numeric sequence (for example, "920" to represent biographies in the library world's Dewey Decimal System), unlike the concept's `skos:prefLabel` property that provides a more human-readable name. For example, a version of the UN Food and Agriculture Organization's AGROVOC SKOS thesaurus about food production terminology declares a subproperty of `skos:notation` called `asfaCode` to store a term's identifier from the Aquatic Sciences and Fisheries Abstracts (ASFA) thesaurus. It declares a special datatype called `ASFACode` for the values of this property so that you know that a value like "asf4523" is not just any string, but has this specialized type:

```
:asfaCode       rdfs:subPropertyOf   skos:notation
:an_agrovoc_uri :asfaCode            "asf4523"^^:ASFACode
```

Other examples of custom datatypes include `t:kilos` and `t:liters` in the SPARQL specification; in these examples the `t:` prefix refers to the *http://example.org/types#* namespace, which means that it was made up for the purposes of the example.

If you wanted to ignore the types and just retrieve everything with a value of "two", you can tell the query to just treat everything like a string using the `str()` function that we'll learn about in "Node Type Conversion Functions" on page 146:

```
# filename: ex223.rq

SELECT ?s
WHERE
{
  ?s ?p ?o .
  FILTER (str(?o) = "two")
}
```

Representing Strings

The sample string data that we've seen so far in this book's example Turtle data files has always been enclosed by double quotes, "like this". You can also enclose strings in Turtle and SPARQL with apostrophes, or single quotes, 'like this'.

In RDF/XML, representation of strings of character data follow the normal XML rules: they are shown as character data between tags or enclosed by double or single quotes in attribute values.

In Turtle and SPARQL, if you begin and end a string with three double quotes, RDF processors will preserve the carriage returns, which is handy for longer blocks of text. (In SPARQL, you can do the same with three single quotes, but not in Turtle.) If you must include a double quote as part of your SPARQL or Turtle string, you can escape it with a backslash character, and in SPARQL you can do the same with a single quote.

The following demonstrates several possible ways to represent strings:

```
# filename: ex224.ttl

@prefix d:    <http://learningsparql.com/ns/data#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

d:item1 rdfs:label "sample string 1" .
d:item2 rdfs:label 'sample string 2' .
d:item3 rdfs:label 'These quotes are "ironic" quotes.' .
d:item4 rdfs:label "These too are \"ironic\" quotes." .
d:item5 rdfs:label "McDonald's is not my kind of place." .
d:item6 rdfs:label """this

has two carriage returns in the middle.""" .
```

This simple query retrieves all the subjects and objects from any dataset and shows us how a SPARQL processor interprets the strings in the sample data above:

```
# filename: ex225.rq

PREFIX d: <http://learningsparql.com/ns/data#>

SELECT ?s ?o
WHERE { ?s ?p ?o }
```

When formatting the strings for output, ARQ uses double quotes to delimit strings. It uses the backslash as an escape character and represents carriage returns as `\r` and line feeds as `\n`; other SPARQL processors may do it differently:

```
-----------------------------------------------------------------
| s       | o                                                   |
=================================================================
| d:item3 | "These quotes are \"ironic\" quotes."               |
| d:item1 | "sample string 1"                                   |
| d:item6 | "this\r\n\r\nhas two carriage returns in the middle."|
| d:item4 | "These too are \"ironic\" quotes."                  |
| d:item2 | "sample string 2"                                   |
| d:item5 | "McDonald's is not my kind of place."               |
-----------------------------------------------------------------
```

This output also reminds us that like the rows of a relational database table, the ordering of a set of triples doesn't matter, unless you choose to sort them with an ORDER BY phrase in your query. (For more on this, see "Sorting Data" on page 89.)

Comparing Values and Doing Arithmetic

"FILTERing Data Based on Conditions" on page 73 showed that comparison operators are a classic way to retrieve data based on certain conditions. To experiment a little more, we'll use the ex138.ttl dataset from the same chapter, but modified to include data typing metadata with the `e:date` values and ":00" added to the end of each date-time value to include the number of seconds, as the `xsd:dateTime` format expects:

```
# filename: ex227.ttl

@prefix e:   <http://learningsparql.com/ns/expenses#> .
@prefix d:   <http://learningsparql.com/ns/data#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

d:m40392 e:description "breakfast" ;
         e:date "2011-10-14T08:53:00"^^xsd:dateTime ;
         e:amount 6.53 .

d:m40393 e:description "lunch" ;
         e:date "2011-10-14T13:19:00"^^xsd:dateTime ;
         e:amount 11.13 .

d:m40394 e:description "dinner" ;
         e:date "2011-10-14T19:04:00"^^xsd:dateTime ;
         e:amount 28.30 .
```

Our first query asks for all the data for each entry that has an `e:amount` value less than 20:

```
# filename: ex228.rq

PREFIX d:   <http://learningsparql.com/ns/data#>
PREFIX e:   <http://learningsparql.com/ns/expenses#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?s ?p ?o
WHERE {
  ?s e:amount ?amount;
     ?p ?o .
  FILTER (?amount < 20)
}
```

We get data for the `d:m40393` and `d:m40392` entries, as you might have guessed:

```
-------------------------------------------------------------------
| s        | p             | o                                    |
===================================================================
| d:m40393 | e:amount      | 11.13                                |
| d:m40393 | e:date        | "2011-10-14T13:19:00"^^xsd:dateTime |
| d:m40393 | e:description | "lunch"                              |
| d:m40392 | e:amount      | 6.53                                 |
| d:m40392 | e:date        | "2011-10-14T08:53:00"^^xsd:dateTime |
| d:m40392 | e:description | "breakfast"                          |
-------------------------------------------------------------------
```

Our second query asks for all the meals that took place at noon or later on October 14th, 2011:

```
# filename: ex230.rq

PREFIX d:   <http://learningsparql.com/ns/data#>
PREFIX e:   <http://learningsparql.com/ns/expenses#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?s ?p ?o
WHERE {
  ?s e:date ?date;
     ?p ?o .
  FILTER (?date >= "2011-10-14T12:00:00"^^xsd:dateTime)
}
```

This retrieves data for the `d:m40394` and `d:m40393` entries:

```
-----------------------------------------------------------------
| s        | p             | o                                  |
=================================================================
| d:m40394 | e:amount      | 28.30                              |
| d:m40394 | e:date        | "2011-10-14T19:04:00"^^xsd:dateTime |
| d:m40394 | e:description | "dinner"                           |
| d:m40393 | e:amount      | 11.13                              |
| d:m40393 | e:date        | "2011-10-14T13:19:00"^^xsd:dateTime |
| d:m40393 | e:description | "lunch"                            |
-----------------------------------------------------------------
```

That chapter also showed us some arithmetic being performed in "Combining Values and Assigning Values to Variables" on page 86, where the following query did some addition and multiplication to calculate the tip and total values for these meals:

```
# filename: ex139.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT ?description ?amount ((?amount * .2) AS ?tip)
       ((?amount + ?tip) AS ?total)
WHERE
{
  ?meal e:description ?description ;
        e:amount ?amount .
}
```

Along with + for addition, - for subtraction, and * for multiplication, you can use / for division.

The parentheses in the expressions (?amount * .2) and (?amount + ?tip) are unnecessary in this particular case—you'd get the same query results without them—but as with many mathematical expressions, they make it easier to see what's going on. Also, ((?amount + ?tip) AS ?total) is on a separate line only to more easily fit on the page. It's part of the SELECT list, just like ?description and ?amount. The extra white space doesn't affect the query's execution.

Arithmetic expressions are especially useful when you use BIND to create a new value, like in this revision of the ex139.rq query above, which produces the same result when run on the ex138.ttl dataset that we used with ex139.rq:

```
# filename: ex232.rq

PREFIX e: <http://learningsparql.com/ns/expenses#>

SELECT ?description ?amount ?tip ?total

WHERE {
  ?meal e:description ?description ;
        e:amount ?amount .
  BIND ((?amount * .2) AS ?tip)
  BIND ((?amount + ?tip) AS ?total)

}
```

When different values are explicitly typed with different numeric types, you can still use them together when performing arithmetic. For example, the ex033.ttl dataset near the beginning of this chapter has a dm:quantity value specified as an xsd:integer and a dm:costPerItem value specified as an xsd:decimal, but the following query multiplies them together:

```
# filename: ex233.rq

PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX d:  <http://learningsparql.com/ns/data#>
SELECT *
WHERE {
  ?item dm:quantity ?quantity;
        dm:costPerItem ?cost .
  BIND ( (?quantity * ?cost) as ?total )
}
```

ARQ has no problem running this query with the ex033.ttl data:

```
---------------------------------------
| item      | quantity | cost | total |
=======================================
| d:item342 | 4        | 3.50 | 14.00 |
---------------------------------------
```

Functions

Functions perform a variety of jobs for us, such as program logic, math, string manipulation, and checking whether certain conditions are true or not. Most of SPARQL 1.0's functions fell into in this last category, because with no equivalent of SPARQL 1.1's BIND keyword to assign new values to variables, there was little reason to include functions for manipulating input values and returning new values.

Because of this, most functions in 1.0 are what the spec calls "test functions" that each answer a particular question about a value. Used in a FILTER statement, these give you more control over exactly what your queries retrieve. Most of these are boolean functions such as `regex()`, and the 1.0 ones that aren't boolean answer questions about a value passed to them such as what datatype it is or what language tag may have been assigned to a string.

The SPARQL spec tells us that along with built-in functions for testing values, "SPARQL provides the ability to invoke arbitrary functions, including a subset of the XPath casting functions." ("Casting" here refers to the conversion of one datatype to another.) The "arbitrary" part of "invoke arbitrary functions" means that a SPARQL processor can offer any extension function that its implementers want to include.

The W3C XPath language gives you a way to describe sets of nodes in a tree representation of an XML document. For example, an XPath expression could refer to all the sibling nodes that precede the current node's parent node, so that when a processor such as an XSLT engine traverses an XML document it can pull values from those nodes to process the node it's currently working on. The original 1999 XPath Recommendation defined this path language and some functions for operating on values. XPath 2.0 defined many more functions, and the newer XQuery spec referenced many of them, so the W3C split "XQuery 1.0 and XPath 2.0 Functions and Operators" out into its own specification.

The SPARQL 1.0 spec names a few basic functions and SPARQL 1.1 offers a wider selection, nearly all of which are based on XPath functions. (They don't always have the same name—for example, the SPARQL 1.1 spec says that its `minutes()` function "corresponds to [the XPath function] fn:minutes-from-dateTime.")

In the SPARQL specifications, some function names are written in all uppercase, some in lowercase, and some in mixed case, with no apparent pattern. I've written each function name the same way it's written in the spec, although a SPARQL processor won't care. For example, it doesn't make any difference whether you write the substring function as `SUBSTR()` or `substr()`.

Program Logic Functions

The `IF()` and `COALESCE()` functions each evaluate one or more expressions and return values based on what they find. This lets you pack a lot of program logic into a brief expression.

> **1.1 Alert**
>
> Both `IF()` and `COALESCE()` are new features of SPARQL 1.1.

The **`IF()`** function takes three arguments. If the first one evaluates to a boolean `true`, the function returns the value of the second argument; otherwise, it returns the third.

Here's a simple example that you can run with any input file (because no patterns in the query try to match any input data, the input will be ignored):

```
# filename: ex235.rq

SELECT ?answer
WHERE
{
  BIND ((IF (2 > 3, "Two is bigger","Three is bigger")) AS ?answer)
}
```

The `IF()` function will bind either the string "Two is bigger" or the string "Three is bigger" to the `?answer` variable, depending on whether the expression `2 > 3` is true. The result is not surprising:

```
---------------------
| answer            |
=====================
| "Three is bigger" |
---------------------
```

All three parameters can be as complex as you like. Let's look at an example that does some real work by using several other functions described later in this chapter. For each `dm:location` value in the following ex104.ttl sample dataset, which comes from Chapter 3, I want to create a new triple that says that the value is an instance of the `dm:Place` class:

```
# filename: ex104.ttl

@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:item432 dm:cost 8 ;
          dm:location <http://dbpedia.org/resource/Boston> .
d:item857 dm:cost 12 ;
          dm:location <http://dbpedia.org/resource/Montreal> .
d:item693 dm:cost 10 ;
          dm:location "Heidelberg" .
```

```
d:item126 dm:cost 5 ;
          dm:location <http://dbpedia.org/resource/Lisbon> .
```

Following the convention of popular object-oriented languages, class names like `dm:Place` usually begin with an uppercase letter and property names such as `dm:location` begin with a lowercase letter.

If we create triples with the ex104.ttl dataset's `dm:location` values as subjects, that will work for three of those items. It won't work for the one with "Heidelberg" as a value, because "Heidelberg" is a string and the subject of a triple must be a URI. To work around this, the following query checks whether the value is a proper URI, and if not, it creates one from the string.

```
# filename: ex237.rq

PREFIX dm:   <http://learningsparql.com/ns/demo#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT { ?locationURI rdf:type dm:Place . }
WHERE
{
  ?item dm:location ?locationValue .
  BIND (IF(isURI(?locationValue),
               ?locationValue,
               URI(CONCAT("http://learningsparql.com/ns/data#",
                          ENCODE_FOR_URI(?locationValue)))
           ) AS ?locationURI

       ) .
}
```

In function expressions, you can put all the white space you want before and after parentheses and the commas that separate parameters. This makes complex expressions easier to read.

After binding each `dm:location` value to the `?locationValue` variable in the first triple pattern, the SPARQL processor will bind the results of the `IF()` function to the `?locationURI` variable. The `IF()` function has three parameters:

1. The first expression uses the `isURI()` function, which we'll learn more about in the next section, to check whether `?locationValue` is a proper URI.
2. If the first parameter's expression evaluates to `true`, then we know that `?locationValue` would work as the subject of the triple being created, so it's the second argument to the `IF()` function and will get bound to `?locationURI`.
3. If the `IF()` function's first parameter's expression evaluates to `false`, the function returns the third parameter: an expression that creates a URI from

the ?locationValue. The ENCODE_FOR_URI() function escapes any characters that may cause problems in the path part of a URI; we don't really need it for "Heidelberg", but if the string "Los Angeles" came up, the space might be a problem, and ENCODE_FOR_URI() would convert that string to "Los%20Angeles". The CONCAT() function concatenates the value returned by ENCODE_FOR_URI() onto the string "http://learningsparql.com/ns/data#", and then the URI() function (also covered in the next section) converts the result from a string to a URI.

The query result has a triple for each of the four subjects in the input document:

```
@prefix d:      <http://learningsparql.com/ns/data#> .
@prefix dm:     <http://learningsparql.com/ns/demo#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://dbpedia.org/resource/Boston>
      rdf:type      dm:Place .

d:Heidelberg
      rdf:type      dm:Place .

<http://dbpedia.org/resource/Montreal>
      rdf:type      dm:Place .

<http://dbpedia.org/resource/Lisbon>
      rdf:type      dm:Place .
```

As you learn about more functions to use in SPARQL queries, remember that you can use any of them—including more IF() function calls—inside of the arguments passed to an IF() function call.

Boolean values can be combined in SPARQL with the && operator for "and" and || for "or."

The **COALESCE()** function has roots in the SQL world. Give it as many parameters as you want, and it will return the first one that doesn't result in an error. This makes it a nice way to say "try this, and if that doesn't work try this, and if that doesn't work..." In a SPARQL query, the parameter expressions that may or may not work are often variables that may or may not be bound, depending on whether the right pattern of data comes along.

For example, in "Data That Might Not Be There" on page 53, we saw one way to use the OPTIONAL keyword to get the ab:nick value for each person in the following data if it's there, and if it's not, to get the ab:firstName value instead:

```
# filename: ex054.ttl
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d:  <http://learningsparql.com/ns/data#> .
```

```
d:i0432 ab:firstName "Richard" .
d:i0432 ab:lastName  "Mutt" .
d:i0432 ab:homeTel   "(229) 276-5135" .
d:i0432 ab:nick      "Dick" .
d:i0432 ab:email     "richard49@hotmail.com" .

d:i9771 ab:firstName "Cindy" .
d:i9771 ab:lastName  "Marshall" .
d:i9771 ab:homeTel   "(245) 646-5488" .
d:i9771 ab:email     "cindym@gmail.com" .

d:i8301 ab:firstName "Craig" .
d:i8301 ab:lastName  "Ellis" .
d:i8301 ab:workTel   "(245) 315-5486" .
d:i8301 ab:email     "craigellis@yahoo.com" .
d:i8301 ab:email     "c.ellis@usairwaysgroup.com" .
```

The following query binds the `?first` variable to the `ab:firstName` value and tries to bind the `?nickname` variable to the `ab:nick` value, if it's there. The `COALESCE()` function then returns the `?nickname` value if it can, and otherwise returns the `?first` value. The returned value gets bound to the `?firstName` value for output.

```
# filename: ex239.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT ?firstName ?last
WHERE {
  ?s ab:lastName ?last;
     ab:firstName ?first .
  OPTIONAL {
    ?s ab:nick ?nickname .
  }
  BIND (COALESCE(?nickname,?first) AS ?firstName)
}
```

The result shows that for resource `d:i0432` the query found and used the `ab:nick` value of "Dick". For each of the other resources, it used the `ab:firstName` value, because they didn't have `ab:nick` values:

```
--------------------------
| firstName | last       |
==========================
| "Craig"   | "Ellis"    |
| "Cindy"   | "Marshall" |
| "Dick"    | "Mutt"     |
--------------------------
```

The example above passes only two parameters to `COALESCE()`, but you can add as many as you like for it to test. Also, as with the `IF()` function, you can pass much more complex expressions as parameters.

Node Type and Datatype Checking Functions

Certain functions expect some of their parameters to be of specific datatypes. For example, you can't ask the `round()` function to round off the string "hello" to the nearest integer. Your application may also expect certain pieces of data to be of specific types; if you're going to add up the total amount spent on breakfasts in a set of expense report data like ex145.ttl, you want to make sure that each `e:amount` value is an actual number and not a string like "5 bucks".

To keep this kind of data from causing problems, SPARQL offers functions to check whether subjects and objects qualify as URIs, literals, numeric literals, or blank nodes. (You can also test a predicate, but if it's anything but a URI, you don't have much of a triple there.) If a value is a typed literal, the `datatype()` function lets you find out what type it is. These functions are especially valuable in data quality rules that you create to identify data that you hadn't planned for, as described in "Finding Bad Data" on page 117 of the previous chapter.

Functions for checking node types and datatypes are also handy in FILTER statements when you want your query to only retrieve triples meeting certain conditions.

Functions that check whether something has a particular node type or datatype have a name that begins with "is" (for example, `isNumeric()`, and they return a boolean `true` or `false` value. Let's try out these functions with this sample data:

```
# filename: ex241.ttl

@prefix dm:   <http://learningsparql.com/ns/demo#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:    <http://learningsparql.com/ns/data#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

d:id1 dm:location _:b1 .
d:id2 dm:location <http://dbpedia.org/resource/Montréal> .
d:id3 dm:amount 3 .
d:id4 dm:amount "4"^^xsd:integer .
d:id5 dm:amount 1.0e5 .
d:id6 rdfs:label "5 bucks" .
d:id7 dm:shipped true  .
```

To check the type of the value in each of these triples' objects, the following query sets several variables depending on what the **`isBlank()`**, **`isLiteral()`**, **`isNumeric()`**, **`isIRI()`**, and **`isURI()`** functions return for that value:

```
# filename: ex242.rq

PREFIX dbr: <http://dbpedia.org/resource/>
SELECT ?o ?blankTest ?literalTest ?numericTest ?IRITest ?URITest
WHERE
```

```
{
  ?s ?p ?o .
  BIND (isBlank(?o) as ?blankTest)
  BIND (isLiteral(?o) as ?literalTest)
  BIND (isNumeric(?o) as ?numericTest)
  BIND (isIRI(?o) as ?IRITest)
  BIND (isURI(?o) as ?URITest)
}
```

The result is a table of what these functions do:

```
----------------------------------------------------------------------------
| o            | blankTest | literalTest | numericTest | IRITest | URITest |
============================================================================
| 3            | false     | true        | true        | false   | false   |
| _:b0         | true      | false       | false       | false   | false   |
| "5 bucks"    | false     | true        | false       | false   | false   |
| 4            | false     | true        | true        | false   | false   |
| dbr:Montréal | false     | false       | false       | true    | true    |
| true         | false     | true        | false       | false   | false   |
| 1.0e5        | false     | true        | true        | false   | false   |
----------------------------------------------------------------------------
```

1.1 Alert

Of the functions demonstrated above, only `isNumeric()` is new for SPARQL 1.1.

There are a few interesting things to note about the results:

- Numbers, strings, and the keywords `true` and `false` (written in all lowercase) are all literals. Only URIs and blank nodes are not.
- There's no difference between `isIRI()` and `isURI()`. They're synonyms, both provided because "IRI" is a more technically correct term while "URI" is a more commonly used term. Although *http://dbpedia.org/resource/Montréal* is not really a URI because of the accented character, `isURI()` returns `true` for it just like `isIRI()` does.

Earlier in this chapter, "Program Logic Functions" on page 140 has a good example of `isURI()` being put to work to check out whether a string needs to be converted to a URI.

Neither RDFS nor OWL provides an explicit way to say that the value of a particular property must be of a given type, but now you have a way to check: by using these functions with the SPARQL rules described in Chapter 4.

The **`datatype()`** functions returns a URI identifying the datatype of the passed parameter. The following query tells us the datatype of the object of each triple that it reads:

```
# filename: ex244.rq
```

```
PREFIX dbr: <http://dbpedia.org/resource/>

SELECT ?o ?datatype
WHERE
{
  ?s ?p ?o .
  BIND (datatype(?o) as ?datatype)
}
```

Running this query with the ex241.ttl data above gives us this result:

```
-------------------------------------------------------------
| o            | datatype                                   |
=============================================================
| 3            | <http://www.w3.org/2001/XMLSchema#integer> |
| _:b0         |                                            |
| "5 bucks"    | <http://www.w3.org/2001/XMLSchema#string>  |
| 4            | <http://www.w3.org/2001/XMLSchema#integer> |
| dbr:Montréal |                                            |
| true         | <http://www.w3.org/2001/XMLSchema#boolean> |
| 1.0e5        | <http://www.w3.org/2001/XMLSchema#double>  |
-------------------------------------------------------------
```

For each value with an identifiable datatype assigned, the query result has the XML Schema Part 2 URI for that datatype. For URIs and blank nodes, it shows nothing.

Another useful function for checking on a variable is **bound()**, which tells us whether a variable has a value bound to it. You can find some classic examples of how this function is used in "Finding Data That Doesn't Meet Certain Conditions" on page 57.

Node Type Conversion Functions

SPARQL offers functions to convert (or "cast") types—not only between XML Schema Part 2 datatypes like `xsd:string` and `xsd:integer`, but also between RDF node types, strings, and URIs. These functions can't work miracles such as casting the string "5 bucks" to an integer, but they can cast the string "5" to one, and they can cast the string "http://www.learningsparql.com" to a URI.

We saw in Chapter 2 that although a triple's object can be either a URI or a literal, it's better for it to be a URI, because it can serve as the subject of other triples. When the same URI is the object of some triples and the subject of others, you can link these triples, do inferencing, and get more out of your data.

The **URI()** function (a synonym for the **IRI()** function, just as `isURI()` is a synonym for `isIRI()`) lets you convert values to URIs if possible. The following copies triples from the input, substituting a `URI()` version of the object in the output:

```
# filename: ex246.rq

BASE <http://learningsparql.com/ns/demo#>

CONSTRUCT {?s ?p ?testURI.}
WHERE
```

```
{
  ?s ?p ?o .
  BIND (URI(?o) AS ?testURI)
}
```

Let's look at what this query does with the ex241.ttl input from above before discussing how it works:

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
@prefix dm:      <http://learningsparql.com/ns/demo#> .

d:id2
      dm:location   <http://dbpedia.org/resource/Montréal> .

d:id1
      dm:location   <_:5d37e0ec:12fd6bd6a74:-7ffe> .

d:id6
      rdfs:label    <http://learningsparql.com/ns/5 bucks> .
```

Here's what happened. Keep in mind that the SPARQL 1.1 specification tells us that "Passing any RDF term [to the `IRI()` or `URI()` functions] other than a simple literal or an IRI is an error":

- It couldn't convert the integer object values with the `d:id3` and `d:id4` triples or the boolean value with `d:id7` to URIs, so there are no output triples for those. (These values are literals, but not simple literals—they're typed literals.)
- It converted the blank node with `d:id1` into… well, it doesn't really matter, because a blank node is not a literal or a URI.
- The URI with `d:id2` came out unchanged.
- To treat the string "5 bucks" as a URI, the processor treats it as a relative URI. Relative to what? The beginning of the query specifies a base URI with the `BASE` keyword, so ARQ used that; without it, ARQ would use a base URI of `file:///` and the directory where the query file is stored to create the URI. You might think that appending "5 bucks" to that base URI would result in a URI of `http://learningsparql.com/ns/demo#5 bucks`, but because the result of the `URI()` and `IRI()` functions "must result in an absolute IRI," according to spec (thereby precluding the use the pound sign), it ends up as `http://learningsparql.com/ns/5 bucks`.

1.1 Alert

`URI()` and `IRI()` are new for SPARQL 1.1.

I don't like the URI *http://learningsparql.com/ns/5 bucks* because of the space in it. The following revision of the query uses the `ENCODE_FOR_URI()` function that we saw in

"Program Logic Functions" on page 140 to escape any URI-unfriendly characters like the space in "5 bucks". The `URI()` function then converts the result of the `ENCODE_FOR_URI()` function call to a URI.

```
# filename: ex248.rq

BASE <http://learningsparql.com/ns/demo#>

CONSTRUCT {?s ?p ?testURI.}
WHERE
{
  ?s ?p ?o .
  BIND (URI(ENCODE_FOR_URI(?o)) AS ?testURI)
}
```

If you pass anything but a literal or an `xsd:string` value to `ENCODE_FOR_URI()`, ARQ throws an error, so to test ex248.rq I made an alternative version of the ex241.ttl input data that doesn't have `d:id1` or `d:id2`:

```
# filename: ex249.ttl

@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:   <http://learningsparql.com/ns/data#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

d:id3 dm:amount 3 .
d:id4 dm:amount "4"^^xsd:integer .
d:id5 dm:amount 1.0e5 .
d:id6 rdfs:label "5 bucks" .
d:id7 dm:shipped true  .
```

The result ignores the numeric and boolean values and does percent sign escaping for the space:

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
@prefix dm:      <http://learningsparql.com/ns/demo#> .

d:id6
      rdfs:label    <http://learningsparql.com/ns/5%20bucks> .
```

Before passing a value to the `URI()` or `IRI()` function, it's a good idea to prepare it with the `ENCODE_FOR_URI()` function, but make sure you're not passing it anything but simple literals or `xsd:string` values.

The **`str()`** function returns a string representation of the argument passed to it. (Technically, it returns the "lexical form" of a literal or the "codepoint representation" of an IRI, but for practical purposes, just think of it as a string version of the argument passed to it.)

The following query passes the object of each triple that it reads to the `str()` function and stores the result in the `?testStr` variable:

```
# filename: ex251.rq

PREFIX d:   <http://learningsparql.com/ns/data#>

SELECT ?s ?testStr
WHERE
{
  ?s ?p ?o .
  BIND (str(?o) AS ?testStr)
}
```

When run with the ex241.ttl dataset from above, the query gives us this result:

```
--------------------------------------------------
| s      | testStr                               |
==================================================
| d:id3 | "3"                                    |
| d:id1 |                                        |
| d:id6 | "5 bucks"                              |
| d:id4 | "4"                                    |
| d:id2 | "http://dbpedia.org/resource/Montréal" |
| d:id7 | "true"                                 |
| d:id5 | "1.0e5"                                |
--------------------------------------------------
```

It's pretty straightforward: it returns nothing when a blank node is passed to it and a string representation of anything else.

This looks pretty simple, but it can be very helpful, especially when combined with the functions described later in "String Functions" on page 164. For example, earlier I had to create the ex249.ttl dataset as an alternative to ex241.ttl because the `ENCODE_FOR_URI()` function expected a string parameter and some of the ex241.ttl values were not strings and would therefore cause an error if passed to this function. The revision of ex251.rq below wraps the parameter passed to the `ENCODE_FOR_URI()` function with the `str()` function so that nonstring values don't trigger errors:

```
# filename: ex253.rq

BASE <http://learningsparql.com/ns/demo#>

CONSTRUCT {?s ?p ?testURI.}
WHERE
{
  ?s ?p ?o .
  BIND (URI(ENCODE_FOR_URI(str(?o))) AS ?testURI)
}
```

When we run this query with the ex241.ttl data, we see that the SPARQL processor didn't do anything with the blank node in the `d:id1` triple, and it did predictable transformations of everything else, except maybe for the URI value that went with `d:id2`:

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
@prefix dm:      <http://learningsparql.com/ns/demo#> .

d:id5
      dm:amount     <http://learningsparql.com/ns/1.0e5> .

d:id3
      dm:amount     <http://learningsparql.com/ns/3> .

d:id2
      dm:location
        <http://learningsparql.com/ns/demo#http%3A%2F%2Fdbpedia%2Eorg%2Fresource%2FMontréal> .

d:id4
      dm:amount     <http://learningsparql.com/ns/4> .

d:id6
      rdfs:label    <http://learningsparql.com/ns/5%20bucks> .

d:id7
      dm:shipped    <http://learningsparql.com/ns/true> .
```

How did the URI end up looking like that? Let's step through the three functions executed on on the *http://dbpedia.org/resource/Montréal* URI in the BIND line of ex253.rq in the order that they occurred:

1. The `str()` function converted it to the string "http://dbpedia.org/resource/Montréal".
2. The `ENCODE_FOR_URI()` function, which expects a simple literal as input, escaped all the characters that would cause a problem if the string was used in the path part of a URI. It did this by converting each of those characters to a percent sign followed by a hexadecimal number representing that character's code point. This included the colon and slashes, so that, for example "http://" ended up as "http%3A%2F%2F".
3. The `URI()` function didn't know that this value had started off as a URI, and thought it was a regular string, so it appended the result of the `ENCODE_FOR_URI()` function call to the base URI declared at the beginning of the query, just like ex246.rq did with the string "5 bucks".

How can we tell the query processor not to do all this if the object value is already a URI? By using the `IF()` and `isURI()` functions that we learned about earlier in this chapter:

```
# filename: ex255.rq

BASE <http://learningsparql.com/ns/demo#>

CONSTRUCT {?s ?p ?testURI.}
WHERE
```

```
{
  ?s ?p ?o .
  BIND( IF(isURI(?o),
           ?o,
           URI(ENCODE_FOR_URI(str(?o)))
          ) AS ?testURI
      )
}
```

In this query, the `IF()` function's first parameter checks whether `?o` is a URI, and if so, it returns the second parameter: `?o`, unchanged by any functions. If it's not a URI, the third parameter, which does all the transformations we saw in ex253.rq (and which worked so well for literal input) gets returned, and we get sensible output:

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
@prefix dm:      <http://learningsparql.com/ns/demo#> .

d:id5
      dm:amount     <http://learningsparql.com/ns/1.0e5> .

d:id3
      dm:amount     <http://learningsparql.com/ns/3> .

d:id2
      dm:location   <http://dbpedia.org/resource/Montréal> .

d:id4
      dm:amount     <http://learningsparql.com/ns/4> .

d:id6
      rdfs:label    <http://learningsparql.com/ns/5%20bucks> .

d:id7
      dm:shipped    <http://learningsparql.com/ns/true> .
```

Datatype Conversion

When converting one typed node to another—for example, when converting an integer to a string—if you know the type you want to convert to, you know the function you need to convert it, because it's the type name. The following list of functions should be familiar; I just copied the list of datatypes from the beginning of this chapter and added parentheses after each:

- `xsd:integer()`
- `xsd:decimal()`
- `xsd:float()`
- `xsd:double()`
- `xsd:string()`

- `xsd:boolean()`
- `xsd:dateTime()`

To be consistent with the use of XML Schema Part 2 datatypes, SPARQL uses casting functions from the XPath specification.

Let's try to convert the objects of the triples from ex241.ttl to the four numeric types with the following query:

```
# filename: ex257.rq

PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?o ?integerTest ?decimalTest ?floatTest ?doubleTest
WHERE
{
  ?s ?p ?o .
  BIND (xsd:integer(?o) as ?integerTest)
  BIND (xsd:decimal(?o) as ?decimalTest)
  BIND (xsd:float(?o) as ?floatTest)
  BIND (xsd:double(?o) as ?doubleTest)
}
```

The result has no big surprises. It converted the ones it could and left the blank node, the `Montréal` URI, and the boolean `true` value alone.

```
----------------------------------------------------------------------------------------
| o            | integerTest | decimalTest      | floatTest          | doubleTest      |
========================================================================================
| 3            | 3           | "3"^^xsd:decimal | "3"^^xsd:float     | "3"^^xsd:double |
| _:b0         |             |                  |                    |                 |
| "5 bucks"    |             |                  |                    |                 |
| 4            | 4           | "4"^^xsd:decimal | "4"^^xsd:float     | "4"^^xsd:double |
| dbr:Montréal |             |                  |                    |                 |
| true         |             |                  |                    |                 |
| 1.0e5        |             |                  | "1.0e5"^^xsd:float | 1.0e5           |
----------------------------------------------------------------------------------------
```

In the output, ARQ represented most of the numbers as quoted strings with ^^ type designators after them and used shortcuts where possible: the `?integerTest` 3 and 4 values and the `?doubleTest` 1.0e5 value. Remember, though, that these are just shortcuts; `3` and `"3"^^xsd:decimal` represent the same thing, and so do `1.0e5` and `"1.0e5"^^xsd:double` (and for that matter, `"1.0e5"^^<http://www.w3.org/2001/XMLSchema#double>`).

To test the use of the other three casting functions, I created an augmented version of the ex241.ttl data file. I also removed the `Montréal` triple, because `xsd:string` does the same thing to it that `str()` did above, and `xsd:boolean` and `xsd:dateTime` can't do anything with it:

```
# filename: ex259.ttl

@prefix dm: <http://learningsparql.com/ns/demo#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:   <http://learningsparql.com/ns/data#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

d:id1 dm:location _:b1 .
d:id3 dm:amount 3 .
d:id4 dm:amount "4"^^xsd:integer .
d:id5 dm:amount 1.0e5 .
d:id6 rdfs:label "5 bucks" .
d:id7 dm:shipped true  .
d:id8 dm:shipped "true"  .
d:id9 dm:shipped "True"  .
d:id10 dm:shipDate "2011-11-12"  .
d:id11 dm:shipDate "2011-11-13T14:30:00"  .
d:id12 dm:shipDate "2011-11-14T14:30:00"^^xsd:dateTime  .
```

This next query is similar to the last one except that it's trying to convert the object of each triple to a string and a boolean value:

```
# filename: ex260.rq

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?o ?stringTest ?booleanTest
WHERE
{
  ?s ?p ?o .
  BIND (xsd:string(?o) as ?stringTest)
  BIND (xsd:boolean(?o) as ?booleanTest)
}
```

The results show that conversion to a string works for everything but the blank node, and that conversion to boolean is pickier:

```
-----------------------------------------------------------------------------------------
| o                                     | stringTest                          | booleanTest |
=========================================================================================
| 3                                     | "3"^^xsd:string                     |             |
| "true"                                | "true"^^xsd:string                  | true        |
| "2011-11-14T14:30:00"^^xsd:dateTime   | "2011-11-14T14:30:00"^^xsd:string   |             |
| _:b0                                  |                                     |             |
| "5 bucks"                             | "5 bucks"^^xsd:string               |             |
| "2011-11-12"                          | "2011-11-12"^^xsd:string            |             |
| 4                                     | "4"^^xsd:string                     |             |
| "True"                                | "True"^^xsd:string                  |             |
| true                                  | "true"^^xsd:string                  | true        |
| "2011-11-13T14:30:00"                 | "2011-11-13T14:30:00"^^xsd:string   |             |
| 1.0e5                                 | "1.0e5"^^xsd:string                 |             |
-----------------------------------------------------------------------------------------
```

Conversion to `xsd:boolean` worked for the lowercase string "true" with the `d:id8` triple, but not the one with the `id:9` triple, because of its uppercase "T". (If you need to convert such strings to boolean values, the `LCASE()` function described in "String Functions" on page 164 will be handy.)

Our last query demonstrating type conversion tries to convert triple objects to a date-time value:

```
# filename: ex262.rq

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?o ?dateTimeTest
WHERE
{
  ?s ?p ?o .
  BIND (xsd:dateTime(?o) as ?dateTimeTest)
}
```

When run with the ex259.ttl dataset, we get this result:

```
-----------------------------------------------------------------------------
| o                                     | dateTimeTest                      |
=============================================================================
| 3                                     |                                   |
| "true"                                |                                   |
| "2011-11-14T14:30:00"^^xsd:dateTime   | "2011-11-14T14:30:00"^^xsd:dateTime |
| _:b0                                  |                                   |
| "5 bucks"                             |                                   |
| "2011-11-12"                          |                                   |
| 4                                     |                                   |
| "True"                                |                                   |
| true                                  |                                   |
| "2011-11-13T14:30:00"                 | "2011-11-13T14:30:00"^^xsd:dateTime |
| 1.0e5                                 |                                   |
-----------------------------------------------------------------------------
```

The value in the `d:id12` triple shows up in the output because it was already an `xsd:dateTime` value. The only input value that got converted from something else to an `xsd:dateTime` was the string value that was formatted exactly as the function expected: the "2011-11-13T14:30:00" one with the `d:id11` triple. The `xsd:dateTime()` function could not convert the "2011-11-12" string to an `xsd:dateTime` datatype, but if you appended "T00:00:00" to it first, the conversion would work.

The **`STRDT()`** ("STRing DataType") function creates a typed literal from its two argument: a value specified as a simple literal and a URI specifying the type.

1.1 Alert

The `STRDT()` function is new for SPARQL 1.1.

To get a general idea of how it works, let's see what the following query does to the ex241.ttl data:

```
# filename: ex264.rq

PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?o ?decimalTest
WHERE
{
  ?s ?p ?o .
  BIND (STRDT(str(?o),xsd:decimal) as ?decimalTest)
}
```

The output shows that in this query the function usually adds the `xsd:decimal` datatype indicator to a string representation of the value, whether it makes sense for that string or not:

```
----------------------------------------------------------------------
| o             | decimalTest                                        |
======================================================================
| 3             | "3"^^xsd:decimal                                   |
| _:b0          |                                                    |
| "5 bucks"     | "5 bucks"^^xsd:decimal                             |
| 4             | "4"^^xsd:decimal                                   |
| dbr:Montréal  | "http://dbpedia.org/resource/Montréal"^^xsd:decimal |
| true          | "true"^^xsd:decimal                                |
| 1.0e5         | 1.0e5                                              |
----------------------------------------------------------------------
```

For example, it doesn't make much sense with strings like "true" and "5 bucks" but `STRDT()` adds it anyway. (ARQ does issue some "Datatype format exception" warning messages for those, for 1.0e5, and for the Montréal URI.)

The real value of `STRDT()` over the conversion functions described above is its flexibility. Imagine that an RDF interface to a relational database manager pulled the following data out of it:

```
# filename: ex266.ttl

@prefix im: <http://learningsparql.com/ns/importedData#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:item1 im:product "kerosene" ;
        im:amount "14" ;
        im:units "liters" .

d:item2 im:product "double-knit polyester" ;
        im:amount "10" ;
        im:units "squareMeters" .

d:item3 im:product "gold-plated chain" ;
        im:amount "30" ;
        im:units "centimeters" .
```

The numeric values are just strings, and the only connection between each numeric value and the associated unit name (for example, "10" and "squareMeters") is that they're objects of triples with a common subject. Let's say that I want to convert them to data values in the *http://learningsparql.com/ns/data* namespace with customized datatypes from the *http://learningsparql.com/ns/importedData* namespace. The following query converts them using `STRDT()` to assign the custom datatypes:

```
# filename: ex267.rq

PREFIX im: <http://learningsparql.com/ns/importedData#>
PREFIX u:  <http://learningsparql.com/ns/units#>

CONSTRUCT { ?s u:amount ?newAmount . }
WHERE
{
  ?s im:product ?prodName ;
     im:amount ?amount ;
     im:units ?units .

  BIND (STRDT(?amount,
              URI(CONCAT("http://learningsparql.com/ns/units#",?units)))

        AS ?newAmount)
}
```

The result shows the values with the custom datatypes assigned:

```
@prefix d:      <http://learningsparql.com/ns/data#> .
@prefix u:      <http://learningsparql.com/ns/units#> .
@prefix im:     <http://learningsparql.com/ns/importedData#> .

d:item2
      u:amount      "10"^^u:squareMeters .

d:item1
      u:amount      "14"^^u:liters .

d:item3
      u:amount      "30"^^u:centimeters .
```

The `CONCAT()` function concatenates the unit's value onto a string version of the URL for the unit's namespace, the `URI()` function converts the result of that to a URI, and the `STRDT()` function assigns that URI as a datatype for the amount values. Because a prefix of `u:` was declared for the units URI, ARQ outputs that prefix with the unit designators.

`STRDT()` and all the other functions in this section join the ones described in "Node Type and Datatype Checking Functions" on page 144 to provide a nice toolbox when you need to clean up data. For example, let's say you've pulled some triples from the Linked Data Cloud, or you've used some utility program to convert a spreadsheet, some XML, or relational database data to triples. If you want to rearrange those triples into a specific structure with the types and properties that your applications expect to see, these functions will give your FILTER statements and your SPARQL rules a lot more power. (See "Finding Bad Data" on page 117 for more on SPARQL rules.)

Checking, Adding, and Removing Spoken Language Tags

In "Making RDF More Readable with Language Tags and Labels" on page 31 of Chapter 2, we saw how a literal can have a tag assigned to it to identify what language it's in and even what country's dialect of the language it uses, such as Brazilian Portuguese or Swiss French. Using these tags, you can assign multiple labels and other descriptive information to a resource so that descriptions are available in a choice of languages. This is why a query of information about a resource in DBpedia often returns many values for the same property: because you have the answer in multiple languages. For example, Figure 5-1 shows a query for the `rdfs:label` value of the city where motorcycle manufacturer Ducati is located and the result.

As you can see there are thirteen results, with the name shown in English, German, Spanish, Finnish, and other languages.

What if you only want the label in one language? The **`lang()`** function returns the language tag attached to a literal, so we can use that in a FILTER statement to indicate that we only want values with a particular language tag—in this case, with the tag for the English language:

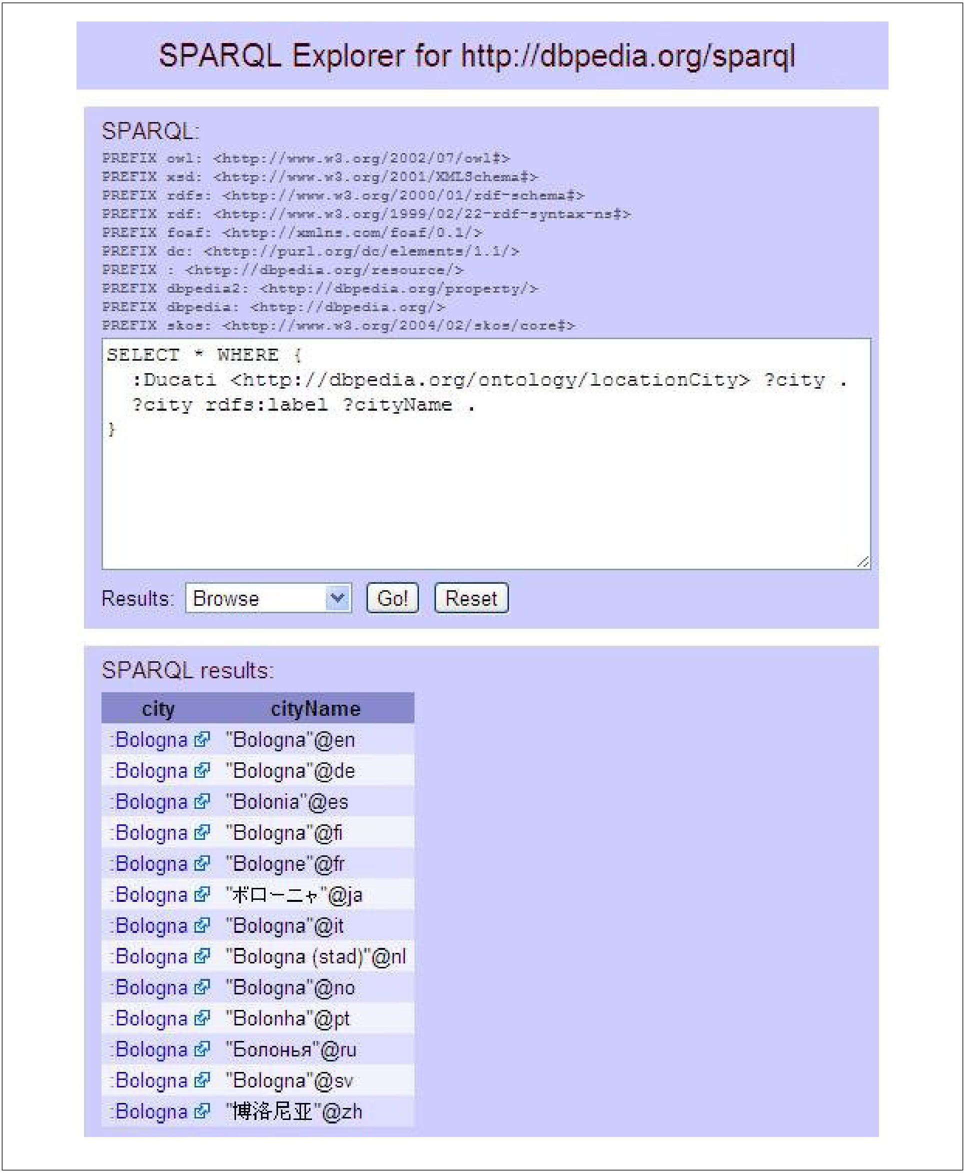
```
# filename: ex269.rq

 SELECT * WHERE {
  :Ducati <http://dbpedia.org/ontology/locationCity> ?city .
  ?city rdfs:label ?cityName .
  FILTER ( lang(?cityName) = "en" )
}
```

Let's look at another example. The ex039.ttl data example from Chapter 2 has four of the 13 `rdfs:label` values for the resource *http://dbpedia.org/resource/Switzerland*.

```
# filename: ex039.ttl

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

<http://dbpedia.org/resource/Switzerland> rdfs:label "Switzerland"@en,
  "Suiza"@es, "Sveitsi"@fi, "Suisse"@fr .
```

The following query retrieves all of them:

```
# filename: ex270.rq

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

SELECT ?label
WHERE { ?s rdfs:label ?label .}
```



```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT * WHERE {
  :Ducati <http://dbpedia.org/ontology/locationCity> ?city .
  ?city rdfs:label ?cityName .
}
```

city	cityName
:Bologna	"Bologna"@en
:Bologna	"Bologna"@de
:Bologna	"Bolonia"@es
:Bologna	"Bologna"@fi
:Bologna	"Bologne"@fr
:Bologna	"ボローニャ"@ja
:Bologna	"Bologna"@it
:Bologna	"Bologna (stad)"@nl
:Bologna	"Bologna"@no
:Bologna	"Bolonha"@pt
:Bologna	"Болонья"@ru
:Bologna	"Bologna"@sv
:Bologna	"博洛尼亚"@zh

Figure 5-1. Using SNORQL to query DBpedia for Ducati's location

Adding a FILTER statement that uses the `lang()` function to check for values with the language tag "en" tells the query engine that we only want those:

```
# filename: ex271.rq

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

SELECT ?label
WHERE {
  ?s rdfs:label ?label .
  FILTER ( lang(?label) = "en" )
}
```

Here's the result of running this query with the ex039.ttl file:

```
--------------------
| label            |
====================
| "Switzerland"@en |
--------------------
```

What if we don't want that "@en" showing up in our results? We learned in "Node Type Conversion Functions" on page 146 that the `str()` function returns a string representation of the argument passed to it. This includes the stripping of language tags, which makes it very helpful here:

```
# filename: ex273.rq

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

SELECT ?strippedLabel
WHERE {
  ?s rdfs:label ?label .
  FILTER ( lang(?label) = "en" )
  BIND (str(?label) AS ?strippedLabel)
}
```

Here's the result of running ex273.rq with the same ex039.ttl data:

```
-----------------
| strippedLabel |
=================
| "Switzerland" |
-----------------
```

1.1 Alert

When using the BIND keyword to store and retrieve the stripped version of the `rdfs:label` value, you would need a SPARQL 1.1 processor.

Here's another data file from the same chapter. This one includes country codes with the language codes to show which terms are American English and which are British English:

```
# filename: ex037.ttl

@prefix :     <http://www.learningsparql.com/ns/demo#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

:sideDish42 rdfs:label "french fries"@en-US .
:sideDish42 rdfs:label "chips"@en-GB .

:sideDish43 rdfs:label "chips"@en-US .
:sideDish43 rdfs:label "crisps"@en-GB .
```

When we run the same ex273.rq query to look for English language `rdfs:label` values in this file and then strip the language tags, we get nothing:

```
-----------------
| strippedLabel |
=================
-----------------
```

Why? Because the query is looking for `@en` tags and the data has `@en-US` and `@en-GB` tags. If the query's FILTER had looked for values where `lang()` returned "en-GB" or "en-US", we would have gotten those.

Fortunately, SPARQL's **`langMatches()`** function offers more flexibility. It compares the language tag in its first argument with the value in its second and returns a boolean `true` if the language matches. If the second argument doesn't mention a specific country variation of the language, the function doesn't care about it. (It also doesn't care about whether the country code in the function argument or the country codes in the data are in upper- or lowercase.) This version of the last query will ignore any country codes:

```
# filename: ex276.rq

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

SELECT ?strippedLabel
WHERE {
  ?s rdfs:label ?label .
  FILTER ( langMatches(lang(?label),"en" ))
  BIND (str(?label) AS ?strippedLabel)
}
```

When run against the ex037.ttl data, we get all the English language values:

```
------------------
| strippedLabel  |
==================
| "crisps"       |
| "chips"        |
| "chips"        |
| "french fries" |
------------------
```

Because of its flexibility, langMatches() is better for testing language values than lang(). For example, to test whether something's in Spanish, you're better off using the boolean expression langMatches(lang(?someVal),"es") than the expression (lang(?someVal) = "es") for the FILTER condition.

Let's say I'm doing some data cleanup and I want to make sure that all of my rdfs:label values have language tags, so I want to list any that don't. The langMatches() function is so flexible that it accepts a wildcard as its second argument, so you can use it to test which values have or are missing language tags. The following data file has three rdfs:label values, but only two have language tags:

```
# filename: ex278.ttl

@prefix d:    <http://learningsparql.com/ns/data#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .

d:item1 rdfs:label "dog" .
d:item2 rdfs:label "cat"@en .
d:item3 rdfs:label "turtle"@en-US .
```

The langMatches(lang(?label),"*") expression in the following query will return true for each ?label value that has a language tag and false for each that doesn't. The !() that wraps this expression flips the boolean value so that the complete filter expression returns true for each ?label that *doesn't* have a language tag:

```
# filename: ex279.rq

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema>

SELECT ?label
WHERE
{
   ?s rdfs:label ?label .
    FILTER (!(langMatches(lang(?label),"*")))
}
```

When we run this query with the ex278.ttl data, it lists the one value without a language tag:

```
---------
| label |
=========
| "dog" |
---------
```

So far in this section we've learned about using language codes as part of a query's search criteria and how to strip off the language code. What if we want to add a language tag to a string? We can't just concatenate the tag on, because it's a special piece of metadata, not an extra few characters of the string value. To do this, we use the

`STRLANG()` function, which takes a literal and a string representing a language tag as arguments and returns the literal tagged with that language code.

1.1 Alert

The `STRLANG()` function is new in SPARQL 1.1.

Imagine that some utility has converted a spreadsheet of equivalent American and British terms into the following triples:

```
# filename: ex281.ttl

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dm:   <http://learningsparql.com/ns/demo#> .
@prefix d:    <http://learningsparql.com/ns/data#> .

d:cell1 dm:row 1 ;
        dm:column 1 ;
        rdfs:label "truck" .

d:cell2 dm:row 1 ;
        dm:column 2 ;
        rdfs:label "lorry" .

d:cell3 dm:row 2 ;
        dm:column 1 ;
        rdfs:label "apartment" .

d:cell4 dm:row 2 ;
        dm:column 2 ;
        rdfs:label "flat" .

d:cell5 dm:row 3 ;
        dm:column 1 ;
        rdfs:label "elevator" .

d:cell6 dm:row 3 ;
        dm:column 2 ;
        rdfs:label "lift" .
```

Utilities that convert spreadsheet files to RDF are easy to find.

Each row of the spreadsheet has an American term in its first column and the corresponding British term in the second, and the following query converts this for use in a SKOS taxonomy. Because it's creating RDF triples, it's a CONSTRUCT query and not a SELECT query. For each row, it binds the `rdfs:label` value from column 1 to the `?USTerm` variable, and then the `STRLANG()` function tags that value as `@en-US` and puts the result in the `?taggedUSTerm` variable. A similar set of logic uses the value from the same row's second column to create a `?taggedGBTerm` value.

```
# filename: ex282.rq

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dm:   <http://learningsparql.com/ns/demo#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

CONSTRUCT
{  ?rowURI rdfs:type skos:Concept ;
           skos:prefLabel ?taggedUSTerm, ?taggedGBTerm . }
WHERE
{
   ?cell1 dm:row ?rownum ;
          dm:column 1 ;
          rdfs:label ?USTerm .

   BIND (STRLANG(?USTerm,"en-US") AS ?taggedUSTerm)

   ?cell2 dm:row ?rownum ;
          dm:column 2 ;
          rdfs:label ?GBTerm .

   BIND (STRLANG(?GBTerm,"en-GB") AS ?taggedGBTerm)

   BIND (URI(CONCAT("http://learningsparql.com/ns/terms#t",str(?rownum)))
         AS ?rowURI)
}
```

The query's last BIND statement uses the `URI()` function that we learned about earlier in this chapter to create a URI that serves as the subject for the three triples that it creates for each `?rownum` value: one saying that that the URI represents a SKOS concept and two more assigning American and British `skos:prefLabel` values to that URI. Here is the result of running the query with the ex281.ttl data:

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix dm:      <http://learningsparql.com/ns/demo#> .
@prefix skos:    <http://www.w3.org/2004/02/skos/core#> .

<http://learningsparql.com/ns/terms#t1>
      rdfs:type     skos:Concept ;
      skos:prefLabel  "truck"@en-US ;
      skos:prefLabel  "lorry"@en-GB .

<http://learningsparql.com/ns/terms#t2>
      rdfs:type     skos:Concept ;
      skos:prefLabel  "flat"@en-GB ;
      skos:prefLabel  "apartment"@en-US .

<http://learningsparql.com/ns/terms#t3>
      rdfs:type     skos:Concept ;
      skos:prefLabel  "elevator"@en-US ;
      skos:prefLabel  "lift"@en-GB .
```

Of course, you could also use this same `STRLANG()` function to assign language tags that do not include country designations.

String Functions

SPARQL provides some basic functions for looking at and manipulating strings of text. They're useful enough that we couldn't have gotten this far in the book without using several, so many will look familiar. If you do a lot of string manipulation, check the SPARQL implementation you're using to see if it offers any additional string functions as extensions.

1.1 Alert

Except for `regex()`, all the functions listed in this section are new in SPARQL 1.1, but many (or some version of them) were popular extensions to SPARQL 1.0 processors.

To try them out, we'll use this little data file:

```
# filename: ex284.ttl

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .
@prefix d:    <http://learningsparql.com/ns/data#> .

d:item1 rdfs:label "My String" .

d:item2 rdfs:label "123456" .
```

This first query demonstrates the use of the `STRLEN()`, `SUBSTR()`, `UCASE()`, and `LCASE()` functions:

```
# filename: ex285.rq

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema>

SELECT ?label ?strlenTest ?substrTest ?ucaseTest ?lcaseTest
WHERE
{
  ?s rdfs:label ?label .
  BIND (STRLEN(?label) AS ?strlenTest)
  BIND (SUBSTR(?label,4,2) AS ?substrTest)
  BIND (UCASE(?label) AS ?ucaseTest)
  BIND (LCASE(?label) AS ?lcaseTest)
}
```

When ARQ applies this query to the ex284.ttl data file, it gives us this result:

```
----------------------------------------------------------------------
| label       | strlenTest | substrTest | ucaseTest   | lcaseTest   |
======================================================================
| "123456"    | 6          | "45"       | "123456"    | "123456"    |
| "My String" | 9          | "St"       | "MY STRING" | "my string" |
----------------------------------------------------------------------
```

The first column of the result shows the input string, and the remaining columns show what each function did with the two input strings:

- The **`STRLEN()`** function returns the length of the string passed as an argument.
- The **`SUBSTR()`** function returns a substring of the string passed as its first argument. The second argument specifies the character to start at, and the optional third argument specifies how many characters to return. The function call in our example asks for 2 characters starting at the fourth character of the **`?label`** value, which is "45" for "123456" and "St" for "My String".
- The **`UCASE()`** function converts the input to uppercase, leaving any numeric digits alone.
- The **`LCASE()`** function is similar to `UCASE()`, but converts its input to lowercase.

The next query demonstrates four functions that each return a boolean value telling you whether the string meets a certain condition: `STRSTARTS()`, `STRENDS()`, `CONTAINS()`, and `regex()`.

```
# filename: ex287.rq

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema>

SELECT ?label ?startsTest ?endsTest ?containsTest ?regexTest
WHERE
{
  ?s rdfs:label ?label .
  BIND (STRSTARTS(?label,"12") AS ?startsTest)
  BIND (STRENDS(?label,"ing") AS ?endsTest)
  BIND (CONTAINS(?label," ") AS ?containsTest)
  BIND (regex(?label,"\\d{3}") AS ?regexTest)
}
```

Here's what this query does with the ex284.ttl data file:

```
-------------------------------------------------------------------
| label       | startsTest | endsTest | containsTest | regexTest |
===================================================================
| "123456"    | true       | false    | false        | true      |
| "My String" | false      | true     | true         | false     |
-------------------------------------------------------------------
```

Being boolean functions, they all return `true` or `false`, depending on what the function found in the string:

- The **`STRSTARTS()`** function checks whether the string in the first argument starts with the string in the second argument. It found that the string "123456" does begin with "12" and "My String" doesn't.
- The **`STRENDS()`** function checks whether the string in the first argument ends with the string in the second argument. It found that the string "123456" does not end with "ing", but "My String" does.

- The **`CONTAINS()`** function checks whether the string in the second argument can be found anywhere in the first argument. Looking for a single space, the **`CONTAINS()`** function call above found it in "My String" but not in "123456".
- The **`regex()`** function is a more flexible version of the `CONTAINS()` function, because you can specify a regular expression as its second argument. The regular expression in ex287.rq represents three numeric digits in a row, which the function found in "123456" but not in "My String". An optional third argument of "i" would tell this function to ignore case differences when searching for the string, but this would be irrelevant when searching for numeric digits.

The `regex()` function expects its first argument to be either an `xsd:string` or a simple literal with no language tag, so you may want to use the `str()` function to ensure that that's what you're passing—for example, `regex(str(?someVar),"jpg")`.

The language used to specify the regular expressions comes from the XML Schema Part 2 specification, and is roughly the same as the one used in the Perl programming language, the grep file searching utility, and their Unix-based cousins. Some of the more popular regular expressions special characters include the period, which is a wildcard that stands in for any character; `\d`, which represents any numeric digit; and `\s`, which represents any space character (a spacebar space, tab, carriage return or line feed).

Several more operators let you specify how many of these characters you're looking for. For example, when adding some of these operators after a period:

- `.*` represents zero or more characters.
- `.+` represents one or more characters.
- `.?` represents zero or one character.
- `.{4}` represents exactly four characters.

These can be mixed and matched; I used `\d{3}` in ex287.rq to look for 3 digits in a row. Although "123456" had more than that, as soon as the function found the "123" in the beginning of the string it had what it was looking for.

I actually used `\\d{3}` as the regular expression, because the backslash used as part of the regular expression language had to be escaped.

The characters shown above are by no means the complete range of special characters that you can use in a regular expression. They can be much fancier, but they can also be much simpler. For example, in "Searching for Strings" on page 12, we saw that the following query retrieves triples where the string "yahoo", in any combination of upper- and lowercase, is found in the triple's object.

```
# filename: ex021.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

SELECT *
WHERE
{
  ?s ?p ?o .
  FILTER (regex(?o, "yahoo","i"))
}
```

The regular expression here uses none of the regular expression special characters. Even without them, the ability to do a case-insensitive search through data specified by the rest of your query means that such a simple use of the `regex()` function can be very handy.

A SPARQL function that we've already seen is **`ENCODE_FOR_URI()`**, which is worth a closer look. This transforms any characters in a string that might cause problems if that string is used in the path part of a URI, usually by converting them to a percent sign followed by a number representing that character as a hexadecimal code point. Let's see what it does to this variation on the sample data file that we've been using:

```
# filename: ex289.ttl

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .
@prefix d:    <http://learningsparql.com/ns/data#> .

d:item1 rdfs:label "My String" .

d:item2 rdfs:label "http://www.learnsparql.com/cgi/func1&color=red" .
```

The following query returns the encoded version of each `?label` value:

```
# filename: ex290.rq

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema>

SELECT ?encodeTest
WHERE
{
  ?s rdfs:label ?label .
  BIND (ENCODE_FOR_URI(?label) AS ?encodeTest)
}
```

The function converted the space in "My String" to `%20`, and it converted all the punctuation characters in the URI using a similar encoding.

```
----------------------------------------------------------------------
| encodeTest                                                         |
======================================================================
| "http%3A%2F%2Fwww%2Elearnsparql%2Ecom%2Fcgi%2Ffunc1%26color%3Dred" |
| "My%20String"                                                      |
----------------------------------------------------------------------
```

This is especially useful, as we'll see inChapter 7, when you pass a URI or a SPARQL query as a parameter to a web service such as a SPARQL endpoint.

Numeric Functions

Before we get to numeric functions, don't forget that you can use all the typical arithmetic operators such as `+`, `-`, `*`, and `/` in your SPARQL expressions. We also saw in "Grouping Data and Finding Aggregate Values within Groups" on page 93 that the `AVG()`, `MIN()`, `MAX()`, `SUM()`, and `COUNT()` functions give you some nice options for working with numeric data in your triples.

THE SPARQL spec also specifies that implementations must support the `abs()`, `round()`, `ceil()`, and `floor()` functions. As with string functions, it's worth checking the implementation of SPARQL that you're using to see if it offers any additional numeric functions as extension functions.

1.1 Alert

All of SPARQL's numeric functions are new in SPARQL 1.1.

To try out these numeric functions, we'll use this sample data:

```
# filename: ex292.ttl

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .
@prefix d:    <http://learningsparql.com/ns/data#> .
@prefix dm:   <http://learningsparql.com/ns/demo#> .

d:item1 dm:amount 4 .
d:item2 dm:amount 3.2 .
d:item3 dm:amount 3.8 .
d:item4 dm:amount -4.2 .
d:item5 dm:amount -4.8 .
```

This next query uses each of the four functions listed above:

```
# filename: ex293.rq

PREFIX dm:  <http://learningsparql.com/ns/demo#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?amount ?absTest ?roundTest ?ceilTest ?floorTest
WHERE
{
  ?s dm:amount ?amount .
  BIND (abs(?amount) AS ?absTest )
  BIND (round(?amount) AS ?roundTest )
  BIND (ceil(?amount) AS ?ceilTest )
  BIND (floor(?amount) AS ?floorTest )
}
```

Running the query with the ex292.ttl data, we get this result:

```
--------------------------------------------------------------------------------
| amount | absTest | roundTest          | ceilTest           | floorTest          |
================================================================================
| -4.8   | 4.8     | "-5"^^xsd:decimal | "-4"^^xsd:decimal | "-5"^^xsd:decimal |
| -4.2   | 4.2     | "-4"^^xsd:decimal | "-4"^^xsd:decimal | "-5"^^xsd:decimal |
| 3.8    | 3.8     | "4"^^xsd:decimal  | "4"^^xsd:decimal  | "3"^^xsd:decimal  |
| 3.2    | 3.2     | "3"^^xsd:decimal  | "4"^^xsd:decimal  | "3"^^xsd:decimal  |
| 4      | 4       | 4                  | 4                  | 4                  |
--------------------------------------------------------------------------------
```

- The **`abs()`** function returns the absolute value of the passed parameter, converting the input's -4.8 and -4.2 values to positive numbers.
- The **`round()`** function rounds off values to the nearest whole number.
- The **`ceil()`** function returns the "ceiling" value: the next whole number up from the argument if it has a fractional part, or the number itself it it's a whole number.
- The **`floor()`** function returns the next whole number below the argument if it has a fractional part or the number itself if it's a whole number.

The **`rand()`** function returns a double-precision number between 0 and 1. It might return 0, but it won't return 1. If you want it to return something else, you can use other numeric functions and operators to produce the values you want. For example, if you multiply the value of `rand()` by 11, take the `floor()` value of that, and add 20, you'll get a whole number between 20 and 30, inclusive.

To demonstrate this, the following query outputs two numbers for every triple passed to it as input. The `?randTest1` variable will have the value of a simple call to the `rand()` function. The `?randTest2` value will have a random whole number between 20 and 30:

```
# filename: ex295.rq

SELECT ?randTest1 ?randTest2
WHERE
{
  ?s ?p ?o .
  BIND (rand() AS ?randTest1)
  BIND (floor(rand()*11)+20 AS ?randTest2)
}
```

(Note that the query doesn't actually use any of the data from the input.) When run with the ex292.ttl data file above, which has five triples, we get these five pairs of random numbers:

```
-------------------------------------
| randTest1             | randTest2 |
=====================================
| 0.20209451122917443e0 | 29.0e0    |
| 0.04707018085243442e0 | 28.0e0    |
| 0.2190604769364065e0  | 25.0e0    |
| 0.5742086203122172e0  | 22.0e0    |
| 0.3674021731250735e0  | 21.0e0    |
-------------------------------------
```

Running it again right away without changing anything, we get a different set of numbers:

```
-------------------------------------
| randTest1            | randTest2 |
=====================================
| 0.8665625585923823e0  | 24.0e0    |
| 0.21184532852211524e0 | 22.0e0    |
| 0.18848604673741176e0 | 25.0e0    |
| 0.9411502523245124e0  | 27.0e0    |
| 0.5816330932580108e0  | 25.0e0    |
-------------------------------------
```

When used with a CONSTRUCT query, the `rand()` function can be valuable for generating sample data.

Date and Time Functions

1.1 Alert

The date and time functions are all new for SPARQL 1.1.

SPARQL gives you eight functions for manipulating date and time data. You can use these with literals typed as `xsd:dateTime` data and, depending on the purpose of the function, on literals that use the `xsd:date` and `xsd:time` types that are based on the `xsd:dateTime` datatype. SPARQL also offers the `now()` function, which tells you the date and time that your query started running.

We saw that the SPARQL datatypes are based on the XML Schema Part 2 datatypes. The Schema Part 2 date and time datatypes are based on the ISO 8601 standard. Using ISO 8601, a full date and time string to represent October 14th, 2011, at 12 noon five timezones west of Greenwich, England (for example, in New York City) would be “2011-10-14T12:00:00.000-05:00”. You could represent the date itself as `"2011-10-14"^^xsd:date` or the time as `"12:00:00.000-05:00"^^xsd:time`. The parts showing the timezone and fractions of a second are not required, so that if you wanted to say that a meeting begins or a flight leaves at `"2011-10-14T12:00:00"^^xsd:dateTime`, you would not get an error.

Except for the `now()` function, all of SPARQL’s date and time functions are designed to pull specific bits out of these date and time values. Let’s see what they do with this sample data, which uses the `starts` property from the Tickets ontology that was designed to work with the GoodRelations ecommerce ontology:

```
# filename: ex298.ttl

@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix d:   <http://learningsparql.com/ns/data#> .
@prefix t:   <http://purl.org/tio/ns#> .
```

```
d:meeting1 t:starts "2011-10-14T12:30:00.000-05:00"^^xsd:dateTime .

d:meeting2 t:starts "2011-10-15T12:30:00"^^xsd:dateTime .
```

The following query pulls out some of the pieces of the meeting dates and times:

```
# filename: ex299.rq

PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX t: <http://purl.org/tio/ns#>

SELECT ?mtg ?yearTest ?monthTest ?dayTest ?hoursTest ?minutesTest
WHERE
{
  ?mtg t:starts ?startTime .
  BIND (year(?startTime) AS ?yearTest)
  BIND (month(?startTime) AS ?monthTest)
  BIND (day(?startTime) AS ?dayTest)
  BIND (hours(?startTime) AS ?hoursTest)
  BIND (minutes(?startTime) AS ?minutesTest)
}
```

The results are mostly predictable except for the two different `?hoursTest` values:

```
-------------------------------------------------------------------------
| mtg        | yearTest | monthTest | dayTest | hoursTest | minutesTest |
=========================================================================
| d:meeting2 | 2011     | 10        | 15      | 12        | 30          |
| d:meeting1 | 2011     | 10        | 14      | 17        | 30          |
-------------------------------------------------------------------------
```

The processor assumes Greenwich Mean Time as the default timezone, so because `meeting1` takes place at 12:30 five timezones west of Greenwich, that's 17:30 in England.

The `seconds()` function returns the seconds portion of a date-time value as a decimal number. We'll see an example of its use shortly. First, lets look at the two functions for checking the timezone of a date-time value, `timezone()` and `tz()`:

```
# filename: ex301.rq

PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX t: <http://purl.org/tio/ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?timezoneTest ?tzTest
WHERE
{
  ?mtg t:starts ?startTime .
  BIND (timezone(?startTime) AS ?timezoneTest)
  BIND (tz(?startTime) AS ?tzTest)
}
```

The **`timezone()`** function returns the timezone part of a date typed as an `xsd:dayTimeDuration`, and **`tz()`** returns a simple literal version of it. Here is how the ex301.rq query looks when run with the ex298.ttl data:

```
--------------------------------------------------------
| mtg         | timezoneTest                   | tzTest   |
========================================================
| d:meeting2 |                                 | ""       |
| d:meeting1 | "-PT5H"^^xsd:dayTimeDuration | "-05:00" |
--------------------------------------------------------
```

It pulled the timezone values out of the `d:meeting1` time, and from the `d:meeting2` time it got nothing as the `timezone()` value and an empty string as the `tz()` value.

The **`now()`** function returns the current date and time—more specifically, the date and time when the query starts running. The following query shows us the date the query was run and, to demonstrate the `seconds()` function, that portion of the current time. This query ignores the input, so you can run it with any input data file you want:

```
# filename: ex303.rq

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?currentTime ?currentSeconds
WHERE
{
   BIND (now() AS ?currentTime)
   BIND (seconds(?currentTime) AS ?currentSeconds)
}
```

The query could have used a call to the `now()` function as the argument to the `seconds()` function, but used the variable created from its value instead. Either way, the `seconds()` function expects an `xsd:dateTime` value as its argument with all the right ISO 8601 pieces in the right places. Here is one result of running this function:

```
------------------------------------------------------------------
| currentTime                                      | currentSeconds |
==================================================================
| "2011-02-05T12:58:27.93-05:00"^^xsd:dateTime | 27.93          |
------------------------------------------------------------------
```

Hash Functions

1.1 Alert

Hash functions are new for SPARQL 1.1.

SPARQL's cryptographic hash functions convert a string of text to a hexadecimal representation of a bit string that can serve as a coded signature for the input string. For example, if you emailed me a paragraph of text and then in a separate email sent me the result of passing that text through a particular hash function, I could send that paragraph through the same hash function myself to see if I got the same result. If the result was different, I'd know that what I received from you was not what you sent.

This is popular in FOAF data, where an email address is a common identifier for a person but fear of spam prevents people from making their email addresses public in a FOAF file. The FOAF vocabulary includes the `foaf:mbox_sha1sum` property, which stores a hash string of an email address ("mailbox") generated with the SHA-1 cryptographic function. This way, you get a reasonably unique value to represent yourself without putting your email address where web crawlers can harvest it for spam mailing lists. A SHA-1 value represents a 160-bit signature string, and the slightly older MD5 algorithm uses a 128-bit string. (The more bits, the more security.) The other cryptographic hash functions supported by SPARQL, which are variations on the more recent SHA-2 algorithm, use a bit string size indicated by the numbers in their names.

SPARQL supports these hash functions:

- **`MD5()`**
- **`SHA1()`**
- **`SHA224()`**
- **`SHA256()`**
- **`SHA384()`**
- **`SHA512()`**

To demonstrate one, we'll take the following data from Chapter 1 and convert it to FOAF with a CONSTRUCT query. To give these people more privacy, we won't copy the phone numbers, and we'll substitute `foaf:mbox_sha1sum` values for their email addresses.

```
# filename: ex012.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName "Richard" .
d:i0432 ab:lastName  "Mutt" .
d:i0432 ab:homeTel   "(229) 276-5135" .
d:i0432 ab:email     "richard49@hotmail.com" .

d:i9771 ab:firstName "Cindy" .
d:i9771 ab:lastName  "Marshall" .
d:i9771 ab:homeTel   "(245) 646-5488" .
d:i9771 ab:email     "cindym@gmail.com" .

d:i8301 ab:firstName "Craig" .
d:i8301 ab:lastName  "Ellis" .
d:i8301 ab:email     "craigellis@yahoo.com" .
d:i8301 ab:email     "c.ellis@usairwaysgroup.com" .
```

The query stores the result of the `SHA1()` function call in the `?hashEmail` variable and uses that as the `foaf:mbox_sha1sum` value:

```
# filename: ex305.rq
```

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ab: <http://learningsparql.com/ns/addressbook#>

CONSTRUCT {
  ?s foaf:givenName ?first ;
     foaf:familyName ?last ;
     foaf:mbox_sha1sum ?hashEmail .
}
WHERE
{
  ?s ab:firstName ?first ;
     ab:lastName ?last ;
     ab:email ?email .
  BIND (SHA1(?email) AS ?hashEmail )
}
```

Here is what this query does with the ex012.ttl data:

```
@prefix d:      <http://learningsparql.com/ns/data#> .
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .
@prefix ab:     <http://learningsparql.com/ns/addressbook#> .

d:i9771
      foaf:familyName  "Marshall" ;
      foaf:givenName  "Cindy" ;
      foaf:mbox_sha1sum  "821be6ab56326d7b08246f2cb9c0f68afe0156d9" .

d:i0432
      foaf:familyName  "Mutt" ;
      foaf:givenName  "Richard" ;
      foaf:mbox_sha1sum  "b7f191315aa6bb4a9ab56b02e334647c4b1104a0" .

d:i8301
      foaf:familyName  "Ellis" ;
      foaf:givenName  "Craig" ;
      foaf:mbox_sha1sum  "396d3e3aef87e0fecd3cbbdd2479eb3797d7af18" ;
      foaf:mbox_sha1sum  "faec1b07cf7c10e302544f22958c02c53844a4fa" .
```

Let's say someone else created these triples and posted them publicly. You have an email address for Richard Mutt and you're wondering if it's the same one listed in this file. You could pass that email address to the `SHA1()` function using a SPARQL query. You could also pass it to a short program written in just about any programming language, because SHA1 support is easy to find. For example, this little Python program will make the conversion:

```
# filename: ex307.py

import hashlib
m = hashlib.sha1()
m.update("richard49@hotmail.com")
print m.hexdigest()
```

If the result is "b7f191315aa6bb4a9ab56b02e334647c4b1104a0", you know you've got the email address for the same Richard Mutt.

Extension Functions

Most SPARQL processor providers include functions above and beyond those required by the SPARQL specification. They do this to differentiate their program from others, to aid their own SPARQL-based application development (that is, they need a function that SPARQL doesn't provide, so they add it to their SPARQL implementation), and to send hints to the SPARQL Working Group about what they consider to be useful functions that are missing from SPARQL. The SPARQL 1.1 spec shows that the Working Group took a lot of the hints; just about all of the functions listed in this chapter as being new in SPARQL 1.1 were extension functions in more than one SPARQL processor before 1.1 was released.

When you develop with a particular SPARQL processor, get familiar with its extension functions, because they can expand the power of your queries, reduce your development time, and perhaps even reduce your execution time.

Some extensions provide more processing efficiency because they are more tightly coupled to that particular implementation. For example, Virtuoso's `bif:contains()` function is a variation on the `CONTAINS()` function that we saw earlier in this chapter, and can be much faster—if you're using Virtuoso.

Remember that using extensions to a standard takes your application outside of the standard, making your applications less portable.

Because extension functions come from outside the SPARQL standard, using them means that you must identify the namespace where they come from. For example, ARQ extension functions are in the *http://jena.hpl.hp.com/ARQ/function#* namespace, so if you use its `afn:localname` function, your query must declare the `afn:` prefix, just as it might be declaring the `rdfs:` or `dc:` prefix.

The following query uses ARQ's `afn:localname` and `afn:namespace` extension functions to split the local and namespace names out from the URI of every triple's subject.

```
# filename: ex308.rq

PREFIX afn: <http://jena.hpl.hp.com/ARQ/function#>
PREFIX d:   <http://learningsparql.com/ns/data#>

SELECT DISTINCT ?s ?sLocalname ?sNamespace
WHERE
{
  ?s ?p ?o .
  BIND (afn:localname(?s) AS ?sLocalname )
  BIND (afn:namespace(?s) AS ?sNamespace )
}
```

When run with the ex012.ttl data file that we saw above, we get this result:

```
----------------------------------------------------------------
| s       | sLocalname | sNamespace                            |
================================================================
| d:i9771 | "i9771"    | "http://learningsparql.com/ns/data#" |
| d:i0432 | "i0432"    | "http://learningsparql.com/ns/data#" |
| d:i8301 | "i8301"    | "http://learningsparql.com/ns/data#" |
----------------------------------------------------------------
```

If there are some new functions you'd like to see in the SPARQL processor you're using, let the developers know. Perhaps others have asked for the same new functions and your vote will tip the balance. And maybe this will help make those functions popular enough to be included in SPARQL 1.2 or 2.0!

Summary

In this chapter, we learned about:

- What datatypes SPARQL supports natively and how queries can use both these and custom datatypes.
- Options for representing strings.
- How to do arithmetic in SPARQL.
- SPARQL 1.1 functions for program logic and node type and datatype checking and conversion.
- SPARQL 1.1 functions for controlling spoken language tags.
- String, numeric, date and time, and hash functions in SPARQL 1.1.
- The role that extension functions can play in your queries.

CHAPTER 6
Updating Data with SPARQL

It's great when you can pull data out of a collection and rearrange and cross-reference it any way you want, but when you can use a standard query language to add data to that collection, and to delete and update its data, you have what you need to build serious full-fledged applications around RDF's flexible data model. The SPARQL 1.1 Update specification describes the syntax and options for doing this, and although it's a more recent addition to SPARQL, several implementations already support it. This chapter uses Fuseki, one of the simplest SPARQL Update implementations, to demonstrate the various examples.

Most implementations you find of the SPARQL Update language will be features of triplestores, letting you act on the data in the triplestore. Being essentially a database management program (if not a relational one), a triplestore should let you query the data with the SPARQL query language and manage it with the update language.

1.1 Alert

SPARQL Update is new for SPARQL 1.1. SPARQL 1.0 defined no way to update data, so triplestores with SPARQL support originally relied on proprietary extensions to allow updating of their data. You may still see this with older triplestores.

In this chapter, we'll learn about:

- "Getting Started with Fuseki" on page 178: Just enough about Fuseki to try out all the key parts of SPARQL Update.
- "Adding Data to a Dataset" on page 180: How to add triples specified in your query and triples from remote sources to the default graph.
- "Deleting Data" on page 186: How to delete data from the default graph.
- "Changing Existing Data" on page 188: How to replace existing triples in the dataset's default graph.
- "Named Graphs" on page 193: How to create named graphs, add triples to them, delete triples from them, and replace both triples and entire graphs.

Getting Started with Fuseki

Fuseki (*http://www.openjena.org/wiki/Fuseki*) is part of the Jena project. It describes itself as a "SPARQL Server" and functions as a web server triplestore that accepts SPARQL queries that you enter on a web form as well as SPARQL queries that you send it as HTTP requests. In this chapter, we'll use the web form.

As of this writing, the latest official release of Fuseki is release 0.2. The range of Fuseki's features and accompanying documentation is already impressive, but if you use a later version you may see some differences from what is described here.

Once you download and unzip the Fuseki distribution file, you're ready to go. It includes a shell script called fuseki-server that will start it up under Linux or on a Mac. On any platform, including Windows, you can start it by calling the included jar file directly. For example, you can find out about Fuseki's command line options with the following command:

```
java -jar fuseki-sys.jar --help
```

The fuseki-sys.jar file may have a different name in future releases.

Before starting up the Fuseki server, I created a subdirectory of the directory where this jar file was stored and called it `dataDir`. Then, to start up Fuseki as a server on a Windows machine, I used this command, which is based on a line I saw in the fuseki-server shell script:

```
java -Xmx1200M -jar fuseki-sys.jar --update --loc=dataDir /myDataset
```

This command line includes the following parameters:

- `--update` tells Fuseki to allow updates to stored data. Without this, it defaults to read-only mode.
- `--loc=dataDir` tells it to store data in a TDB database and store it in the `dataDir` directory that I just created. (TDB is another part of the Jena project designed to store RDF.) This will be a persistent database, keeping your data on your hard disk even after you shut down Fuseki.
- `/myDataset` is the dataset path name, and must begin with a slash. (This is a Fuseki detail, and unrelated to the SPARQL spec.)

The `--help` switch describes many other options for configuring Fuseki's behavior, but the ones in the command line above are fine for experimenting with SPARQL's update language.

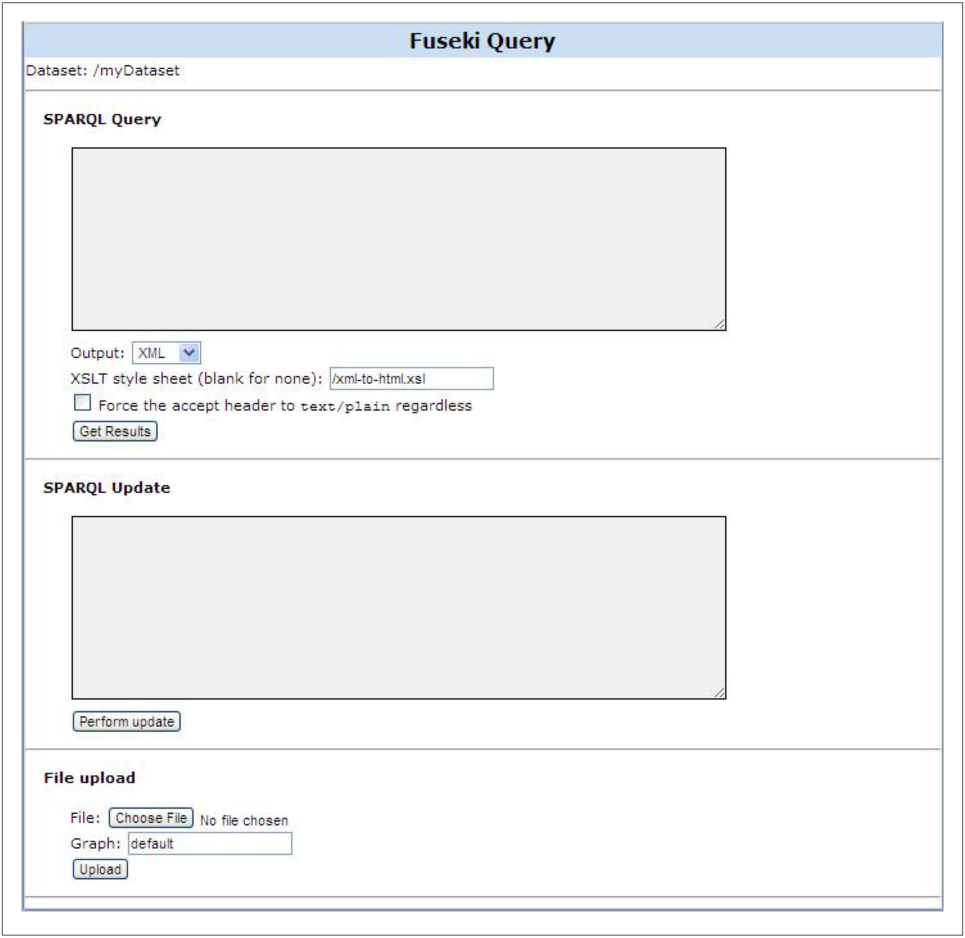
As the server starts up, a few status messages will scroll up in the window where you entered the command to start it. (Later, when you've finished using Fuseki and you're ready to shut it down, press `Ctrl+C` in this same window.)

When the startup messages stop appearing, Fuseki is ready to use. Send your browser to *http://localhost:3030/* to see Fuseki's main screen.

If you'd prefer Fuseki to use a different port besides 3030, `--help` shows you how.

Click the main screen's Control Panel link. On the Fuseki Control Panel screen that this leads to, you need to pick a dataset; the `/myDataset` one created when you started Fuseki will be the only choice, so click the Select button.

This brings you to the Fuseki Query screen, as shown in Figure 6-1. This is where we'll do our experiments for the rest of this chapter.


Fuseki Query

Dataset: /myDataset

SPARQL Query

Output: XML

XSLT style sheet (blank for none): /xml-to-html.xsl

☐ Force the accept header to `text/plain` regardless

Get Results

SPARQL Update

Perform update

File upload

File: Choose File No file chosen

Graph: default

Upload

Figure 6-1. Fuskeki's Query form screen

Adding Data to a Dataset

Most triplestores with a form-based interface offer a way to load data by filling out a form. To provide some baseline data for our first few experiments with SPARQL Update requests, use the File upload section at the bottom of the Fuseki Query form to load ex012.ttl, a sample data file that will be familiar from this book's early chapters:

```
# filename: ex012.ttl

@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix d:  <http://learningsparql.com/ns/data#> .

d:i0432 ab:firstName "Richard" .
d:i0432 ab:lastName  "Mutt" .
```

```
d:i0432 ab:homeTel   "(229) 276-5135" .
d:i0432 ab:email     "richard49@hotmail.com" .

d:i9771 ab:firstName "Cindy" .
d:i9771 ab:lastName  "Marshall" .
d:i9771 ab:homeTel   "(245) 646-5488" .
d:i9771 ab:email     "cindym@gmail.com" .

d:i8301 ab:firstName "Craig" .
d:i8301 ab:lastName  "Ellis" .
d:i8301 ab:email     "craigellis@yahoo.com" .
d:i8301 ab:email     "c.ellis@usairwaysgroup.com" .
```

After clicking the form's Choose File button, select that file, leave the Graph setting at "default", and click the form's Upload button. Fuseki will read in the data and display a short message about how many triples it read:

```
Triples = 12
```

Clicking your browser's Back button will return you to the Fuseki Query form.

To check on what data is now in the dataset, enter the following simple query in the SPARQL Query section at the top of the form:

```
# filename: ex311.rq

SELECT *
WHERE
{ ?s ?p ?o }
```

When you click the Get Results button, Fuseki will display the 12 triples:

s	p	o
<http://learningsparql.com/ns/data#i9771>	<http://learningsparql.com/ns/addressbook#email>	"cindym@gmail.com"
<http://learningsparql.com/ns/data#i9771>	<http://learningsparql.com/ns/addressbook#homeTel>	"(245) 646-5488"
<http://learningsparql.com/ns/data#i9771>	<http://learningsparql.com/ns/addressbook#lastName>	"Marshall"
<http://learningsparql.com/ns/data#i9771>	<http://learningsparql.com/ns/addressbook#firstName>	"Cindy"
<http://learningsparql.com/ns/data#i0432>	<http://learningsparql.com/ns/addressbook#email>	"richard49@hotmail.com"
<http://learningsparql.com/ns/data#i0432>	<http://learningsparql.com/ns/addressbook#homeTel>	"(229) 276-5135"
<http://learningsparql.com/ns/data#i0432>	<http://learningsparql.com/ns/addressbook#lastName>	"Mutt"
<http://learningsparql.com/ns/data#i0432>	<http://learningsparql.com/ns/addressbook#firstName>	"Richard"

s	p	o
<http://learningsparql.com/ns/data#i8301>	<http://learningsparql.com/ns/addressbook#email>	"c.ellis@usairwaysgroup.com"
<http://learningsparql.com/ns/data#i8301>	<http://learningsparql.com/ns/addressbook#email>	"craigellis@yahoo.com"
<http://learningsparql.com/ns/data#i8301>	<http://learningsparql.com/ns/addressbook#lastName>	"Ellis"
<http://learningsparql.com/ns/data#i8301>	<http://learningsparql.com/ns/addressbook#firstName>	"Craig"

Fuseki returns data in the standard SPARQL Query Results XML format, and an XSLT stylesheet included with Fuseki formats it for display in your browser. I converted the XML for each set of query results into a table like the one shown here for easier rendering in this book.

As we learn about inserting data into and deleting it from your dataset, we'll use query ex311.rq often to check on the results of various update queries, so I'll refer to it as the "List All Default Graph Triples" query. (I originally called it the "List All Triples" query, but in "Named Graphs" on page 193 we'll use a different query that really lists *all* triples, whether they're in named graphs or not.)

When viewing these query results in Fuseki, note how the query has become part of the URL in your browser's Navigation toolbar. Bookmarking these results means that every time you go to that bookmark, you'll run this query, so it's a handy way to run your favorite queries more easily.

Let's do some updates on this data. We'll start by adding two triples: one that names an `ab:homeTel` value for resource `d:i8301` and another saying that `ab:Person` is a class.

Enter the following update request on the SPARQL Update panel of the Fuseki Query form and click the Perform update button under it:

```
# filename: ex312.ru

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ab:   <http://learningsparql.com/ns/addressbook#>
PREFIX d:    <http://learningsparql.com/ns/data#>

INSERT DATA
{
  d:i8301 ab:homeTel "(718) 440-9821" .
  ab:Person a rdfs:Class .
}
```

The SPARQL Update specification recommends that files storing SPARQL update requests have an extension of .ru, in lowercase.

After Fuseki displays a screen telling you that the update succeeded, click your browser's **Back** button to return to the Fuseki Query form. The SELECT query that you entered in the SPARQL Query panel at the top of the form will still be there, so click Get Results underneath it (or, if you bookmarked the query results, go to that bookmark) to see the data that Fuseki is now storing for you: the original 12 triples from before and the two new ones inserted by ex313.ru.

In the stored data, you'll see that the triple about `ab:Person` being an `rdfs:Class` has a predicate of `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`, even though in the update request that inserted this triple the predicate is "`a`", because "`a`" is just shorthand for this URI.

Before looking too closely at our first update request's syntax, let's look at another one that does exactly the same thing:

```
# filename: ex313.ru

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ab:   <http://learningsparql.com/ns/addressbook#>
PREFIX d:    <http://learningsparql.com/ns/data#>

INSERT
{
  d:i8301 ab:homeTel "(718) 440-9821" .
  ab:Person a rdfs:Class .
}
WHERE {}
```

It doesn't have the DATA keyword after INSERT like ex312.ru does, but it does have WHERE and a pair of curly braces at the end. In fact, it looks a lot like a CONSTRUCT query, and it is similar: it's creating new triples. This particular one doesn't have any conditions specified with triple patterns between the WHERE clause's curly braces, but as we'll see, an INSERT update request can include triple patterns there, and the INSERT clause's triple patterns can refer to those variables. An INSERT DATA statement cannot have a WHERE clause, and you can only put triples between the curly braces after the INSERT DATA operation. You can't put triple patterns there.

Triple patterns are just triples where you're allowed to substitute variables in any of the three positions.

Why does SPARQL give you two different ways to insert triples? As with a CONSTRUCT query, the use of triple patterns between the curly braces in an INSERT update request's WHERE clause gives you the flexibility to be creative when you specify the data to insert—for example, you can make up new triples based on patterns found with the WHERE clause. (Later in this chapter we'll see plenty of examples.) The INSERT DATA operation, by not allowing a WHERE clause or the use of variables, makes things simpler for the SPARQL processor, which can therefore process the data faster.

The difference in the data loading speed between an INSERT update request and an INSERT DATA update request is negligible when you compare the two examples above, but it's good to remember that you have this option when you need to load a large amount of data.

If patterns play no role in the data to insert, it's a best practice to use INSERT DATA instead of INSERT… WHERE {}. The ex313.ru update request above is just here for demonstration purposes.

Before we look at this flexibility in action, let's review a typical CONSTRUCT query. When the following query find any resource that has both `ab:firstName` and `ab:lastName` values, it stores the resource's URI in the variable `?person` and creates a new triple saying that this resource is a member of our new class `ab:Person`:

```
# filename: ex314.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

CONSTRUCT
{ ?person a ab:Person . }
WHERE
{
  ?person ab:firstName ?firstName ;
          ab:lastName  ?lastName .
}
```

If you used the ARQ command line query engine to run this query against the ex012.ttl data, you would see the following three triples in the result:

```
@prefix d:       <http://learningsparql.com/ns/data#> .
@prefix ab:      <http://learningsparql.com/ns/addressbook#> .

d:i9771
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
              ab:Person .

d:i0432
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
              ab:Person .

d:i8301
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
              ab:Person .
```

You can save this output in a file and use it for further work, but a CONSTRUCT query has no effect on the original input.

Our next example is the same as the ex314.rq CONSTRUCT one, except that the CONSTRUCT keyword has been changed to INSERT:

```
# filename: ex316.ru

PREFIX ab: <http://learningsparql.com/ns/addressbook#>

INSERT
{ ?person a ab:Person . }
WHERE
{
  ?person ab:firstName ?firstName ;
          ab:lastName  ?lastName .
}
```

The SPARQL specification uses the term "update request" and not "query" for these, so while ex314.rq is a query, ex316.ru is an update request.

Paste this into the SPARQL Update panel of the Fuseki Query and click Perform update to run it. If you then run the ex311.rq List All Default Graph Triples query, you'll see that the same three triples that were created by the ex314.rq query above have been added to the dataset by ex316.ru.

When we started up Fuseki, we told it to store this data in a persistent database, so if you shut down Fuseki and start it up again after running the ex316.ru update request, you'll find that Fuseki is still storing the triples inserted by ex312.ru and ex316.ru.

At the beginning of this chapter, we used Fuseki's Upload button to load an entire file at once. Another way to load data is the SPARQL Update LOAD operation, which lets you load an entire web-accessible dataset at once. For example, the following will load RDF data about early bebop bass player Tommy Potter from the Freebase community database:

```
# filename: ex317.ru

LOAD <http://rdf.freebase.com/rdf/en.tommy_potter>
```

After you paste this update request into Fuseki's SPARQL Update panel and click the Perform update button, the List All Default Graph Triples query will show that you loaded over 130 triples into your database.

Deleting Data

SPARQL Update's DELETE DATA and DELETE operations correspond to the INSERT DATA and INSERT operations. With the first, you list specific triples to delete; with the second you can do that and also use triple patterns for more flexibility.

Before looking at some examples, let's review some of the data inserted with the LOAD operation in the previous example by entering this query on the SPARQL Query section of the Fuseki Query form:

```
# filename: ex318.rq

SELECT * WHERE
{<http://rdf.freebase.com/ns/en.tommy_potter> ?p ?o }
```

When you run this query, you'll see the predicates and objects from the 47 or so triples that have that URI as a subject. This includes the predicates and objects from the following two triples, which show:

- When Potter was born.
- The fact that this URI represents the same resource as a particular DBpedia URI.

(Remember, the ex318.rq query result won't show these complete triples, because the query only asks for the predicates and objects.)

```
<http://rdf.freebase.com/ns/en.tommy_potter>
    <http://rdf.freebase.com/ns/people.person.date_of_birth>
    "1918-09-21" .

<http://rdf.freebase.com/ns/en.tommy_potter>
    owl:sameAs
    <http://dbpedia.org/resource/Tommy_Potter> .
```

In public data sources, these `owl:sameAs` triples provide an excellent hook for linking up data about a particular topic from multiple sources.

The following DELETE DATA update request removes these two triples from the dataset:

```
# filename: ex320.ru

PREFIX fb: <http://rdf.freebase.com/ns/>
PREFIX db: <http://dbpedia.org/resource/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

DELETE DATA
{
  fb:en.tommy_potter  fb:people.person.date_of_birth  "1918-09-21" ;
                      owl:sameAs db:Tommy_Potter .
}
```

After executing it, try the ex318.rq SELECT query above, and you'll see that those two triples are gone. (You will see another **owl:sameAs** triple linking Freebase's URI for Potter to one at the BBC—another nice example of how links in the Linked Data cloud can help you automate the gathering of information on a topic.)

The following DELETE update request would do the same thing as the DELETE DATA update request in ex320.ru:

```
# filename: ex321.ru

PREFIX fb: <http://rdf.freebase.com/ns/>
PREFIX db: <http://dbpedia.org/resource/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

DELETE
{
  fb:en.tommy_potter fb:people.person.date_of_birth "1918-09-21" ;
                     owl:sameAs db:Tommy_Potter .
}
WHERE {}
```

Like the comparable INSERT DATA and INSERT queries, the DELETE DATA version can be faster if you're dealing with a lot of data, and the DELETE version is more flexible. If you know exactly which triples you want to delete and won't be adding any triple patterns between the WHERE curly braces, you're better off using DELETE DATA.

Let's try out that flexibility. The following update request has a graph pattern in the WHERE clause with a single triple pattern that looks for all triples with "Tommy_Potter" as an object, and the DELETE clause deletes triples fitting the same triple pattern. Before running it, if you run the ex311.rq List All Default Graph Triples SELECT query (and not ex318.rq, which lists some but not all of the Tommy Potter triples), you'll see at least eight triples that fit this pattern. Then, run this DELETE update request:

```
# filename: ex322.ru

DELETE { ?s ?p "Tommy_Potter" }
WHERE { ?s ?p "Tommy_Potter" }
```

Running the ex311.rq List All Default Graph Triples SELECT query again will show that these triples are gone. (You might see a few with object values of "tommy_potter" in all lowercase, but this was a case-sensitive search. "String Functions" on page 164 describes how a WHERE clause can do case-insensitive pattern matching.)

DELETE clauses, like WHERE, CONSTRUCT, and INSERT clauses, can have as many triple patterns as you like.

The SPARQL Update language offers a shortcut when you're deleting triples that match a certain pattern. A DELETE WHERE update request that has no graph pattern after the DELETE keyword assumes that you want to delete any triples matched in the WHERE graph pattern. This means that the following would have the same effect as the ex322.ru update request:

```
# filename: ex323.ru

DELETE WHERE { ?s ?p "Tommy_Potter" }
```

We'll see more interesting uses of DELETE with graph patterns in the upcoming section on changing existing data, which is really just a DELETE operation combined with an INSERT one. First, though, let's look at the simplest but most powerful deletion command of all: CLEAR.

The CLEAR command clears out a graph's triples. Enter the following to clear all the triples from the default graph (which, for now, are all the triples you have, because we won't be discussing named graph examples until "Named Graphs" on page 193):

```
# filename: ex324.ru

CLEAR DEFAULT
```

After doing this, running the ex311.rq List All Default Graph Triples query will show that they're all gone.

Changing Existing Data

Changes to existing triples are performed as a delete operation followed by an insert operation in a single update request. The specification refers to this as "DELETE/INSERT." It will be easier to discuss with an example in front of us:

```
# filename: ex325.ru

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ab: <http://learningsparql.com/ns/addressbook#>

DELETE
{ ?s ab:email ?o }
INSERT
{ ?s foaf:mbox ?o }
WHERE
{?s ab:email ?o }
```

A SPARQL query processor evaluates the graph pattern in the WHERE clause, performs everything in the DELETE clause, and then performs the INSERT clause's instructions. For the update request above, it will:

1. Find all the triples with a predicate of `ab:email`.
2. Delete them.
3. Insert new triples with the same subject and object and a predicate of `foaf:mbox`.

To try this update request out,

1. If you currently have any triples in the default graph, use the CLEAR command described in the previous section to clear them out.
2. Use Fuseki's Choose File and Upload buttons to load the ex012.ttl sample address book data.
3. Execute the ex311.rq List All Default Graph Triples query in the Fuseki Query form's SPARQL Query panel to review what data you have in the dataset.
4. Execute the following CONSTRUCT query in the same panel to see what triples would be created by the INSERT/DELETE update request above:

   ```
   # filename: ex326.rq

   PREFIX foaf: <http://xmlns.com/foaf/0.1/>
   PREFIX ab: <http://learningsparql.com/ns/addressbook#>

   CONSTRUCT
   { ?s foaf:mbox ?o }
   WHERE
   {?s ab:email ?o }
   ```

   This is a nice way to see what will be added to your dataset before you actually change it.
5. Paste the ex325.ru update request above into the SPARQL Update part of the panel and click Perform update to run it.
6. Run the ex311.rq List All Default Graph Triples query again to see what triples you now have. You'll see that, in effect, the update request converted the demo `ab:email` triples to use the more well-known FOAF equivalent.

Even though the deletions happen before the insertions, the INSERT graph pattern still has all the information stored by the WHERE clause available to it.

Before writing an INSERT or INSERT/DELETE update request, a CONSTRUCT query is not just a good way to get a preview of what will be added to the data. It can also serve as a prototype that you eventually revise into the INSERT update request you need. (Remember that in Fuseki, though, you'd be executing the INSERT update request in a different part of the Fuseki Query form than the one you would use for the CONSTRUCT query.)

Let's look at a slightly more complex example. The following dataset defines three nodes of a little taxonomy using the SKOS ontology. Each concept includes a `skos:prefLabel` ("preferred label") property whose value is a string literal.

```
# filename: ex327.ttl

@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix d:    <http://learningsparql.com/ns/data#> .

d:c1 a skos:Concept ;
     skos:prefLabel "Mammal" .

d:c2 a skos:Concept ;
     skos:prefLabel "Dog" ;
     skos:broader d:c1 .

d:c3 a skos:Concept ;
     skos:prefLabel "Cat" ;
     skos:broader d:c1 .
```

The triple `d:c2 skos:broader d:c1` may look like it's saying that `d:c2` is broader than `d:c1`, but it really means that `d:c2` has a `skos:broader` value of `d:c1`. This may seem counterintuitive, but it's consistent with RDF practice—for example, `d:i0432 ab:firstName "Richard"` means that `d:i0432` has a `ab:firstName` value of "Richard".

What if you want to assign metadata to the preferred label strings like "Cat" and "Dog"? RDF lets you assign metadata to anything that you can express as a URI, because you can create triples that have that URI as a subject and any property names and values you want as the predicates and objects of those triples. Because "Cat" and "Dog" above are strings, though, you can't use them as triple subjects.

To let people assign metadata to individual terms, the W3C's SKOS eXtension for Labels (SKOS-XL) specification extends SKOS by allowing a different kind of preferred label: instead of being a string literal, it's another resource (a member of the `xl:Literal` class) with its own URI and its own properties. The most important of these properties is `xl:literalForm`, which stores the actual term such as "Cat" or "Dog". Then, this `xl:Label` instance can have as many other property name/value pairs as you want to use to store metadata about that term. The sample SKOS data above, revised to be SKOS-XL data, might look like this:

```
# filename: ex328.ttl

@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix xl:   <http://www.w3.org/2008/05/skos-xl#> .
@prefix d:    <http://learningsparql.com/ns/data#> .
@prefix dc:   <http://purl.org/dc/elements/1.1/> .

d:c1 a skos:Concept ;
     xl:prefLabel d:label1 .

d:c2 a skos:Concept ;
     xl:prefLabel d:label2 ;
     skos:broader d:c1 .
```

```
d:c3 a skos:Concept ;
     xl:prefLabel d:label3 ;
     skos:broader d:c1 .

d:label1 a xl:Label ;
         xl:literalForm "Mammal" ;
         dc:source <http://en.wikipedia.org/wiki/Mammal> .


d:label2 a xl:Label ;
         xl:literalForm "Dog" ;
         dc:source <http://en.wikipedia.org/wiki/Dog> .

d:label3 a xl:Label ;
         xl:literalForm "Cat" ;
         dc:source <http://en.wikipedia.org/wiki/Cat> .
```

Note that this SKOS-XL example includes extra triples about the source of each term, using the Dublin Core `source` property, to show SKOS-XL's flexibility. You can add all the metadata you want, from any namespaces you want, to these terms.

If you CLEAR the data currently in your Fuseki dataset (see update request ex324.ru) and upload the ex327.ttl data above (the SKOS data, not the ex328.ttl SKOS-XL data) into it, you can then run the following update request in Fuseki's SPARQL Update panel to convert the stored SKOS data into SKOS-XL data:

```
# filename: ex329.ru

PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX xl:   <http://www.w3.org/2008/05/skos-xl#>

DELETE
{ ?concept skos:prefLabel ?labelString . }
INSERT
{
  ?newURI a xl:Label ;
          xl:literalForm ?labelString .
  ?concept xl:prefLabel ?newURI .
}
WHERE
{
  ?concept skos:prefLabel ?labelString .
  BIND (URI(CONCAT("http://learningsparql.com/ns/data#",
                   ENCODE_FOR_URI(str(?labelString)))
           ) AS ?newURI)
}
```

(It doesn't add any Dublin Core source values, which I included in the ex328.ttl sample just to show how SKOS-XL data might include extra metadata.) This update request's WHERE clause finds the current uses of the `skos:prefLabel` predicate and, because the new xl:Label resources that replace them will need URIs to identify them, creates these URIs using a combination of functions that we saw in Chapter 5.

In order to create URIs for the new xl:Label instances, ex329.ru uses the ?labelString variable to pass the skos:prefLabel value to the ENCODE_FOR_URI() function, but it could use other techniques as well. There is no need to use that value in the URI.

The DELETE clause then deletes the triples that have these skos:prefLabel predicates. Next, the INSERT clause creates new members of the xl:Label class, assigning the original preferred label string as the xl:literalForm value for each new xl:Label instance, and making this xl:Label instance the xl:prefLabel value of the same concept.

In the triples being created, the concept's preferred label is specified with an xl:prefLabel property and not a skos:prefLabel one. It's a different property declared as part of the SKOS-XL specification, just as xl:Label is a new class defined in that specification.

After running the ex329.ru INSERT/DELETE update request, the ex311.rq List All Default Graph Triples query will show that the three concepts have xl:prefLabel properties instead of skos:prefLabel properties, and the xl:prefLabel values are resources with their own data. For example, concept *http://learningsparql.com/ns/data#c3* has an xl:prefLabel value of *http://learningsparql.com/ns/data#Cat*, which is an xl:Label instance that has an xl:literalForm value of "Cat":

s	p	o
<http://learningsparql.com/ns/data#c3>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2004/02/skos/core#Concept>
<http://learningsparql.com/ns/data#c3>	<http://www.w3.org/2004/02/skos/core#broader>	<http://learningsparql.com/ns/data#c1>
<http://learningsparql.com/ns/data#c3>	<http://www.w3.org/2008/05/skos-xl#prefLabel>	<http://learningsparql.com/ns/data#Cat>
<http://learningsparql.com/ns/data#c1>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2004/02/skos/core#Concept>
<http://learningsparql.com/ns/data#c1>	<http://www.w3.org/2008/05/skos-xl#prefLabel>	<http://learningsparql.com/ns/data#Mammal>
<http://learningsparql.com/ns/data#c2>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2004/02/skos/core#Concept>
<http://learningsparql.com/ns/data#c2>	<http://www.w3.org/2004/02/skos/core#broader>	<http://learningsparql.com/ns/data#c1>
<http://learningsparql.com/ns/data#c2>	<http://www.w3.org/2008/05/skos-xl#prefLabel>	<http://learningsparql.com/ns/data#Dog>

s	p	o
<http://learningsparql.com/ns/data#Cat>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2008/05/skos-xl#Label>
<http://learningsparql.com/ns/data#Cat>	<http://www.w3.org/2008/05/skos-xl#literalForm>	"Cat"
<http://learningsparql.com/ns/data#Dog>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2008/05/skos-xl#Label>
<http://learningsparql.com/ns/data#Dog>	<http://www.w3.org/2008/05/skos-xl#literalForm>	"Dog"
<http://learningsparql.com/ns/data#Mammal>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2008/05/skos-xl#Label>
<http://learningsparql.com/ns/data#Mammal>	<http://www.w3.org/2008/05/skos-xl#literalForm>	"Mammal"

Named Graphs

The W3C's SPARQL 1.1 Graph Store HTTP Protocol specification extends the SPARQL Protocol to cover communication between a client and a SPARQL processor about graphs. Because it includes HTTP ways to say "here's a graph to add to the dataset" or "delete the graph *http://my/fine/graph* from the dataset," it provides an alternative approach to the SPARQL Update language methods for dealing with entire graphs that this section describes.

To get a feel for creating, adding to, deleting from, and replacing triples in graphs (as well as deleting and replacing entire graphs), we'll start by playing with each of SPARQL Update's relevant keywords using simple dummy data in which resources named `d:x` and `d:w` get `dm:tag` values of spelled-out numbers liked "one" and "two". Then, we'll review several of the update operations using more realistic data.

To provide a baseline for our next few experiments, the next update request does two operations: it clears the triples in the default graph and then loads two more into it.

SPARQL Update lets you connect multiple operations with semicolons.

Go ahead and execute this update request in the SPARQL Update part of the Fuseki form.

```
# filename: ex330.ru
```

```
PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

CLEAR DEFAULT;

INSERT DATA
{
  d:x dm:tag "one" .
  d:x dm:tag "two" .
}
```

Now we'll put some new triples into a specific named graph. As with the querying of named graphs, which we saw in Chapter 3, we refer to a graph pattern of triples from a particular graph by preceding the graph pattern with the keyword GRAPH and the name of the graph. We'll use the same INSERT operation that we used earlier to insert triples into the default graph.

When you insert triples into a graph that doesn't exist, a SPARQL processor creates that graph.

The following update request will create the *d:g1* graph and add the triple shown to it:

```
# filename: ex331.ru

PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT DATA
{ GRAPH d:g1
  { d:x dm:tag "three" }
}
```

Because graph names are URIs like any other, you can represent them as prefixed names. Depending on how you choose to organize your data, you may put your named graphs in their own namespace, but in order to keep these examples simple, I put them in the same *http://learningsparql.com/ns/data#* namespace as the other sample data.

After you run ex331.ru in the SPARQL Update part of the Fuseki form, run the following SELECT query in the SPARQL Query part of the Fuseki form:

```
# filename: ex332.rq

SELECT ?g ?s ?p ?o
WHERE
{
  { ?s ?p ?o }
  UNION
```

```
    { GRAPH ?g { ?s ?p ?o } }
  }
```

This really is the List All Triples query, because it lists a union of all triples in the default graph and all the triples in any named graph along with the associated graph names. With the triples inserted by the previous two INSERT queries, this SELECT query will show you the following:

g	s	p	o
	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	"one"
	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	"two"
<http://learningsparql.com/ns/data#g1>	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	"three"

Bookmarking the results of the ex332.rq query is a simple way to rerun the query whenever you want. You may find this handy for reviewing the effects of this chapter's remaining update queries.

This next update request resembles the last INSERT update request, but inserts another triple into the same graph:

```
# filename: ex333.ru

PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT DATA
{ GRAPH d:g1
  { d:x dm:tag "four" . }
}
```

After you execute it and run the List All Triples query, you'll see that the *d:g1* graph now has two triples:

g	s	p	o
	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	"one"
	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	"two"
<http://learningsparql.com/ns/data#g1>	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	"three"
<http://learningsparql.com/ns/data#g1>	<http://learningsparql.com/ns/data#x>	<http://learningsparql.com/ns/demo#tag>	"four"

Dropping Graphs

Before we learn how to delete a graph, add a second named graph with the following update request and run the ex332.rq List All Triples SELECT query to see what's there:

```
# filename: ex334.ru

PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT DATA
{ GRAPH d:g2
  {
    d:x dm:tag "five" .
    d:x dm:tag "six" .
  }
}
```

The DROP operation deletes a graph from the dataset. The following drops our first named graph; run it, then run the List All Triples SELECT query, and you'll see that the default graph's triples and the *d:g2* triples are still there, but the *d:g1* triples are gone:

```
#filename: ex335.ru

PREFIX d: <http://learningsparql.com/ns/data#>

DROP GRAPH d:g1
```

DROP DEFAULT clears the default graph. (You can't actually drop a default graph, because it always exists, even if it's empty.) Execute the following, run the ex332.rq List All Triples SELECT query, and you'll see that the *d:g2* graph of triples is there, but the default graph's triples are now gone:

```
# filename: ex336.ru

DROP DEFAULT
```

Run the ex330.ru update request again to put some triples back into the default graph, and then run the ex332.rq List All Triples SELECT query to make sure that they're there. Now you're ready for the most powerful update request of all: DROP ALL, which drops the default graph and all named graphs—in other words, it deletes everything.

```
# filename: ex337.ru

DROP ALL
```

Try it out, then run the List All Triples SELECT query and you'll see just how much these two little words can do to a dataset.

Any command in any language that says "delete everything" is something to be careful with. It's always worth an extra pause before actually executing it. SPARQL Update offers no UNDO operation, although your triplestore, like any database management system, may offer something like this as part of a set of commit and rollback features.

Next, we'll learn about dropping all named graphs while leaving the default graph alone. To set up some data to test this, put all the triples from the last few INSERT update queries back into Fuseki with this next update request. Run it, then run the ex332.rq List All Triples SELECT query to see the six triples spread out across the default graph and the two named graphs:

```
# filename: ex338.ru

PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT DATA
{
  d:x dm:tag "one" .
  d:x dm:tag "two" .

  GRAPH d:g1
  {
    d:x dm:tag "three" .
    d:x dm:tag "four" .
  }

  GRAPH d:g2
  {
    d:x dm:tag "five" .
    d:x dm:tag "six" .
  }
}
```

The DROP NAMED graph drops all named graphs. Try the following, and then run the List All Triples SELECT query. You'll see that the *d:g1* and *d:g2* graphs are gone, but the default graph's two triples are still there:

```
# filename: ex339.ru

DROP NAMED
```

Earlier we learned how CLEAR DEFAULT clears all the triples in the default graph. If you substitute the GRAPH keyword and the name of a specific graph for the DEFAULT keyword, you can clear the triples from that graph. For example, `CLEAR GRAPH <http://mygraph>` deletes all triples from the named graph *http://mygraph*. Also, `CLEAR NAMED` removes triples from all named graphs, and `CLEAR ALL` removes all triples from all named graphs and from the default graph.

Another way to create graphs is with the CREATE GRAPH operation, "for stores that record empty graphs," as the SPARQL Update spec puts it. (If you wondered about the difference between the DROP and CLEAR operations, DROP removes complete graphs and CLEAR removes the triples from within them while leaving the empty graphs—"for stores that record empty graphs.")

The following update request would create a new *d:g3* graph. If you try it with Fuseki 0.2 and then run the ex332.rq List All Triples SELECT query, you won't see any indication that Fuseki has recorded the existence of this graph:

```
# filename: ex340.ru

PREFIX d: <http://learningsparql.com/ns/data#>

CREATE GRAPH d:g3
```

After running the ex340.ru update request, running the following query in Fuseki's SPARQL Query panel should list all named graphs with or without any triples in them. The result has no indication that Fuseki 0.2 knows about the *d:g3* graph, but the CREATE command would still be worth trying with other triplestores that you use and with future versions of Fuseki:

```
# filename: ex341.rq

SELECT ?g
WHERE
{ GRAPH ?g {} }
```

Named Graph Syntax Shortcuts: WITH and USING

The WITH keyword tells the SPARQL processor the name of a graph to use whenever a graph isn't named in the remainder of the update request. For example, the following does the same thing as the ex334.ru update request earlier, inserting the two triples shown into the *d:g2* graph:

```
# filename: ex342.ru

PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

WITH d:g2
INSERT
{
  d:x dm:tag "five" .
  d:x dm:tag "six" .
}
WHERE {}
```

INSERT DATA will not work with the WITH keyword, which is why ex342.ru has WHERE {} at the end. Of course, you can put triple patterns in the WHERE clause's curly braces and then reference their variables in the INSERT clause's triple patterns.

If an update request only mentions a particular named graph once, like ex334.ru does with `d:g2`, naming that graph with the WITH keyword instead of the GRAPH keyword makes little difference. When we get to updating and replacing triples in graphs, though, we'll see how WITH can make your queries less verbose by saving you the trouble of naming the same graph over and over.

The USING keyword does for update queries what FROM does for SELECT queries: it specifies a graph that the WHERE clause should look at. Before we see it in action, run the following update request, which adds two new triples to the default graph. Like the two triples in the *d:g2* graph, these new ones have object values of "five" and "six", but they have subjects of *d:w* instead of *d:x*.

```
# filename: ex343.ru

PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT DATA
{
  d:w dm:tag "five" .
  d:w dm:tag "six" .
}
```

When you use the USING keyword, don't use the WITH keyword.

This next update request has a WHERE clause that looks for triples with a predicate of *dm:tag* and an object of "five" or "six". It then inserts copies of these triples into a new *d:g4* graph.

```
# filename: ex344.ru

PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

INSERT
{ GRAPH d:g4
  { ?s dm:tag "five", "six" . }
}
USING d:g2
WHERE
{
  ?s dm:tag "five" .
  ?s dm:tag "six" .
}
```

After running it, run the ex332.rq List All Triples SELECT query, and you'll see that the two triples inserted into graph *d:g4* both have subjects of *d:x*. Although the default graph has triples that match the WHERE clause's patterns (the ones that you inserted with update request ex343.ru), the USING keyword specifically tells the SPARQL processor to look in the *d:g2* graph for triples that match those patterns, so those are the ones that got copied to *d:g4*.

If the USING keyword in an update request is like FROM in a SELECT query, then USING NAMED is like FROM NAMED: it specifies a graph that will be referenced by name. If the ex344.ru graph had said USING NAMED instead of just USING, the query processor wouldn't have found those triples in the *d:g2* graph unless the name was explicitly included in the WHERE clause, like this:

```
# filename: ex345.ru

PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>

WITH d:g4
INSERT
{
  ?s dm:tag "five" .
  ?s dm:tag "six" .
}
USING NAMED d:g2
WHERE
{ GRAPH d:g2
  {
    ?s dm:tag  "five" .
    ?s dm:tag  "six" .
  }
}
```

Deleting and Replacing Triples in Named Graphs

Before trying some queries that delete triples from specific graphs, run the DROP ALL update request (ex337.ru) to clear out the dataset, run the ex338.ru update request that adds triples to two new graphs and the default graph, and then run the ex332.rq List All Triples SELECT query to review the data that you're about to delete from.

Just as you can use DELETE DATA when you know exactly which triples you want to delete from the default graph, you can use it when you know which triples you want to delete from named graphs. The following will delete the specified triple from the *d:g2* graph:

```
# filename: ex346.ru

PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX dm: <http://learningsparql.com/ns/demo#>
```

```
DELETE DATA
{ GRAPH d:g2
  { d:x dm:tag "six" }
}
```

Try it, run the List All Triples SELECT query, and you'll see that the query processor has removed that triple from the *d:g2* graph.

We also saw that DELETE without the DATA keyword lets you delete triples matching triple patterns. This works with named graphs, too; you can even use a variable in place of a graph name, like this:

```
# filename: ex347.ru

DELETE
{ GRAPH ?g { ?s ?p "three" } }
WHERE
{ GRAPH ?g { ?s ?p "three" } }
```

After running this and then running the List All Triples SELECT query, you'll see that the triple with "three" as an object is gone from the *d:g1* graph.

We saw that the WITH keyword lets you tell the SPARQL processor the name of a graph to use whenever a graph isn't explicitly named. While the ex347.ru update request above had to identify the graph in both the DELETE and WHERE clauses (although it used the placeholder variable `?g`, there still had to be something there), the following one just names the graph once, and both the DELETE and WHERE clauses know that they're supposed to act on graph *d:g1*. If you run this query and then run the ex332.rq List All Triples SELECT query, you'll see that the "four" triple has been removed from that graph:

```
# filename: ex348.ru

PREFIX d: <http://learningsparql.com/ns/data#>

WITH d:g1
DELETE { ?s ?p "four"}
WHERE { ?s ?p "four"}
```

As with the INSERT update queries, the USING keyword in a DELETE update request overrides whatever a WITH clause says about a graph to delete from.

Replacement of triples in named graphs is a combination of deleting and inserting them, just like it was when replacing triples in the default graph. This is where the WITH keyword becomes particularly useful. First, let's look at an example of the explicit, verbose way to replace triples within a specific graph:

```
# filename: ex349.ru

PREFIX d: <http://learningsparql.com/ns/data#>

DELETE
{ GRAPH d:g2 { ?s ?p "five" } }
```

```
INSERT
{ GRAPH d:g2 { ?s ?p "cinco" } }
WHERE
{ GRAPH d:g2 { ?s ?p "five" } }
```

This update request looks for triples in the *d:g2* graph that have "five" as an object and replaces them with triples that have the same subject and object but "cinco" as an object. Go ahead and run it, then run the ex332.rq List All Triples to see the effect that this update request had on the dataset.

This next update request does the same thing, but uses the WITH keyword so that it only has to mention *d:g2* once. With no USING keywords to override this choice of graph, the DELETE, INSERT, and WHERE clauses will all take their actions on that named graph.

```
# filename: ex350.ru

PREFIX d: <http://learningsparql.com/ns/data#>

WITH d:g2
DELETE
{ ?s ?p "five" }
INSERT
{ ?s ?p "cinco" }
WHERE
{ ?s ?p "five" }
```

Let's try some graph update requests with a more realistic set of data. The following update request drops all the graphs, default or otherwise, and creates three new named graphs:

```
# filename: ex351.ru

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX g:  <http://learningsparql.com/graphs/>

DROP ALL;

INSERT DATA
{

  # people
  GRAPH g:people
  {
    d:i0432 ab:firstName "Richard" ;
            ab:lastName  "Mutt" ;
            ab:email     "richard49@hotmail.com" .

    d:i9771 ab:firstName "Cindy" ;
            ab:lastName  "Marshall" ;
            ab:email     "cindym@gmail.com" .

    d:i8301 ab:firstName "Craig" ;
            ab:lastName  "Ellis" ;
```

```
              ab:email     "c.ellis@usairwaysgroup.com" .
    }

    # courses
    GRAPH g:courses
    {
      d:course34 ab:courseTitle "Modeling Data with OWL" .
      d:course71 ab:courseTitle "Enhancing Websites with RDFa" .
      d:course59 ab:courseTitle "Using SPARQL with non-RDF Data" .
      d:course85 ab:courseTitle "Updating Data with SPARQL" .
    }

    # who's taking which courses

    GRAPH g:enrollment
    {
      d:i8301 ab:takingCourse d:course59 .
      d:i9771 ab:takingCourse d:course34 .
      d:i0432 ab:takingCourse d:course85 .
      d:i0432 ab:takingCourse d:course59 .
      d:i9771 ab:takingCourse d:course59 .
    }
  }
```

Run the ex332.rq List All Triples SELECT query to see what you'll be working with for the next few examples.

Now, let's say that we have an application used by students to sign up for courses. They're filling out web forms to search and sign up for these courses, but the backend is implemented with a triplestore. (We'll learn more about creating such applications in Chapter 7.)

Shortly after adding the data in ex351.ru to our course tracking application's dataset, we hear from the education department. They tell us that there are some corrections to the course listings: course 34 is now called "Modeling Data with RDFS and OWL" and there's a new course, number 86, called "Querying and Updating Named Graphs." We could make the corrections with this update request:

```
# filename: ex352.ru

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d: <http://learningsparql.com/ns/data#>
PREFIX g: <http://learningsparql.com/graphs/>

DELETE
{ GRAPH g:courses
    { d:course34 ab:courseTitle ?courseTitle }
}
INSERT
{ GRAPH g:courses
  { d:course34 ab:courseTitle "Modeling Data with RDFS and OWL" . }
}
WHERE
{ GRAPH g:courses
```

```
    { d:course34 ab:courseTitle ?courseTitle }
  } ;

  INSERT DATA
  { GRAPH g:courses
    { d:course86 ab:courseTitle "Querying and Updating Named Graphs" . }
  }
```

It applies the techniques we learned about earlier for updating the triples in a graph by deleting course 34's existing title value and then adding a new one. Along the way, it also adds a title for the new course 86.

Another approach for updating the course data would be to replace the entire courses graph with two steps. The following update request drops the graph and then inserts a corrected version of it:

```
# filename: ex353.ru

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d:  <http://learningsparql.com/ns/data#>

DROP GRAPH <http://learningsparql.com/graphs/courses> ;

INSERT DATA
{
  GRAPH <http://learningsparql.com/graphs/courses>
  {
    d:course34 ab:courseTitle "Modeling Data with RDFS and OWL" .
    d:course71 ab:courseTitle "Enhancing Websites with RDFa" .
    d:course59 ab:courseTitle "Using SPARQL with non-RDF Data" .
    d:course85 ab:courseTitle "Updating Data with SPARQL" .
    d:course86 ab:courseTitle "Querying and Updating Named Graphs" .
  }
}
```

After using either technique to fix the course listings, run the following SELECT query to see who's taking which courses:

```
# filename: ex354.rq

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX d:  <http://learningsparql.com/ns/data#>
PREFIX g:  <http://learningsparql.com/graphs/>

SELECT ?first ?last ?courseTitle WHERE
{

  { GRAPH g:people
    { ?student ab:firstName ?first ;
               ab:lastName ?last .
    }
  }

  { GRAPH g:enrollment
    { ?student ab:takingCourse ?course . }
  }

  { GRAPH g:courses
    { ?course ab:courseTitle ?courseTitle . }
  }

}
```

Note how this query asks for the triples by identifying the named graphs where they're stored.

The results of this query show that the data modeling course being taken by Cindy Marshall has the up-to-date name:

first	last	courseTitle
"Cindy"	"Marshall"	"Modeling Data with RDFS and OWL"
"Cindy"	"Marshall"	"Using SPARQL with non-RDF Data"
"Richard"	"Mutt"	"Using SPARQL with non-RDF Data"
"Richard"	"Mutt"	"Updating Data with SPARQL"
"Craig"	"Ellis"	"Using SPARQL with non-RDF Data"

Relational database developers might compare the named graphs in this dataset to the tables of a relational database, because each stores a specific set of data and the query cross-references between them to list the desired data. There are some parallels here, but remember that named graphs are much more flexible than tables. Storing new kinds of data in one doesn't require any schema modification. Also, because the graphs are referenced by the same kind of IDs as the data itself (that is, URIs), the named graphs themselves can have metadata assigned to them or be the metadata values of other resources. The use of URIs also lets us cross-reference this data with data from completely different datasets.

Summary

In this chapter, we learned:

- How to start up the Fuseki SPARQL server from its jar file after downloading the distribution zip file.
- How the INSERT operation (with and without the DATA keyword) and the LOAD keyword can add local and remote triples your dataset's default graph.
- How DELETE and DELETE DATA can remove triples from the default graph.

- How to combine the INSERT and DELETE operations to replace existing triples in a default graph.
- How to create named graphs, and how to INSERT triples into and DELETE them from these graphs, along with several ways to refer to the graphs being managed.

CHAPTER 7

Building Applications with SPARQL: A Brief Tour

SPARQL isn't something for all end users on the web to learn any more than JavaScript is. It's a tool that gives you and your applications access to a greater variety of data and metadata. In this chapter, we'll learn a little about how to incorporate it into applications so that you can bring these benefits to users who never need to know about SPARQL or the related standards.

Before looking at how different aspects of SPARQL technology can contribute to an application, let's step back and look at the bigger picture. What role does any query language play in an application? In a typical application scenario, regardless of the choice of technology used, you might have a central store of data and several client processes sending requests to that central store for delivery and perhaps updating of that data. These requests usually ask for subsets of the data that meet certain conditions, and they may specify that the columns of results be in a particular order and that the results are sorted according to the values of one or more of the columns. Upon receiving the results, the client processes use the returned information to achieve their own goals —perhaps rendering information on a display, or turning a machine on or off, or saving some data for use in future calculations.

Using a web browser to look through an online clothing retailer's T-shirt selection or using a specialized app on your phone to check next week's weather both fit into this scenario. Perhaps the clothing and weather data are stored in relational databases, and SQL queries are being sent between the processes assembling the information you requested; perhaps the data is stored in a specialized NoSQL database and queries within the application use a customized language developed for that implementation. The choice is up to the system architects for that application, and the end users don't (and shouldn't have to) care. They just want the information displayed on their screens quickly.

As more system architects see the advantages of the RDF data model for certain kinds of applications, they're using triplestores as the backend storage system and SPARQL as the query language for processes within the system to request information from each other. One aspect of this approach is the flexibility we gain from the ease with which we can treat different data formats as triples even if they weren't designed that way—for example, by using a middleware layer that lets us send SPARQL queries to relational databases. Because of this, there doesn't necessarily need to be a triplestore in the overall architecture of an application that takes advantage of SPARQL. Still, it's ultimately about processes sending requests to storage resources in order to get their work done.

In this chapter, we'll take a quick look at how various SPARQL-related technologies can contribute to different parts of this picture:

- "SPARQL and Web Application Development" on page 208: You can send a query to a SPARQL endpoint via HTTP from a program or a web form with very little code.
- "SPARQL Query Results XML Format" on page 217: This is the W3C standard format for query processors to return query results. It's a simple, XML-based format that lets you use XSLT or another XML tool to convert the results into whatever you like. We'll also learn about a JSON alternative.
- "SPARQL Processors" on page 221: We've seen how ARQ can read a query and some data and then run the query on that data. In Chapter 6, we saw how Fuseki can store data and respond to queries about that data. More robust, scalable processors are also available, typically included as part of a triplestore. There are also middleware options to let you run SPARQL queries against relational databases and other data formats.

SPARQL and Web Application Development

Whether your application requests data from a public SPARQL endpoint such as DBpedia or from an endpoint included with a triplestore running on your laptop, the endpoint is identified by a URI. The most common way to send a query to an endpoint is to add an escaped version of the query as a parameter to that URI.

For example, let's say I want to ask DBpedia when Elvis Presley was born. I want to send the following query to the DBpedia endpoint at the URI `http://dbpedia.org/sparql`:

```
# filename: ex355.rq
SELECT ?elvisbday WHERE {
  <http://dbpedia.org/resource/Elvis_Presley>
  <http://dbpedia.org/property/dateOfBirth> ?elvisbday .
}
```

I could paste this query into the form on DBpedia's SNORQL interface, but I want to have a program send the query and retrieve the answer so that I can incorporate this into an application.

First, I create a version of the query with the appropriate characters escaped so that they won't cause problems when appended to the endpoint URI. Any modern programming language lets you do this with a simple function call; when I do it with the query above, I get this (carriage returns added here for easier display on this page):

```
SELECT%20%3Felvisbday%20WHERE%20%7B%0A%20%20
%3Chttp%3A%2F%2Fdbpedia.org%2Fresource%2FElvis_Presley%3E%20%0A%20%20%3Chttp
%3A%2F%2Fdbpedia.org%2Fproperty%2FdateOfBirth%3E%20%3Felvisbday%20.%0A%7D%0A
```

DBpedia's endpoint expects the query to be passed as the value of a `query` parameter, so assembling all the pieces gives us this (again, the actual assembled URI would not have the carriage returns shown here):

```
http://dbpedia.org/sparql?query=SELECT%20%3Felvisbday%20WHERE%20%7B%0A%20%20
%3Chttp%3A%2F%2Fdbpedia.org%2Fresource%2FElvis_Presley%3E%20%0A%20%20%3Chttp
%3A%2F%2Fdbpedia.org%2Fproperty%2FdateOfBirth%3E%20%3Felvisbday%20.%0A%7D%0A
```

Other SPARQL endpoints may or may not call the parameter `query` like DBpedia does. Check the documentation of any that you are using to make sure. There may also be other parameters that give you greater control over how the query results are returned.

Pasting that URI (without the carriage returns) into most browsers will show you the result of the query, but again, this time I don't want to paste this into a browser; I want to write a program that can send that query off and then store the retrieved result.

A command line utility such as wget or curl can take this URI as a parameter and save the returned results in a file. Because the returned results are well-formed XML, you can also supply this URI as the document parameter to a command line XSLT processor such as Saxon or Xalan. When supplying such a complex URI as a parameter to a command line utility, you may need to enclose the URI in quotes, depending on your operating system. (Both wget and curl come with the Linux distributions that I checked, and curl comes with Mac OS. You can easily get free versions of wget for Mac OS and both programs for Windows.)

Here's a Python script that escapes the query, builds the URI above, and then uses it to retrieve Elvis's birthday:

```
# filename: ex358.py
# Send SPARQL query to SPARQL endpoint, store and output result.

import urllib2
```

```
endpointURL = "http://dbpedia.org/sparql"
query = """
SELECT ?elvisbday WHERE {
  <http://dbpedia.org/resource/Elvis_Presley>
  <http://dbpedia.org/property/dateOfBirth> ?elvisbday .
}
"""
escapedQuery = urllib2.quote(query)
requestURL = endpointURL + "?query=" + escapedQuery
request = urllib2.Request(requestURL)

result = urllib2.urlopen(request)
print result.read()
```

The output is in the XML format that we'll learn about in "SPARQL Query Results XML Format" on page 217:

```
<sparql xmlns="http://www.w3.org/2005/sparql-results#"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.w3.org/2001/sw/DataAccess/rf1/result2.xsd">
  <head>
    <variable name="elvisbday"/>
  </head>
  <results distinct="false" ordered="true">
    <result>
      <binding name="elvisbday">
        <literal
            datatype="http://www.w3.org/2001/XMLSchema#date">1935-01-08</literal>
      </binding>
    </result>
  </results>
</sparql>
```

Here's a Perl script that does the same thing:

```
# filename: ex360.pl
# Send SPARQL query to SPARQL endpoint, store and output result.

use LWP::UserAgent;
use URI::Escape;

$endpointURL = "http://dbpedia.org/sparql";
$query = "
SELECT ?elvisbday WHERE {
  <http://dbpedia.org/resource/Elvis_Presley>
  <http://dbpedia.org/property/dateOfBirth> ?elvisbday .
}
";
$escapedQuery = uri_escape($query);
$requestURL = $endpointURL . "?query=" . $escapedQuery;
$request = new HTTP::Request 'GET' => $requestURL;
$ua = new LWP::UserAgent;

$result = $ua->request($request);
print $result->content;
```

Serious Python and Perl developers will know ways to make these scripts more robust. They may also know other libraries to perform some of the steps more efficiently.

Specialized SPARQL libraries for your favorite programming language can store the result of a SPARQL endpoint query in data structures that are native to that programming language instead of storing it in one big XML string like the two examples above do. These libraries include SPARQLWrapper for Python, RDF-Query for Perl, SPARQL/Grammar for Ruby, and the ARQ source code for Java.

The remaining examples in this section are in Python, but the same principles would apply with any development language.

For example, the following Python script uses the SPARQLWrapper library to query the SPARQL endpoint at the Linked Movie Database about actors who have appeared in at least one Steven Spielberg movie and also at least one Stanley Kubrick movie. (You'll need to install that library and the JSON one before running this example.) It requests the results in a JSON format so that it can easily iterate through the returned data:

```
# filename: ex361.py
# Query Linked Movie database endpoint about common actors of two directors

from SPARQLWrapper import SPARQLWrapper, JSON

sparql = SPARQLWrapper("http://data.linkedmdb.org/sparql")
queryString = """
PREFIX m: <http://data.linkedmdb.org/resource/movie/>
SELECT DISTINCT ?actorName WHERE {

  ?dir1     m:director_name "Steven Spielberg" .
  ?dir2     m:director_name "Stanley Kubrick" .

  ?dir1film m:director ?dir1 ;
            m:actor ?actor .

  ?dir2film m:director ?dir2 ;
            m:actor ?actor .

  ?actor    m:actor_name ?actorName .
}
"""

sparql.setQuery(queryString)
sparql.setReturnFormat(JSON)
results = sparql.query().convert()
```

```
if (len(results["results"]["bindings"]) == 0):
  print "No results found."
else:
  for result in results["results"]["bindings"]:
      print result["actorName"]["value"]
```

Here are the results:

```
Wolf Kahler
Slim Pickens
Tom Cruise
Arliss Howard
Ben Johnson
Scatman Crothers
Philip Stone
```

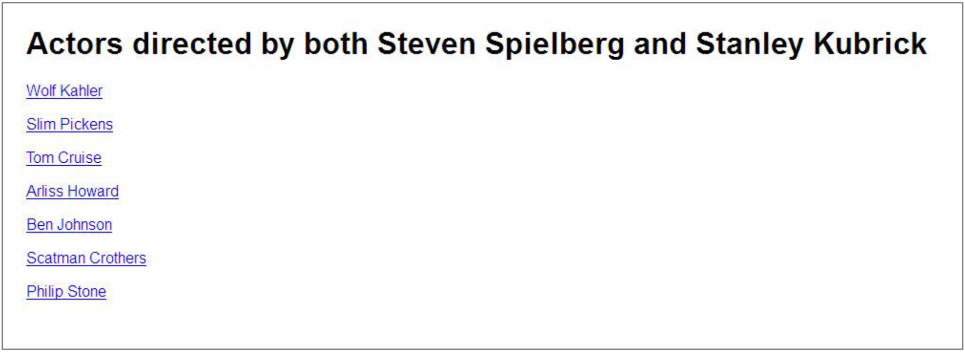
Creating this kind of list would be very time-consuming without SPARQL and a collection of data that let you search the connections between actors, films, and directors. If you want to create a similar actor list for other pairs of directors, all you need to do is to replace Spielberg and Kubrick's name and run the script again. (Make sure to use each director's "official" name—even if Martin Scorsese's friends call him Marty, a search of the Linked Movie Database for data on "Marty Scorsese" won't find anything.) Wouldn't it be nice, though, if film buffs who've never heard of SPARQL could enter two director names on a web form and then see the list of common actors by just clicking a button?

With a few modifications to the Python script above, you can create this. To make it easier to understand how, we'll make two rounds of revisions to ex361.py. Here is the first round:

```
# filename: ex363.py
# Query Linked Movie database endpoint about common actors of
# two directors and output HTML page with links to Freebase.

from SPARQLWrapper import SPARQLWrapper, JSON

director1 = "Steven Spielberg"
director2 = "Stanley Kubrick"

sparql = SPARQLWrapper("http://data.linkedmdb.org/sparql")
queryString = """
PREFIX m:    <http://data.linkedmdb.org/resource/movie/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT DISTINCT ?actorName ?freebaseURI WHERE {

  ?dir1     m:director_name "DIR1-NAME" .
  ?dir2     m:director_name "DIR2-NAME" .

  ?dir1film m:director ?dir1 ;
            m:actor ?actor .

  ?dir2film m:director ?dir2 ;
            m:actor ?actor .
  ?actor    m:actor_name ?actorName ;
            foaf:page ?freebaseURI .
}
"""

queryString = queryString.replace("DIR1-NAME",director1)
queryString = queryString.replace("DIR2-NAME",director2)
sparql.setQuery(queryString)

sparql.setReturnFormat(JSON)
results = sparql.query().convert()

print """
<html><head><title>results</title>
<style type="text/css"> * { font-family: arial,helvetica}</style>
</head><body>
"""

print "<h1>Actors directed by both " + director1 + " and " + director2 + "</h1>"

if (len(results["results"]["bindings"]) == 0):
  print "<p>No results found.</p>"

else:

  for result in results["results"]["bindings"]:
      actorName = result["actorName"]["value"]
      freebaseURI = result["freebaseURI"]["value"]
      print "<p><a href=\"" + freebaseURI + "\">" + actorName + "</p>"

print "</body></html>"
```

The ex363.py script has three basic differences from ex361.py:

- Instead of hardcoding the directors' names in the query, the script stores them in `director1` and `director2` Python variables. A replace function then replaces the strings "DIR1-NAME" and "DIR2-NAME" in the query with the values of these variables.
- The SPARQL query asks for the URI of each actor's Freebase page in addition to his or her name, because this property is also stored in the Linked Movie Database for each actor.
- Instead of a simple list of names, the final output is an HTML document with an `h1` title naming the directors, and each actor's name is a link to the Freebase page for that actor, as shown in Figure 7-1.


Actors directed by both Steven Spielberg and Stanley Kubrick

Wolf Kahler

Slim Pickens

Tom Cruise

Arliss Howard

Ben Johnson

Scatman Crothers

Philip Stone

Figure 7-1. Web page created by ex363.py Python script as displayed by browser

The final version of this script, which will serve as the CGI script called by the web form that we'll create for it, adds the following to ex363.py:

- The new first line points to the Python executable on the local system.
- Along with the libraries it imported before, it imports a few that it needs to work as a CGI in a hosted environment.
- It takes values passed to it in **`dir1`** and **`dir2`** variables and stores them in Python **`director1`** and **`director2`** variables.
- After preparing the SPARQL query to send off the same way it did before, the script sends the query from within a try/except block so that it can check for communication problems before attempting to render the results.
- Like any CGI script that creates HTML and sends it to a browser, it sends a `Content-type` header and two carriage returns before sending the actual web page.

```
#!/usr/local/bin/python
# filename: ex364.cgi
# CGI version of ex363.py

import sys
sys.path.append('/usr/home/bobd/lib/python/') # needed for hosted version
from SPARQLWrapper import SPARQLWrapper, JSON
import cgi

form = cgi.FieldStorage()
director1 = form.getvalue('dir1')
director2 = form.getvalue('dir2')

sparql = SPARQLWrapper("http://data.linkedmdb.org/sparql")
queryString = """
PREFIX m:    <http://data.linkedmdb.org/resource/movie/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT DISTINCT ?actorName ?freebaseURI WHERE {

  ?dir1     m:director_name "DIR1-NAME" .
  ?dir2     m:director_name "DIR2-NAME" .

  ?dir1film m:director ?dir1 ;
            m:actor ?actor .

  ?dir2film m:director ?dir2 ;
            m:actor ?actor .

  ?actor    m:actor_name ?actorName ;
            foaf:page ?freebaseURI .
}
"""

queryString = queryString.replace("DIR1-NAME",director1)
queryString = queryString.replace("DIR2-NAME",director2)
sparql.setQuery(queryString)

sparql.setReturnFormat(JSON)

try:
  results = sparql.query().convert()
  requestGood = True
except Exception, e:
  results = str(e)
  requestGood = False

print """Content-type: text/html

<html><head><title>results</title>
<style type="text/css"> * { font-family: arial,helvetica}</style>
</head><body>
"""

if requestGood == False:
  print "<h1>Problem communicating with the server</h1>"
  print "<p>" + results + "</p>"
elif (len(results["results"]["bindings"]) == 0):
  print "<p>No results found.</p>"

else:

  print "<h1>Actors directed by both " + director1 + \
        " and " + director2 + "</h1>"

  for result in results["results"]["bindings"]:
    actorName = result["actorName"]["value"]
    freebaseURI = result["freebaseURI"]["value"]
    print "<p><a href=\"" + freebaseURI + "\">" + actorName + "</p>"

print "</body></html>"
```

We can test this script before we've created the form by pasting a URL like the following into a browser with the domain and directory names adjusted for where you have the ex364.cgi script stored. Use any director names you like, substituting the plus sign for any spaces, like this:

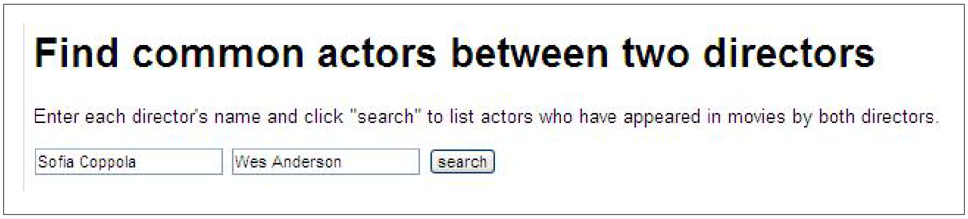
```
http://learningsparql.com/examples/ex364.cgi?dir1=John+Ford&dir2=Howard+Hawks
```

The final step is to create the web form that your film buff friends will fill out to list the actors that two directors have in common. All you need is a simple form like this:

```
<!-- filename: ex365.html -->
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Find common actors between two directors</title>
    <style type="text/css"> * { font-family: arial,helvetica}</style>
  </head>
  <body>
    <h1>Find common actors between two directors</h1>

    <form action="ex364.cgi" method="get">

      <p>Enter each director's name and click "search" to list actors
      who have appeared in movies by both directors.</p>
      <p>
        <input type="text" name="dir1"/>
        <input type="text" name="dir2"/>
        <input type="submit" value="search"/>
      </p>

    </form>

  </body>
</html>
```

When the search button is clicked, the form will pass the two entered values in the `dir1` and `dir2` variables. If a user entered "John Ford" and "Howard Hawks" into the two fields, the form would essentially call the CGI script the same way the URI above with these two directors' names does.

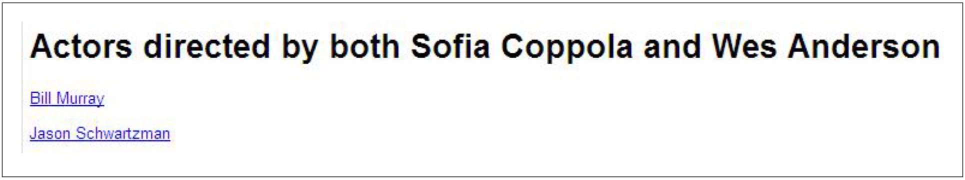
To try out this little application, first store ex365.html on a server capable of executing CGI scripts in the same directory as ex364.cgi. Then, display this form in a web browser and enter two director names on the form, as shown in Figure 7-2. Figure 7-3 shows the result of clicking search in Figure 7-2.


Find common actors between two directors

Enter each director's name and click "search" to list actors who have appeared in movies by both directors.

Sofia Coppola | Wes Anderson

Figure 7-2. ex365.html in a web browser with the two fields filled out


Actors directed by both Sofia Coppola and Wes Anderson

Bill Murray

Jason Schwartzman

Figure 7-3. Results of clicking "search" in

The ex365.html web page and the HTML generated by the ex364.cgi script are both very simple, but there's no reason not to bring in all of your CSS and JavaScript skills to make these web pages as sophisticated as you need them to be. For example, you could use some of HTML5's new features, or you could use the jQuery Mobile JavaScript and CSS libraries to make your application mobile-friendly.

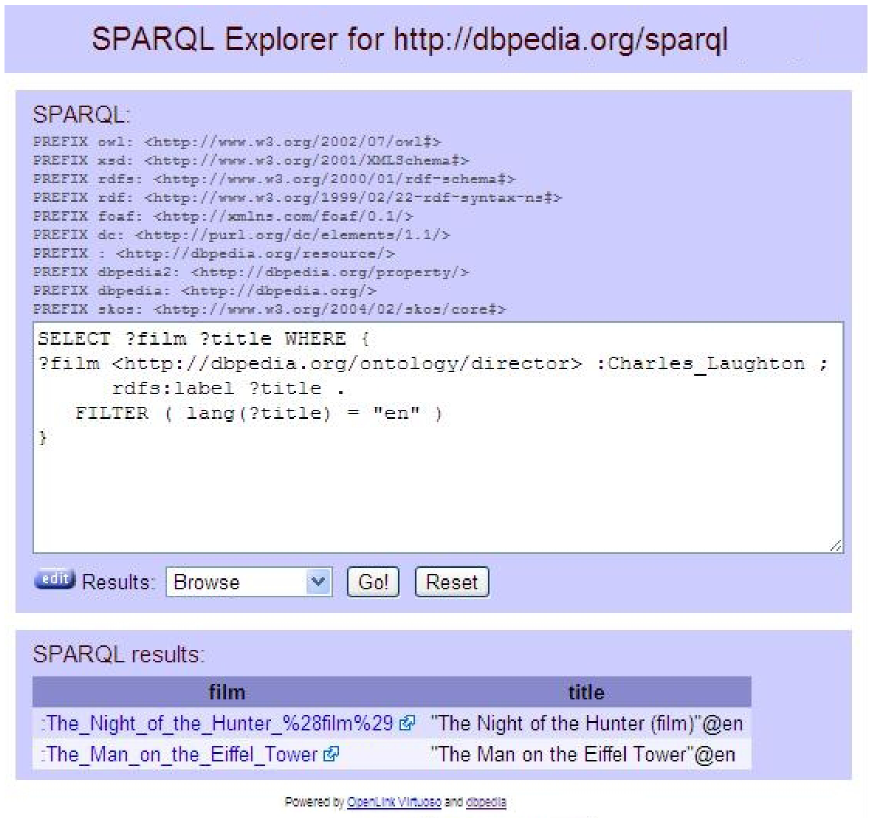
On the other hand, the script doesn't have to generate HTML or be called from a web form. It can return JSON, more specialized XML, or anything that may be useful to your client process that is making the request. With the results of your SPARQL query loaded into the typical data structures of your favorite development language or returned as XML, it's pretty easy to turn that data into whatever you want. The ability to invoke this script with a URL means that you have everything you need to create a specialized web service that takes advantage of data from a SPARQL endpoint (or endpoints!) and makes it available as part of a service-oriented architecture.

We'll take a closer look at the SPARQL Query Results XML format in the next section.

SPARQL Query Results XML Format

The SPARQL Query Results XML Format is a W3C Recommendation whose name is self-explanatory: it describes a standard XML format for returning the results of a SPARQL query. No matter how complex your query is, this format is simple and straightforward—especially when compared with RDF/XML—so that you need very little effort to use those query results with your favorite XML processor. With all of the XML-based tools currently used in publishing and data exchange applications, this makes it easy to let SPARQL queries and results play a role in those applications.

It's easiest to understand the format with a simple example. Figure 7-4 shows DBpedia's response when we ask its SNORQL interface for the English language titles of films directed by Charles Laughton. If you change the "Results:" setting from "Browse" to "XML" and click the Go! button, your browser will request a plain XML version of the result. (The section "SPARQL and Web Application Development" on page 208 shows how programs that you write and utilities such as wget and curl can send queries whose results will be delivered in this XML format.)


SPARQL Explorer for http://dbpedia.org/sparql
SPARQL:
edit
Results:
Browse
Go!
Reset
SPARQL results:
Powered by OpenLink Virtuoso and dbpedia

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT ?film ?title WHERE {
?film <http://dbpedia.org/ontology/director> :Charles_Laughton ;
       rdfs:label ?title .
    FILTER ( lang(?title) = "en" )
}
```

film	title
:The_Night_of_the_Hunter_%28film%29	"The Night of the Hunter (film)"@en
:The_Man_on_the_Eiffel_Tower	"The Man on the Eiffel Tower"@en

Figure 7-4. Using DBpedia's SNORQL interface to list films directed by Charles Laughton

The following shows the XML version of the Charles Laughton query's results:

```
<sparql xmlns="http://www.w3.org/2005/sparql-results#"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.w3.org/2001/sw/DataAccess/rf1/result2.xsd">

  <head>
    <variable name="film"/>
    <variable name="title"/>
  </head>

  <results distinct="false" ordered="true">

    <result>
      <binding name="film">
        <uri>http://dbpedia.org/resource/The_Night_of_the_Hunter_%28film%29</uri>
      </binding>
      <binding name="title">
        <literal xml:lang="en">The Night of the Hunter (film)</literal>
      </binding>
    </result>

    <result>
      <binding name="film">
        <uri>http://dbpedia.org/resource/The_Man_on_the_Eiffel_Tower</uri>
      </binding>
      <binding name="title">
        <literal xml:lang="en">The Man on the Eiffel Tower</literal>
      </binding>
    </result>

  </results>

</sparql>
```

When you compare this XML with the query results in Figure 7-4, its basic structure is pretty clear:

- The document element is called `sparql`, and it has two child elements: `head`, which lists the selected variable names, and `results`, which contains the actual results.
- Each row of the results is stored in a `result` child of the `results` element and has a `binding` child for each bound variable in that row.
- If the value bound to the variable is a URI, it's in a `uri` child of a `binding` element, and if it's a literal, it's in a `literal` element. A language tag for a literal is stored in an `xml:lang` attribute, and a datatype would be stored in a `datatype` attribute.

The `distinct` and `ordered` attributes that DBpedia added to the `results` element above are no longer part of the SPARQL Query Results Format, so don't expect to always find those there.

To make things even easier, the SPARQL Query Results XML Format specification links to an XSLT stylesheet at *http://www.w3.org/TR/rdf-sparql-XMLres/result-to-html.xsl* that converts XML in this format to HTML. Fuseki includes another one in its `pages-update` subdirectory, and you can download the one that SNORQL uses from *http://dbpedia.org/snorql/xml-to-html.xsl*. Using any of these as a starting point, you can create a stylesheet pretty quickly that creates HTML with the look and feel that you want, or even creates a different kind of XML. For example, I modified the Fuseki one to convert the query results to DocBook XML in order to create the query result tables shown in Chapter 6.

The query results XML format spec includes links to RELAX NG and W3C schemas of the result format.

The W3C specification "Serializing SPARQL Query Results in JSON" describes how SPARQL processors can return JSON instead of XML. (Keep in mind that this is a Working Group Note and not a Recommendation and therefore not an official standard like the SPARQL Query Results XML Format document. In standards-speak, the JSON document is not "normative," but a more official version of a Query Results JSON Format specification is underway as part of the SPARQL 1.1 effort.) Here's the result of the query above in JSON:

```
{ "head": { "link": [], "vars": ["film", "title"] },
 "results":
  {"distinct": false, "ordered": true,
   "bindings": [

    {"film":
     { "type": "uri",
       "value": "http://dbpedia.org/resource/The_Night_of_the_Hunter_%28film%29"},
     "title":
      { "type": "literal", "xml:lang": "en",
        "value": "The Night of the Hunter (film)"
      }
    },

    {"film":
     { "type": "uri",
       "value": "http://dbpedia.org/resource/The_Man_on_the_Eiffel_Tower"
     },
     "title":
      { "type": "literal", "xml:lang": "en",
         "value": "The Man on the Eiffel Tower"
      }
    }

   ]
 }
}
```

As you can see, the general structure of JSON SPARQL query results corresponds roughly to the XML version, although not every substructure is labeled the way they are when XML start- and end-tags are used to delimit them. For example, instead of naming individual `binding` objects as members of the JSON `bindings` collection object, this `bindings` object has an array of objects that each associate a `type`/`value` pair with a variable name from the query. In this case, the variable names are "film" and "title".

If you can access this data with JavaScript, you can access and iterate through the query results with all the standard JavaScript control structures. Libraries to read and write JSON are available in most other popular programming languages as well; ex363.py and ex364.cgi showed how Python scripts can take advantage of this.

SPARQL Processors

A SPARQL processor can be a standalone program, a built-in feature of a triplestore, or part of a middleware layer that lets you send SPARQL queries to a dataset that couldn't accept them otherwise.

Standalone Processors

The ARQ processor is handy when you're learning the SPARQL language, because as soon as you unzip it you can give it a file of data and a file with a query and then immediately see the results of running that query against that data. There are no setup or configuration steps necessary to get a server up and running.

The extra steps necessary to set up a server that can accept SPARQL requests and return the results are often worth it when you're assembling a more complex application, but before discussing them, it's worth saying a little more about how ARQ can be used in an application. First, because ARQ is part of the open source Jena project, you can use its libraries and source code to integrate it as the query engine component of a larger JVM-based project, just as the Jena Fuseki and Joseki SPARQL servers do. Without any coding or compiling, you can still use the standalone ARQ binary that you used in this book.

As you'll see if you run it with only the switch `--help` as a parameter, you can tell it to deliver your query result in SPARQL Query Results XML Format, JSON, and comma- and tab-separated output formats. We've also seen how ARQ can both read remote RDF data and query remote SPARQL endpoints, so an application can consist of a simple shell script or batch file that uses ARQ to query some data, saves the results, and then calls other processes to perform additional steps with that data. These steps can use any tools you want: scripts in your favorite development language, XSLT processors, and even spreadsheet programs, which can read comma- and tab-separated value files.

This approach may not result in the fastest, most robust and scalable application, but it's great for prototyping when you're evaluating data you have to work with and what kinds of things you can do with it.

Triplestore SPARQL Support

Instead of querying a set of data that is stored in a single file and gets completely read into memory every time you want to to query it, which is what ARQ does, you're more likely to be querying a larger set of data that is better off stored with a data management system that indexes it for efficient retrieval. This is why, instead of using a standalone SPARQL processor such as ARQ, you'll usually find yourself sending SPARQL queries to a triplestore that happens to have SPARQL support as one of its features.

A triplestore with no SPARQL support is like a relational database manager with no SQL support, which is why the data storage and query language parts of your application are rarely separate choices to make. A triplestore may offer an API in Java, C, or another language to let other processes communicate with it efficiently, but for standards support and quick development, it should support not only the SPARQL query and update languages, but also the "SPARQL Protocol for RDF" and "SPARQL Query Results XML Format" standards.

The triplestore's documentation should explain how to use it to create your own SPARQL endpoints on your own computers using your own domain names as the endpoint URIs. For example, several of this chapter's scripts sent queries to the SPARQL endpoints at *http://data.linkedmdb.org/sparql* and *http://dbpedia.org/sparql*; if you've installed a triplestore on the mysupercompany.com system at your employer "My Super Company," then a proper triplestore would let you create an endpoint with a URI that you choose such as *http://mysupercompany.com/my/endpoint* or some close equivalent.

Middleware SPARQL Support

If you have data in a relational database and want to make it available for others to query, of course SQL is a standard query language, but I know of no standard for passing SQL queries across a network to a relational database management system (RDBMS) and receiving the answer sets. Many developers are finding that the easiest way to make relational data available is to install a middleware layer that accepts SPARQL queries, translates them to SQL, queries the relational data, and returns the data using the SPARQL Query Results XML Format, as shown in Figure 7-5.

If you've followed along with the examples in this chapter, then you've already used one of these combinations of an RDBMS and SPARQL middleware without knowing it: the Linked Moved Database stores its data using the MySQL database and uses the D2RQ server as a middleware layer. D2RQ is free, and once you give it read access to a relational database, it can read the database schema and generate the files it needs to map SPARQL queries to SQL for that database. (These include the resource and property names that you'll use in your queries, and can be customized.) If the same D2RQ server has this kind of access to multiple databases, a single SPARQL query can ask for data from multiple databases at once, which is not possible with a standard SQL query.

The Free University of Berlin team that developed D2RQ came up with their own ontology for mapping between relational schemas and RDF vocabularies. As of this writing, the W3C is developing a standardized mapping to improve consistency between the use of different relational systems in different semantic web application environments.

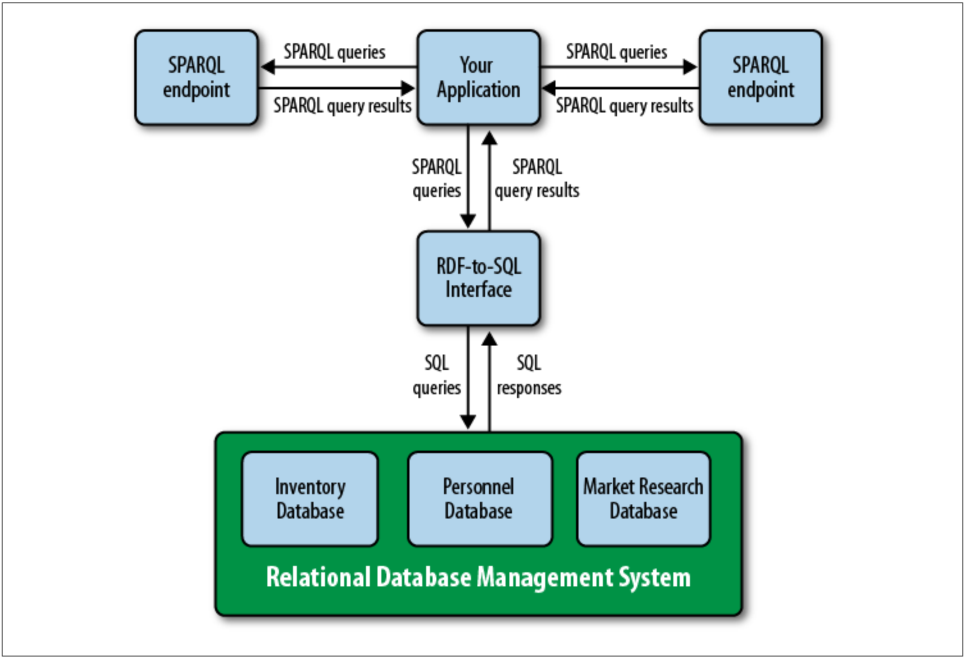


Figure 7-5. Relational database SPARQL endpoint middleware lets your application communicate with relational databases as if they were SPARQL endpoints

The Oracle Corporation's most well-known products are relational database managers, and plenty of semantic web applications have middleware like D2RQ serving up triples created from the relational data in these products. The separate Oracle Database Semantic Technologies product, however, lets you use Oracle Database 11g as a native triplestore, complete with SPARQL support. (Using it requires use of the Oracle Spatial add-on, which optimizes handling of large amounts of location data.)

Other RDF middleware applications include TopQuadrant's TopBraid Live, OpenLink Software's Virtuoso Sponger, and the Triplr project. These offer dynamic creation and integration of RDF triples from sources such as relational databases, spreadsheets, HTML, and other formats. They also typically coordinate multiple different sources in these formats to appear as a single source to someone or something sending a SPARQL query. Each offers a SPARQL endpoint, and more advanced ones let you configure the URI to use as an endpoint just like the better triplestores do.

Public Endpoints, Private Endpoints

In "Linked Data" on page 41 we learned that the Linked Open Data movement is making data from sources around the world available to the public for you to query with SPARQL and use in your applications. When you set up a SPARQL endpoint, though, you don't have to share it with the whole world. Just as your company may have an intranet of web pages that are accessible to staff members logged in to the company system but are inaccessible to people on the public web, everything described in this chapter can be used to build applications behind the firewall. (In fact, you may be able to use the same servers used for your intranet web pages and let them take care of security and data access the same way they already do for those web pages.)

The ability of RDF-based middleware to offer query access to multiple different data sources as if they were one is making this technology increasingly popular for dynamic integration of multiple datasets from different silos. This costs far less than a typical data warehousing project, and it's much more agile, because adding and removing data sources is so much easier.

An easy, low-cost way to mix and match data from different silos, whether this data is stored as triples or not, opens up a lot of new possibilities in any enterprise, especially when you can access that data using a standardized query language supported by both free and commercial software. When you can drive those queries with user-friendly applications developed using the techniques described in this chapter, the possibilities are pretty inspiring.

Summary

In this chapter, we learned:

- How to send a SPARQL query to an endpoint from popular scripting languages and from a web form.
- How the SPARQL Query Results XML Format standard provides an easy way for XML-aware programs to get at query results, and how a JSON equivalent is often available.
- The role that standalone, triplestore, and middleware SPARQL processors can play in making RDF and non-RDF data available to an application that uses SPARQL queries.

Glossary

binding
A pairing between a SPARQL variable and an RDF term. In practical terms, it's a variable that has had a value assigned.

bnode
See **blank node**.

blank node
A subject or object in an RDF graph that has no identity. These are typically used to group together other values. For example, an address book entry may have an email address of "jsmith@example.com", a phone number of 943-234-9664, and an address whose value is a blank node that has its own values: one for a street address, one for a city name, one for a postal code, and so forth. The resource that has these property values is represented by a prefixed name with an underscore prefix (for example, `_:xyz`) or as a pair of square braces (`[]`).

See also **graph**, **prefixed name**.

cast
To convert a piece of data from one datatype to another—for example, converting the string "123" to the integer 123 or "2011-10-14T13:19:00"^^xsd:dateTime to "2011-10-14T13:19:00"^^xsd:string. "Cast" is a common programming term and not specific to SPARQL.

default graph
The triples in an RDF dataset that don't belong to a named graph.

Dublin Core
A popular vocabulary providing a basic set of metadata terms such as `title`, `creator`, and `date`. Many specialized metadata vocabularies are usually based on Dublin Core.

See also **vocabulary**.

endpoint
See **SPARQL endpoint**.

entailment
If A entails B, and A is true, then we know that B is true. If A is a complicated set of facts, it can be very handy to have technology such as an RDFS- or OWL-aware SPARQL processor to help you discover whether B is true.

See also **SPARQL processor**.

FOAF
The Friend of a Friend (FOAF) vocabulary lets you describe facts about a person such as his or her name, home page, work place, and job title. The general idea is to provide the foundation for a distributed, RDF-based social networking system, but the FOAF vocabulary identifies such basic facts about people that it gets used in a wide variety of applications.

See also **vocabulary**.

graph
In RDF, a set of triples. While it is not unusual to feed triples to a utility that creates a graphical representation of them, the term comes from the computer science sense of

the term as a data structure that is like a tree structure but lets any node connect to any other instead of being hierarchical.

graph pattern
: A set of triple patterns between curly braces that specifies the set of triples that a SPARQL processor should retrieve from a dataset.

  See also **triple pattern**, **SPARQL processor**.

inferencing
: Deriving additional facts from existing information. In a semantic web application, this often means creating new triples based on logic applied to existing ones; RDFS and OWL provide additional possibilities.

  See also **OWL**.

IRI
: Internationalized Resource Identifier: a URI that allows a wider choice of characters, making it “internationalized.”

Linked Data
: A set of best practices for connecting related data on the web for use by applications. Because these best practices recommend the use of URIs and standardized data formats, data that follows these practices is easier to use with semantic web technology.

literal
: A value, as opposed to a URI, which is a name for something. A literal may have a datatype or a spoken language tag associated with it, but not both. A simple literal is a literal with no language tag or datatype.

  See also **node**, **URI**.

local name
: A prefixed name without its prefix. For example, in `dc:title`, the local name is `title`.

  See also **prefixed name**.

N3
: A non-XML RDF serialization format developed by Tim Berners-Lee. Turtle is a simplified version of N3.

  See also **serialization**, **Turtle**.

N-Triples
: A very simple RDF serialization format that shows complete URIs with no abbreviation and a triple on each line. Often used as a graph dump format.

  See also **serialization**.

named graph
: A set of triples, typically within a larger collection of them, that can be referenced with a particular name. The name is a URI.

node
: A subject or object in an RDF graph. The three kinds of nodes are literals, IRIs, and blank nodes.

  See also **literal**, **IRI**, **blank node**.

object
: The third part of an RDF triple. You can think of it as an attribute value. An object can be a literal, a URI, or a blank node. If it’s a URI, it’s easier to link it up to other triples that have this URI as a subject in order to describe that resource. In a triple expressing the fact “the book with ISBN 006251587X has a title of ‘Weaving the Web’,” the object would be the string “Weaving the Web”.

  See also **subject**, **predicate**, **literal**, **blank node**.

ontology
: This term can mean different things to different people, especially philosophers, but in the semantic web world, ontologies are formal definitions of vocabularies that allow you to define classes of resources, resource properties, and relationships between resource class members.

OWL
: A W3C standard that builds on RDFS to let you define ontologies.

  See also **serialization**, **ontology**, **schema**.

predicate
: The second part of an RDF triple, also known as a property. You can think of it as an attribute name. The predicate must be a URI in order to identify the predicate’s namespace; otherwise, for a predicate like

"title," we wouldn't know whether it referred to the title of a work, a job title, or something else.

See also **subject**, **object**.

property
See **predicate**.

prefixed name
A name with a prefix to indicate the name's namespace. For example, assuming that the prefix `dc` has been declared to represent the Dublin Core namespace represented by the URI *http://purl.org/dc/elements/1.1/*, the prefixed name `dc:title` refers to the term "title" in the Dublin Core namespace and not the same term from the vCard business card namespace. These are sometimes referred to as qnames, which is strictly correct only in an XML context because of slightly different rules about how they're formed.

qname
See **prefixed name**.

RDF/XML
RDF's original serialization format, based on XML.

See also **serialization**.

RDFa
The use of specialized attributes to embed RDF in HTML and in XML files that were not originally designed to accommodate RDF.

RDFS
See **schema**.

schema
In relational database development, XML, and other areas of information technology, a schema is a set of rules about structure and datatypes used for validation to ensure data quality and more efficient systems. In the semantic web world, the RDF Schema (RDFS) specification lets you specify classes, properties, and metadata about those classes and properties. These serve as metadata to let you infer new facts about your data, not as validation rules to indicate correct versus incorrect data.

See also **OWL**.

screen scraping
A program that does screen scraping retrieves the HTML of a web page such as an airline flight schedule or a weather report and then extracts the data it needs from those pages based on patterns in the HTML that the screen scraper's developers found. Linked Data principles provide ways to share data on the web that reduce the need for screen scraping.

See also **Linked Data**.

serialization
The syntax for how a set of data is represented in a file. You can think of this as meaning "file format," but a set of RDF data may never actually be stored as a file, being passed from one processor to another in a variable instead. The most well-known RDF formats are Turtle, RDF/XML, and N3, although N3 includes features that take it beyond RDF and is used less as Turtle gains in popularity.

See also **Turtle**, **RDF/XML**, **N3**.

simple literal
See **literal**.

SPARQL
The SPARQL Protocol and RDF Query Language, a set of W3C standards for querying and updating data conforming to the RDF model.

SPARQL endpoint
An endpoint is a resource that a process can contact and use as a service; a SPARQL endpoint accepts SPARQL queries and returns the results using the SPARQL Protocol for RDF. One SPARQL service can provide multiple endpoints, each identified by its own URL.

SPARQL engine
See **SPARQL processor**.

SPARQL processor
(Or SPARQL engine) a program that applies a SPARQL query against a dataset and re-

turns the result. This can be a local or remote program.

SPARQL protocol
: The specification for how a program should pass SPARQL queries and updates to a SPARQL query processing service and how that service should return the results.

SQL
: Structured Query Language, an ISO standard query language for relational databases such as MySQL and Oracle. SQL has been around for almost 25 years.

striping
: The nesting of elements in an RDF/XML file that is made possible when the same resource is the object of one triple and the subject of others.

subject
: The first part of an RDF triple. The subject must be a URI to to make it absolutely clear what resource is being described. In a triple expressing the fact "the book with ISBN 006251587X has a title of 'Weaving the Web'," the subject would be a URI that represents the book with ISBN 006251587X.

See also **predicate**, **object**.

triple
: The basic data structure of RDF. The three-part combination of the subject, predicate, and object combine to express a single statement such as "the book with ISBN 006251587X has a title of 'Weaving the Web'."

See also **subject**, **predicate**, **object**.

triple pattern
: Like an RDF triple, but potentially with variables substituted for any parts of the triple.

triplestore
: A specialized database manager designed for storing triples.

Turtle
: An increasingly popular RDF serialization format based on N3. This book's examples of data to query are mostly in Turtle.

See also **serialization**, **N3**.

URI
: Universal Resource Identifier, which encompasses both URLs and URNs. URNs never caught on much, so the terms URI and URL are often used interchangeably, but remember the last letter in each acronym: "URI" is used more often to refer to an identifier, and "URL" to refer to a locator, or address. We use URIs to identify resources and property names in RDF.

URL
: A Uniform Resource Locator, or web address, such as *http://www.learningsparql.com/resources/index.html*. Usually used to refer to something that is actually on the web such as an HTML web page, a graphic image file, or a sound file.

See also **URI**.

URN
: Universal Resource Names, an alternative form of URI that doesn't look like a web address such as *urn:isbn:006251587X*.

variable bindings
: The values assigned to a variable. For example, if your query includes the variable `?postalCode` and a two-row query result has the values "49320" and "22943" assigned to this variable, then those are its two variable bindings.

vCard
: A popular vocabulary for storing business card information such as someone's first name, family name, email address, and job title.

See also **vocabulary**.

vocabulary
: In semantic web development, a set of terms stored using a standard format that people can reuse. RDF Schema and OWL are the key formats for doing this. Vocabularies are often used to name (and possibly specify metadata of) sets of properties to use as predicates in triples. FOAF, Dublin Core, and vCard are popular vocabularies.

See also schema, OWL, Dublin Core, vCard, FOAF.

Index

We’d like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

About the Author

Bob DuCharme (http://www.snee.com/bob) is a solutions architect at TopQuadrant, a provider of software for modeling, developing, and deploying semantic web applications. He came to TopQuadrant from Innodata Isogen, where he did system and architecture analysis and design for a wide range of global publishing clients as well as cochairing the 2008 Linked Data Planet conference in New York City. Earlier in his career, he oversaw SGML and XML development at Moody's Investors Service and then moved on to LexisNexis, where he did data and systems architecture as they made the transition to XML-based systems.

In the XML.com newsletter, editor Kendall Clark once wrote "Does anyone write tech prose as clear as Bob?" Bob is the author of Manning Publications' "XSLT Quickly," Prentice Hall's "XML: The Annotated Specification" and "SGML CD," and McGraw Hill's "Operating Systems Handbook." He's written over 70 pieces for XML.com and has contributed to Dr. Dobb's Journal, IBM developerWorks, Nodalities, DevX, perl.com, XML Magazine, XML Journal, XML Developer, O'Reilly Books' "XML Hacks," and Prentice Hall's "XML Handbook." Bob received his BA in Religion from Columbia University and his Master's in Computer Science from New York University. He lives in Charlottesville, Virginia, with his wife Jennifer and their daughters Madeline and Alice.

Colophon

The animal on the cover of *Learning SPARQL* is an anglerfish.

The cover image is from *Johnson's Natural History*. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.