Querying and Updating with SPARQL 1.1

2nd Edition

Learning SPARQL

O'REILLY®

Bob DuCharme

SECOND EDITION

Learning SPARQL

Querying and Updating with SPARQL 1.1

Bob DuCharme

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Learning SPARQL, Second Edition
by Bob DuCharme

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

Editors: Simon St. Laurent and Meghan Blanchette
Production Editor: Kristen Borg
Proofreader: Amanda Kersey
Indexer: Bob DuCharme
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

August 2013: Second Edition.

Revision History for the Second Edition:
2013-06-27 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449371432* for release details.

ISBN: 978-1-449-37143-2

[LSI]

1372271958

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 that use this data model make it possible to expose diverse sets of data (including, as we'll see, relational databases) with a common, standardized interface. Accessing this data doesn't require learning new APIs because both open source and commercial software (including Oracle 11g and IBM's DB2) are available with SPARQL support that lets you take advantage of these data sources. Because of this data and tool availability, SPARQL has let people access a wide variety of public data and has provided easier integration of data silos within many enterprises.

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 that retrieved data. Once you can do this, you can take advantage of the extensive choice of tools and application libraries that use SPARQL to retrieve, update, and mix and match the huge amount of RDF-accessible data out there.

1.1 Alert

The W3C promoted the SPARQL 1.0 specifications into Recommendations, or official standards, in January of 2008. The following year the SPARQL Working Group began work on SPARQL 1.1, and this larger set of specifications became Recommendations in March of 2013. SPARQL 1.1 added new features such as new functions to call, greater control over variables, and the ability to update data.

While 1.1 was widely supported by the time it reached Recommendation status, there are still some triplestores whose SPARQL engines have not yet caught up, so this book's discussions of new 1.1 features are highlighted with "1.1 Alert" boxes like this to help you plan around the use of software that might be a little behind. The free software described in this book is completely up to date with SPARQL 1.1.

Organization of This Book

You don't have to read this book cover-to-cover. After you read Chapter 1, feel free to skip around, although it might be easier to follow the later chapters if you begin by reading at least through Chapter 5.

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, *Query Efficiency and Debugging*
: Things to keep in mind that can help your queries run more efficiently as you work with growing volumes of data

Chapter 8, *Working with SPARQL Query Result Formats*
: How your applications can take advantage of the XML, JSON, CSV, and TSV formats defined by the W3C for SPARQL processors to return query results

Chapter 9, *RDF Schema, OWL, and Inferencing*
: How SPARQL can take advantage of the metadata that RDF Schemas, OWL ontologies, and SPARQL rules can add to your data

Chapter 10, *Building Applications with SPARQL*
: Different roles that SPARQL can play in applications that you develop

Chapter 11, *A SPARQL Cookbook*
: A set of SPARQL queries and update requests that can be useful in a wide variety of situations

Glossary
: A glossary of terms and acronyms used when discussing SPARQL and RDF technology

You'll find an index at the back of the book to help you quickly locate explanations for SPARQL and RDF keywords and concepts. The index also lets you find where in the book each sample file is used.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, 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, datatypes, 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 shown in the same monospace 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, this book has 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.

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

You'll 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.

This book is here to help you get your job done. In general, if this book includes code examples, you may use the code 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*, 2nd edition, by Bob DuCharme (O'Reilly). Copyright 2013 O'Reilly Media, 978-1-449-37143-2."

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*.

Safari® Books Online

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://oreil.ly/learn-sparql-2e*.

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 first edition, 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 the second edition, I received many great suggestions from Rob Vesse, Gary King, Matthew Gibson, and Christine Connors; Andy also reviewed some of the new material on its way into the book.

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, Kristen Borg, Amanda Kersey, 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, JSON, spreadsheets, and other formats as RDF so that you can issue SPARQL queries against data in these formats—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 Framework"), 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 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 dataset 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 following ex003.rq file 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 lowercase or in mixed case.

In a set of data triples or a set of 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.

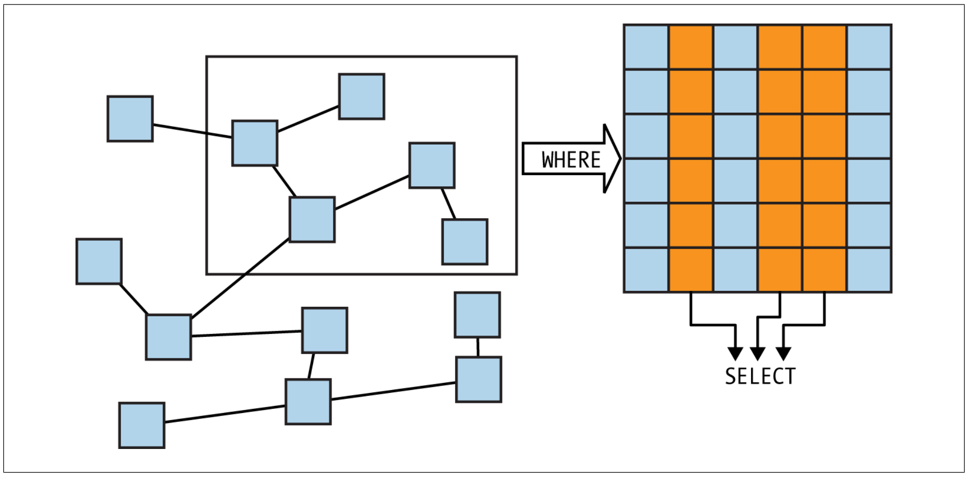


Figure 1-1. WHERE specifies data to pull out; SELECT picks which data to display

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 both local and remote data. (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. ARQ is part of the Apache Jena framework, so to get it, follow the Downloads link from ARQ's homepage at *http://jena.apache.org/documentation/query* and download the binary file whose name has the format `apache-jena-*.zip`. Unzipping this will create a subdirectory with a name similar to the ZIP file name; this is your Jena home directory. Windows users will find `arq.bat` and `sparql.bat` scripts in a `bat` subdirectory of the home directory, and users with Linux-based systems will find `arq` and `sparql` shell scripts in the home directory's `bin` subdirectory. (The former of each pair enables the use of ARQ extensions unless you tell it otherwise. Although I don't use the extensions much, I tend to use that script simply because its name is shorter.)

On either a Windows or Linux-based system, add that directory to your path, create an environment variable called `JENA_HOME` that stores the name of the Jena home directory, and you're all set to use ARQ. On either type of system, you can then 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
```

Running either ARQ script with a single parameter of `--help` lists all the other command-line parameters that you can use with it.

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 that uses full URIs. When you do use a full URI, enclose it in angle brackets to show the processor that it's a URI.
- Whitespace doesn't affect SPARQL syntax. The new query has carriage returns separating the triple pattern's three parts and 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 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 dataset, 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 and a `?propertyValue` variable in the object 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.

When working with RDF, 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.

When referring to a triple in the middle of a sentence, like in the first sentence of the above paragraph, I usually wrap it in curly braces to show that the three pieces go together.

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 query's 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 second 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 the variable that the query's SELECT clause is asking for.

As it turns out, two triples in ex012.ttl have `d: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 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.

Let's say that my phone showed me that someone at "(229) 276-5135" had called me and I used the same ex008.rq query about that number that I used before—but this time, I queried the more detailed ex012.ttl data instead. The result 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 meets the four conditions listed in the graph pattern. Even though resource `ab:i8301` meets 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 meets all four conditions because no one with an `ab:firstName` of "Craig" and an `ab:lastName` of "Ellis" has an `ab:homeTel` value. 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."

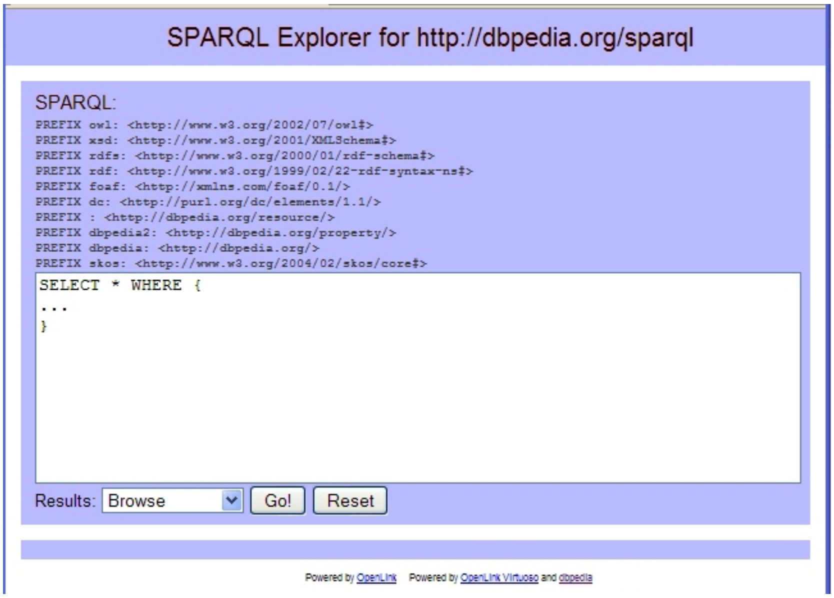
Without the OPTIONAL keyword, a SPARQL processor will only return data for a graph pattern if it can match every single triple pattern in that graph pattern.

Querying a Public Data Source

Querying data on your own hard drive is useful, but the real fun of SPARQL begins 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.

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`. I found plenty of data there, so I knew that this was the right URI to represent him in queries. (The browser was actually 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.) This URI will represent him just like *http://learningsparql.com/ns/data#i8301* (or its shorter, prefixed name version, *d:i8301*) represents Craig Ellis in ex012.ttl.


SPARQL Explorer for http://dbpedia.org/sparql
SPARQL:
Results: Browse
Go!
Reset
Powered by OpenLink 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 * WHERE {
...
}
```

Figure 1-2. DBpedia's SNORQL web form

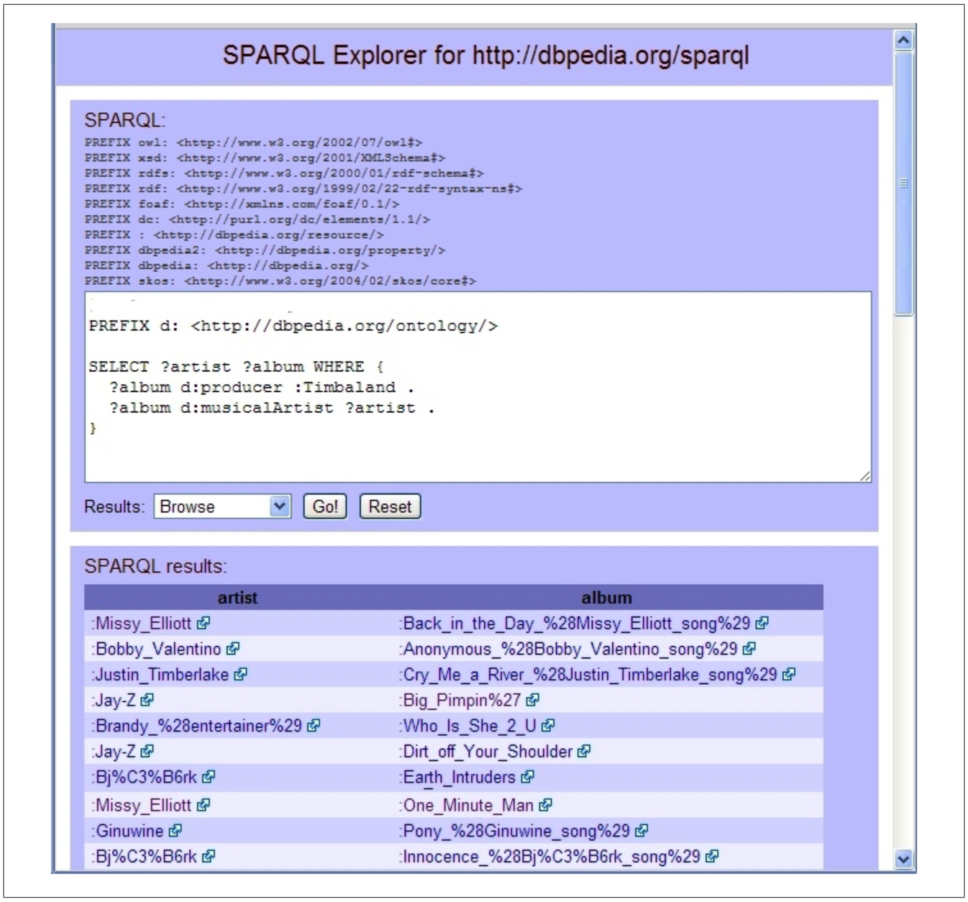
I now see on the upper half of the SNORQL query in Figure 1-2 that *http://dbpedia.org/resource/* is already declared with a prefix of just ":", so I know that I can refer to the producer as `:Timbaland` in my query.

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 it 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.



```
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 .
}
```

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

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.

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, the potential role of inferencing, and the technology's roots in the semantic web world, 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)

The SPARQL query language is 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.

The flexibility of the RDF data model means that it's being used more and more with projects that have nothing to do with the "semantic web" other than their use of technology that uses these standards—that's why you'll often see references to "semantic web technology."

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 hypertext system 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 his 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 symmetric (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!" They also let the server say "OK, here you go!" or "Sorry, I don't know about that resource." We'll learn more about HTTP in "SPARQL and HTTP" on page 295.
- 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 a specification for Universal Resource Names, or URNs. A 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 IRI is the most inclusive 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 his 1999 book 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 know 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 the attribute setting `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 Framework (RDF)

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

- 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 that used different syntaxes to represent the triples. (Although they may be files sitting on disks, they may also be generated dynamically, like so many HTML pages are.) 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 such as 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 (three years before 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 in 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 expresses 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 understand the same triples expressed with 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 when processing 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. There were 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 problem was the 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.

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 ," as he put it) 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 about him and his book 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 whitespace 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 used 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 the triplestore serve as a SPARQL endpoint?
- 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    false .
d:item342 dm:costPerItem 3.50 .
```

Assignment of datatypes is pretty straightforward in RDF/XML: store the URI 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="http://www.w3.org/2001/XMLSchema#date">2011-02-14</dm:shipped>
    <dm:quantity
     rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">4</dm:quantity>
    <dm:invoiced
     rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">false</dm:invoiced>
    <dm:costPerItem
     rdf:datatype="http://www.w3.org/2001/XMLSchema#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 in 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 particular 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 this:

```
# 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 15 `rdfs:label` values for the resource *http://dbpedia.org/resource/Switzerland*. The following shows triples that assign 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 RDF 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 even more `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 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 8, 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 literal objects, a string or other typed value.

Wait—"nearly" all of the nodes? What purpose could a nameless node serve? Let's look at an example. First, we have of a set of triples with no blank nodes, and then we'll see 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.

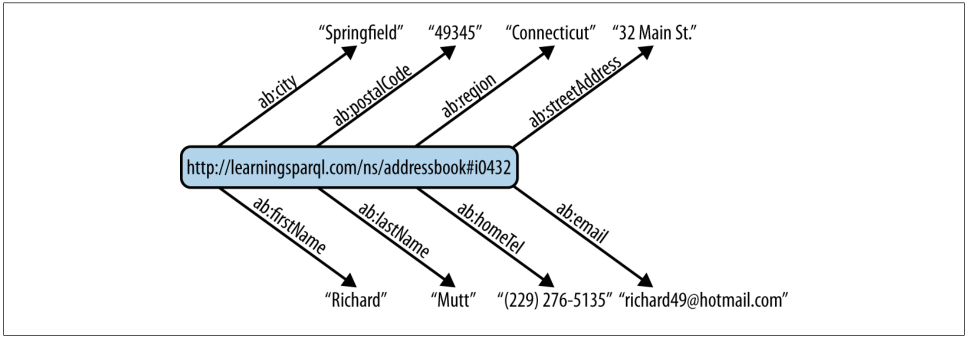


Figure 2-1. Graph of the ab:i0432 address book entry

The following version has a new predicate, `ab: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:

```
# 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 the names of all the other subjects, predicates, and objects in the ex041.ttl dataset, the `b1` after the `_:` means nothing to it, 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 by an RDF-based application, but that application is responsible for maintaining all the same connections to and from each blank node.

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

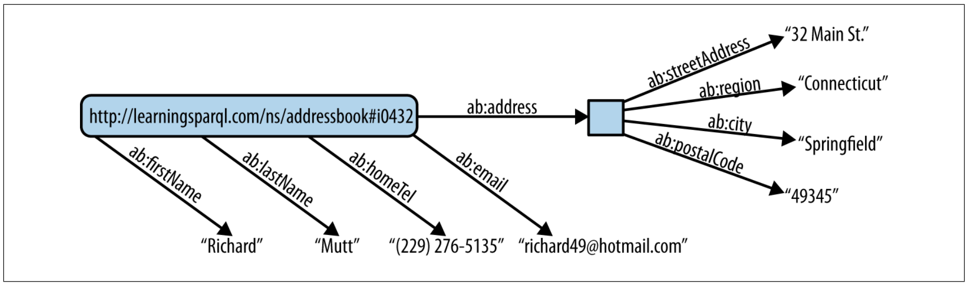


Figure 2-2. Using a blank node to group together postal address data

In the example, the `_:b1` blank node is the object of one triple and the subject of several others. This is common in RDF and SPARQL because they 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 look up Richard's street address by asking for the `ab:streetAddress` value of the `ab:address` node from the address book entry that has a `ab:firstName` of "Richard" and a `ab:lastName` of "Mutt". The `ab: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 `ab:firstName` of "Richard" and a `ab: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, you can then 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 Chapters 3 and 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 provide 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, when describing the `rdfs:label` and `rdfs:comment` properties, I mentioned the W3C RDF Schema specification. 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 I 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 tell us 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 do for `dc:creator` 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 `ab:playsInstrument` value, then it's a member of class `ab:Musician` and its `ab: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 aforementioned address book data, it will list Richard Mutt, even though the data has no triple saying that he is a member of that class. 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 RDF-based 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 RDF 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. We'll see some examples in Chapter 9.

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 RDF 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, Cindy 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 `ab:i8301`. Now we know who Cindy's spouse and doctor are, even though these facts are not explicitly included in the dataset.

If you have a lot of classes, properties, and relationships among them to define and modify, then instead of dealing directly with the RDF/XML or Turtle syntax of your RDFS or OWL models, it's easier if you use a graphical tool that lets you specify what you want by clicking and dragging and filling out dialog boxes. A well-known open source tool for this is Protégé, developed at the University of Stanford. The most popular commercial tool is TopBraid Composer. (The free version of TopBraid Composer lets you do all the modeling you want, as well as performing SPARQL queries on your data; the commercial versions add features for developing and deploying applications.) Although these tools greatly reduce the time you spend looking at syntax, they both save your work using the standard syntaxes.

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 showed 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—especially if your application may need to scale way up, because these profiles are designed to make it easier to implement large-scale systems for particular domains.

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. We'll learn more about RDFS and OWL in Chapter 9.

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).** While a URI can just be a name and not actually the address of a web page, 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 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. Going beyond this HTML linking element, various RDF vocabularies provide other properties that let you 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 awarding 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 would 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 would 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 wonderful 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. They completed their work (including several new specifications) at the end of 2012, and after a few more review steps, the 1.1 specifications became W3C Recommendations in March of 2013.

The SPARQL Specifications

SPARQL 1.0 had three 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. Chapters1 and 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 Chapter 8.

The SPARQL 1.1 effort revised these three (with no noticeable changes to the Query Results XML Format specification) and added eight new Recommendations:

- The **Overview** document describes the set of 11 Recommendations and the role of each.
- The **Federated Query** 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 105 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 specification.
- The **Service Description** specification describes how a client program can ask a SPARQL engine exactly what features it supports.
- **Query Results JSON Format** specification describes a JSON equivalent of the Query Results XML Format. This is described further in Chapter 8.
- **Query Results CSV and TSV Formats** specification describes the comma-separated and tab-separated value equivalents of the Query Results XML Format. These are also described further in Chapter 8.
- The **Graph Store HTTP Protocol** specification 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." This is described in "SPARQL and HTTP" on page 295 in Chapter 10.

- 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.

You can find the SPARQL specifications (and all other W3C standards and drafts) at *http://www.w3.org/TR/*.

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 48
: 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 55
: In Chapter 1, we started learning 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 59
: 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 61
: SPARQL offers some simple ways to ask for a set of triples and additional triples that may be connected to them.

"Eliminating Redundant Output" on page 69
: 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 72
: 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 75
: Specifying boolean conditions, which may include regular expressions, lets you focus your result set even more.

"Retrieving a Specific Number of Results" on page 78
: If your query might retrieve more results than you want, you can limit the returned results to a specific amount.

"Querying Named Graphs" on page 80
: 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 87
: 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 88
: 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 the retrieved data.

"Creating Tables of Values in Your Queries" on page 91
: SPARQL 1.1 lets you create tables of values to use as filter conditions.

"Sorting, Aggregating, Finding the Biggest and Smallest and..." on page 95
: 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 102
: 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 around the world that are accessible to your computer.

"Federated Queries: Searching Multiple Datasets with One Query" on page 105
: 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 8 in Chapter 1, we saw this query, which asks who in the address book data has the phone number (229) 276-5135:

```
# 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: “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.

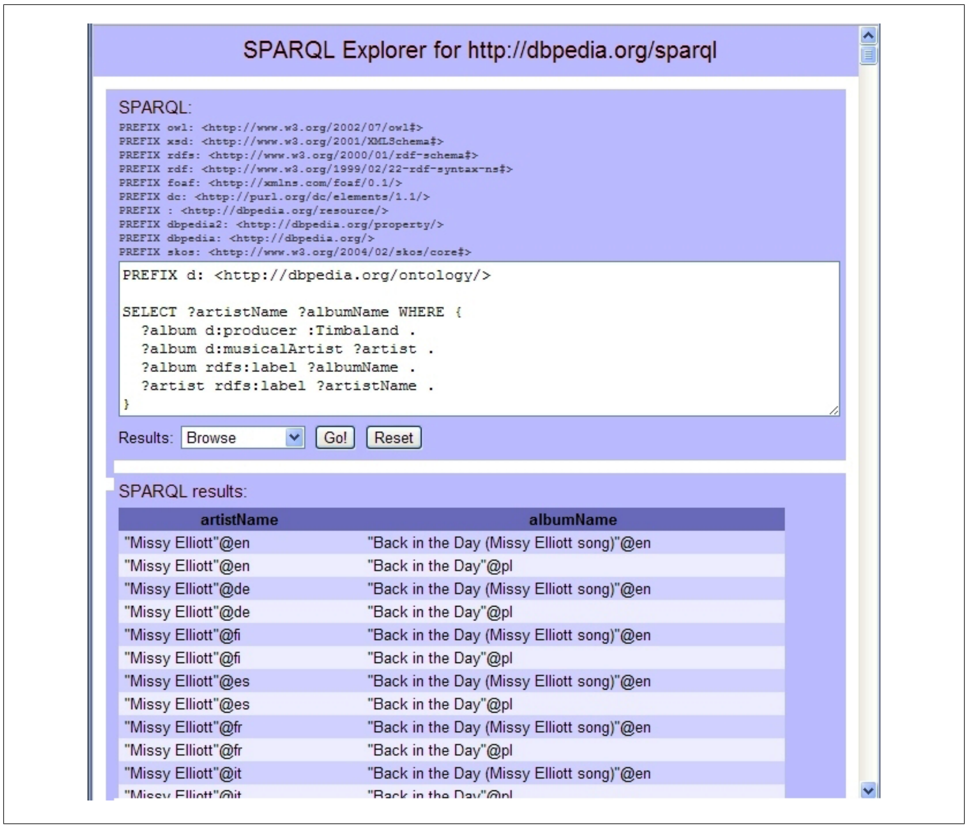
Using the Labels Provided by DBpedia

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 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 has multiple `rdfs:label` values with different language tags assigned to each—for example "en" for English and "de" (Deutsch) 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.



```
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 .
}
```

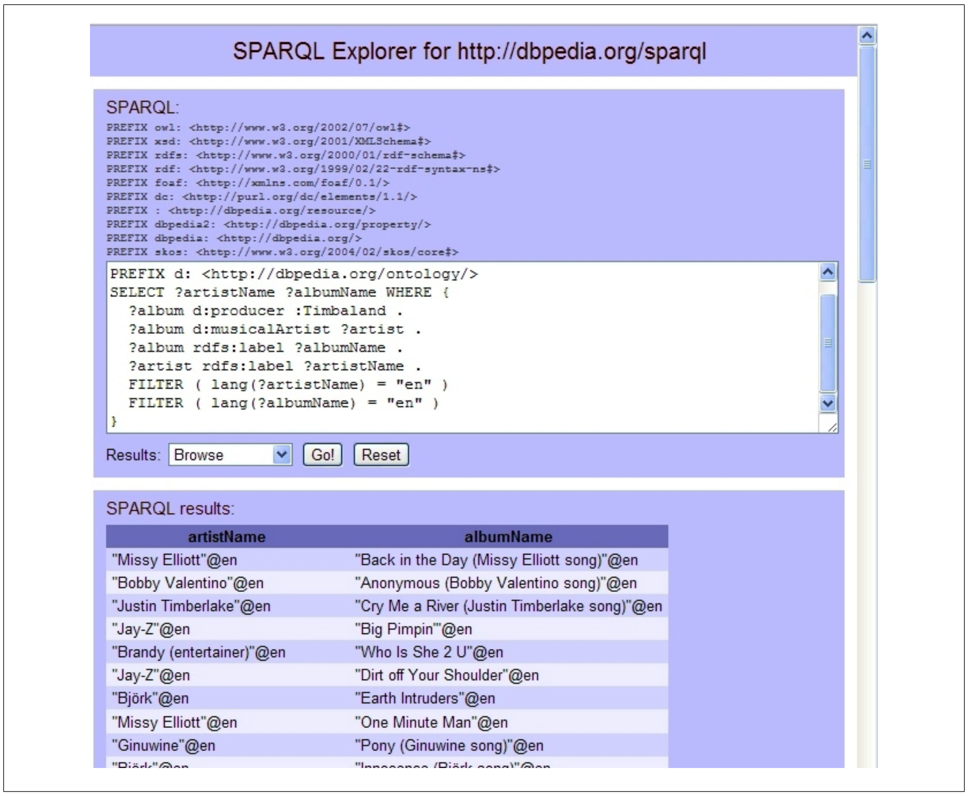
artistName	albumName
"Missy Elliott"@en	"Back in the Day (Missy Elliott song)"@en
"Missy Elliott"@en	"Back in the Day"@pl
"Missy Elliott"@de	"Back in the Day (Missy Elliott song)"@en
"Missy Elliott"@de	"Back in the Day"@pl
"Missy Elliott"@fi	"Back in the Day (Missy Elliott song)"@en
"Missy Elliott"@fi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

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

The FILTER keyword, which we'll learn more about in "FILTERing Data Based on Conditions" on page 75, lets 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" )

}
```

Figure 3-2 shows the result.


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" )
}
```

Results: Browse

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 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 very clear, 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 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 ex052.rq does with ?property in the example 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 version.

Data That Might Not Be There

Let's augment our address book dataset 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 puts it, as a graph pattern. This 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 pattern 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 make 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" and not "Richard" to show up as Mr. Mutt's first name.

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 finds 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 `?first` and `?last` values and that last triple pattern is optional, it's going to output the third line of data that follows:

```
------------------------
| 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.

When querying large datasets, overuse of OPTIONAL graph patterns can slow down your queries. See "OPTIONAL Is Very Optional" on page 219 in Chapter 7 for more on this.

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 55:

```
# 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 you 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 50 we saw the use of a FILTER to retrieve only 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 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, FILTER NOT EXISTS, is a FILTER condition that returns a boolean `true` if the specified graph pattern does not exist. If the following query 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 .
  FILTER 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 FILTER 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 this were all stored 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, 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 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 were 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 different places assembled by people unaware of one another's 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, things, and relationships in your data: because it makes it easier for people to connect that data to other data. (If the data doesn'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 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.

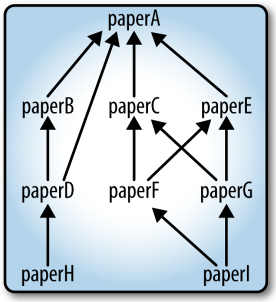


Figure 3-3. Which papers cite which papers in ex074.ttl data

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 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, 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 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 more complex query using the UNION keyword (which we'll learn about in "Combining Different Search Conditions" on page 72) 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 query 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) 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 |
| :paperD |
| :paperH |
| :paperC |
| :paperB |
-----------
```

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, using a property path to ask for papers that are exactly three links away (that is, asking for papers that cited papers that cited papers that cited paper A):

```
# 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 . }
```

This 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.

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 |
-----------
```

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—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 query 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 FILTER condition removes `:paperF` from the results because we don't need the output telling us that `:paperF` is one of the papers that cite papers cited by `:paperF`.) The answer is paper G, which makes sense when you look at the diagram in Figure 3-3:

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

It’s listed twice because it cites two papers cited by `:paperF`.

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 dataset, 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 we have with several 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#takingCourse>  |
| <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 with Figure 1-1 near the beginning of Chapter 1.

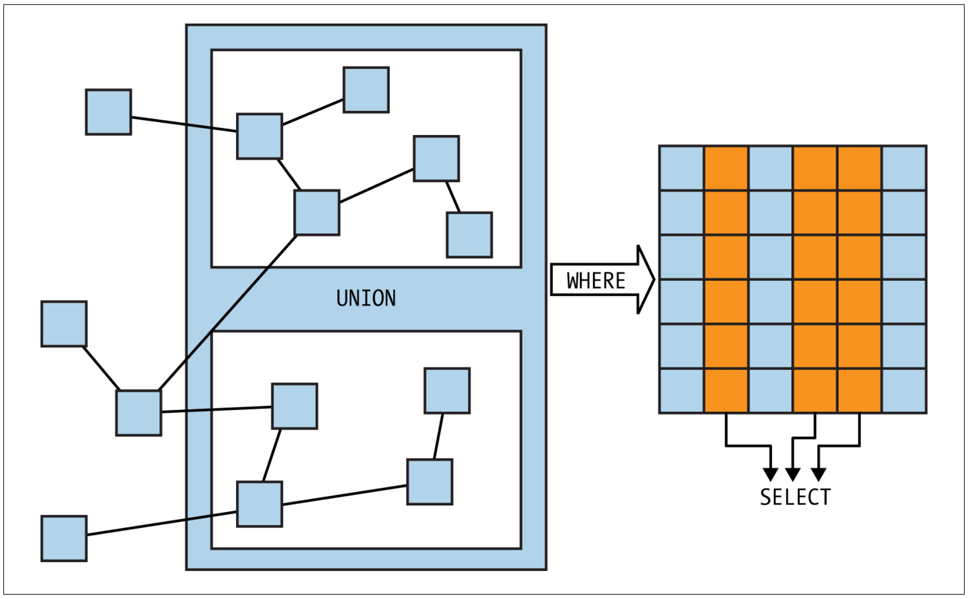


Figure 3-4. The UNION keyword lets WHERE clause grab two sets of data

For example, with our ex069.ttl data that lists 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, in this case, 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 in the output.

The results show who plays these instruments. For each person who plays the sax or trumpet, it lists all the instruments he plays, 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 graph 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, which we'll learn about in "Querying Named Graphs" on page 80) 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, with the "i" for "insensitive" showing that the query doesn't care whether it's in uppercase 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 output's 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 reads in the input data, it maps 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 50, 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 with its SNORQL query form, 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 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 95, 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 three of the six 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 triples out of a larger collection. For example, the following query 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 input, we get only the one value that the query asks for:

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

When using LIMIT and OFFSET together, don't expect much consistency in the results if you don't use ORDER BY with them.

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 the section "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 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 currently 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.

Earlier, 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.

"Node Type Conversion Functions" on page 153 in Chapter 5 demonstrates the use of the BASE keyword, which gives you more control over how a query processor resolves relative URIs.

The next 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. That's why the

query's comment describes it as unnecessary to the query—I just added it 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 in 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 even 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 } }
}
```

Earlier, 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

Queries inside of queries are known as subqueries. This SPARQL feature 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 0.2 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, when I stored the calculated value in the `?total` variable in ex139.rq, I had to move that part 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 . }
  }
}
```

A more common 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)
}
```

Creating Tables of Values in Your Queries

SPARQL's VALUES keyword lets you create tables of values, giving you new options when filtering query results.

> **1.1 Alert**
>
> The VALUES keyword was new for SPARQL 1.1.

The following query ignores any input you pass to it (make sure to pass some anyway if you're using command-line ARQ, which complains if you don't include a `--data` parameter) and demonstrates how you can create a table of values. This example populates the table with prefixed names and literal values, but you can use any kinds of RDF values you want:

```
# filename: ex492.rq

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

SELECT *
WHERE { }
   VALUES (?color ?direction) {
   ( dm:red  "north" )
   ( dm:blue  "west" )
}
```

Here's the result:

```
-----------------------
| color   | direction |
=======================
| dm:red  | "north"   |
| dm:blue | "west"    |
-----------------------
```

This result isn't particularly exciting, but it shows how simple it is to create a two-dimensional table in a SPARQL query. To see what VALUES can add to our queries, we'll use an expense report dataset similar to the one 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 "2011-10-14" ;
         e:amount 6.53 .

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

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

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

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

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

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

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

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

As a baseline before we try the VALUES keyword, we'll start with a simple query that asks for the values of all the dataset's properties without using the VALUES keyword:

```
# filename: ex494.rq

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

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

When run with the dataset above, this query lists all the description, date, and amount values:

```
---------------------------------------
| description | date         | amount |
=======================================
| "dinner"    | "2011-10-16" | 25.05  |
| "lunch"     | "2011-10-16" | 10.00  |
| "breakfast" | "2011-10-16" | 6.65   |
| "dinner"    | "2011-10-15" | 31.45  |
| "lunch"     | "2011-10-15" | 9.45   |
| "breakfast" | "2011-10-15" | 4.32   |
| "dinner"    | "2011-10-14" | 28.30  |
| "lunch"     | "2011-10-14" | 11.13  |
| "breakfast" | "2011-10-14" | 6.53   |
---------------------------------------
```

This next version of the query adds a VALUES clause saying that we're only interested in results that have "lunch" or "dinner" in the `?description` value:

```
# filename: ex496.rq

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

SELECT ?description ?date ?amount
WHERE
{
  ?meal e:description ?description ;
        e:date ?date ;
        e:amount ?amount .
  VALUES ?description { "lunch" "dinner" }
}
```

In this case, the VALUES data structure being created is one dimensional, not two; this is still a step up from the BIND keyword's ability to only assign a single value to a variable at a time.

With the same meal expense data, this new query's output is similar to the output of the one above without the "breakfast" result rows:

```
---------------------------------------
| description | date         | amount |
=======================================
| "lunch"     | "2011-10-16" | 10.00  |
| "lunch"     | "2011-10-15" | 9.45   |
| "lunch"     | "2011-10-14" | 11.13  |
| "dinner"    | "2011-10-16" | 25.05  |
| "dinner"    | "2011-10-15" | 31.45  |
| "dinner"    | "2011-10-14" | 28.30  |
---------------------------------------
```

This query's VALUES clause could go after the SELECT clause's closing curly brace, instead of before it, and it wouldn't affect the results. (This won't always be the case when using the VALUES clause in GROUP BY and federated queries.)

This next query of the same data creates a two-dimensional table to use for filtering output results:

```
# filename: ex498.rq

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

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

  VALUES (?date ?description) {
         ("2011-10-15" "lunch")
         ("2011-10-16" "dinner")
  }

}
```

After retrieving all the meal data, this query only passes along the results that have either a `?date` value of "2011-10-15" and a `?description` value of "lunch" or a `?date` value of "2011-10-16" and a `?description` value of "dinner":

```
---------------------------------------
| description | date         | amount |
=======================================
| "lunch"     | "2011-10-15" | 9.45   |
| "dinner"    | "2011-10-16" | 25.05  |
---------------------------------------
```

When you use VALUES to create a data table, you don't have to assign a value to every position. The UNDEF keyword acts as a wildcard, accepting any value that may come up there. The following variation on the above query asks for any result rows with "lunch" as the `?description` value, regardless of the `?date` value, and also for any result rows with a `?date` value of "2011-10-16", regardless of the `?description` value:

```
# filename: ex500.rq

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

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

  VALUES (?date ?description) {
         (UNDEF "lunch")
         ("2011-10-16" UNDEF)
  }

}
```

The output of this query has more rows than the previous query:

```
---------------------------------------
| description | date         | amount |
=======================================
| "lunch"     | "2011-10-16" | 10.00  |
| "lunch"     | "2011-10-15" | 9.45   |
| "lunch"     | "2011-10-14" | 11.13  |
| "dinner"    | "2011-10-16" | 25.05  |
| "lunch"     | "2011-10-16" | 10.00  |
| "breakfast" | "2011-10-16" | 6.65   |
---------------------------------------
```

When you saw the descriptions of what each of these queries did, it may have occurred to you that most of these query conditions could have been specified without the VALUES keyword—for example, with a FILTER IN clause in the ex496.rq query, although that would only work to replace a one-dimensional VALUES setting. That's true, but I was using a small amount of data to demonstrate different ways to use the new keyword. When you work with larger amounts of data and especially with more complex filtering conditions, VALUES offers an extra layer of result filtering that can give you more control over your final search results with very little extra code in your query.

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 the expanded version of the expense report data that we saw in the last section:

```
# filename: ex145.ttl

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

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

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

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

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

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

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

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

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

d:m40400 e:description "dinner" ;
         e:date "2011-10-16" ;
         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" | "2011-10-15" | 4.32   |
| "breakfast" | "2011-10-14" | 6.53   |
| "breakfast" | "2011-10-16" | 6.65   |
| "lunch"     | "2011-10-15" | 9.45   |
| "lunch"     | "2011-10-16" | 10.00  |
| "lunch"     | "2011-10-14" | 11.13  |
| "dinner"    | "2011-10-16" | 25.05  |
| "dinner"    | "2011-10-14" | 28.30  |
| "dinner"    | "2011-10-15" | 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 treats it as an integer or a decimal number, depending on whether or not the figure includes 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 example 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 description values listed alphabetically and the amounts sorted in descending order within each meal category:

```
---------------------------------------
| description | date         | amount |
=======================================
| "breakfast" | "2011-10-16" | 6.65   |
| "breakfast" | "2011-10-14" | 6.53   |
| "breakfast" | "2011-10-15" | 4.32   |
| "dinner"    | "2011-10-15" | 31.45  |
| "dinner"    | "2011-10-14" | 28.30  |
| "dinner"    | "2011-10-16" | 25.05  |
| "lunch"     | "2011-10-14" | 11.13  |
| "lunch"     | "2011-10-16" | 10.00  |
| "lunch"     | "2011-10-15" | 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 78) to only retrieve the first value. For example, the ex148.rq query 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"    | "2011-10-15" | 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 query 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 header and the border characters that ARQ adds for display in its default output format) 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. 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 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 so that it can run the query and then send back 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 was a new feature in SPARQL 1.1.

The SERVICE keyword can pass a graph pattern or an entire query to the endpoint. The following shows an example of sending a whole query:

```
# 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 query 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, 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 and minister is actually an entire paragraph.

Usually, instead of using SERVICE to pass an entire query to the remote service, queries just send a graph pattern indicating the triples that they're interested in, and then they let the outer query's SELECT (or CONSTRUCT or other query form) indicate which values they're interested in. The following query returns the same data as the previous one:

```
# filename: ex539.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>
  { <http://dbpedia.org/resource/Joseph_Hocking> ?p ?o . }
}
```

This next query sends a similar request about the same British novelist to a German 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://wifo5-04.informatik.uni-mannheim.de/gutendata/resource/people/>

SELECT ?p ?o
WHERE
{
  SERVICE <http://wifo5-04.informatik.uni-mannheim.de/gutendata/sparql>
  { 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 ability to access his work by following the retrieved data gets even more interesting when you use the same technique with more well-known authors whose works are also in the public domain—if you're interested in literature, it's a fascinating corner of the Linked Data cloud.

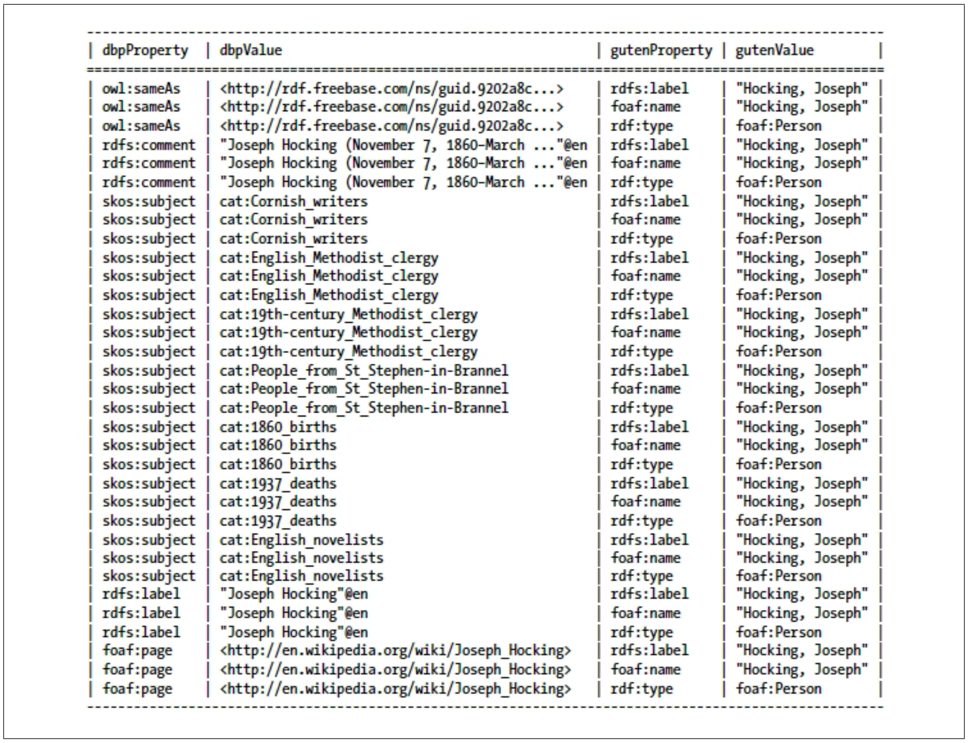
There are many data sources out there offering interesting sets of triples for you to query, but remember that a remote service doesn't have to be very remote, and it doesn't even have to store triples. The open source D2RQ interface lets you use SPARQL to query relational databases, and it's fairly 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 RDF 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

A SERVICE keyword lets you put a subquery inside of another 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 87 for more on subqueries.) The following combines the two examples that demonstrated the SERVICE keyword in "Querying a Remote SPARQL Service" on page 102, 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://wifo5-04.informatik.uni-mannheim.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>
  {
    <http://dbpedia.org/resource/Joseph_Hocking> ?dbpProperty ?dbpValue .
  }

  SERVICE <http://wifo5-04.informatik.uni-mannheim.de/gutendata/sparql>
  {
    gp:Hocking_Joseph ?gutenProperty ?gutenValue .
  }
}
```

The results, shown in Figure 3-5 (again, with data trimmed at the `...` parts), might not be quite what you expected.



```
----------------------------------------------------------------------------------------------------------
| 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

Remember, queries use the SERVICE keyword to retrieve data from SPARQL endpoints.

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 later 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 your 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 the VALUES keyword lets you create tables of values to use as filter conditions
- 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 110
: 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 111
: 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 115
: 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 RDF technology.

"Converting Data" on page 120
: 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 123
: 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 133
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 1 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 SELECTed variable name, 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 Turtle text, 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://wifo5-04.informatik.uni-mannheim.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>
  {
    <http://dbpedia.org/resource/Joseph_Hocking> ?dbpProperty ?dbpValue .
  }

  SERVICE <http://wifo5-04.informatik.uni-mannheim.de/gutendata/sparql>
  {
    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 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://wifo5-04.informatik.uni-mannheim.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 can also 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 you're creating a copy of something you 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.

If you don't have a graph pattern after your CONSTRUCT clause, the SPARQL processor assumes that you meant the same one as the one shown in your WHERE clause. This can save you some typing when you're simply copying triples. For example, the following query would work identically to the previous one:

```
# filename: ex540.rq

CONSTRUCT
FROM <http://rdf.freebase.com/rdf/en.joseph_hocking>
WHERE
{ ?s ?p ?o }
```

Creating New Data

As the above ex178.rq query 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 its area code, then 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 returned all the triples, including the `ab:homeTel` value, even if the `{?person ab:homeTel ?phone}` triple pattern wasn't there. The WHERE clause 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 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 use a CONSTRUCT clause to 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 creates two triples about people having 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 applications that use RDF technology, 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 new 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 36, 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. We'll learn more about this in Chapter 9, but for now, 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 good practice because it makes it 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 `owl: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 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 your input data isn't using any particular schema and you're just replacing one set of predicates with another. Ideally, though, a schema 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 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 55 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 the query above, 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 RDF software 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://wifo5-04.informatik.uni-mannheim.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>
  {
    <http://dbpedia.org/resource/Joseph_Hocking> ?dbpProperty ?dbpValue .
  }

  SERVICE <http://wifo5-04.informatik.uni-mannheim.de/gutendata/sparql>
  {
    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 triples about *http://learningsparql.com/ns/data#HockingJoseph* created from the two sources:

```
@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://wifo5-04.informatik.uni-mannheim.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://wifo5-04.informatik.uni-mannheim.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 the 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 data structures and types 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.

RDF-based 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 to 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 75 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 three-step process of reading the dataset 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 an RDF-based 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 48 for more on this.)

Running this query against the ex198.ttl dataset, we're not just asking whether there's a bad `dm:location` value somewhere. We're asking which resources have a problem and what that problem 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 .
```

The output tells us that resource `d:item693` (the Heidelberg purchase) has the named problem.

> As we'll see in "Using Existing SPARQL Rules Vocabularies" on page 131, a properly modeled vocabulary for problem identification declares a class and related properties for the potential problems. 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 property values. 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 which resource has 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 the SPARQL engine finds triples 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 one another'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, two vocabularies that I can use are 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 the generated error message triples.

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.

We saw earlier that SPARQL isn't only for querying data stored as RDF. (We'll see more about this in "Middleware SPARQL Support" on page 293 in Chapter 10.) 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 this dataset, we get 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 on DESCRIBE, "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 respond to a DESCRIBE request with 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 135
: RDF supports a defined set of types as well as customized types, and your SPARQL queries can work with both.

"Functions" on page 145
: 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 majority of the functions defined by the SPARQL specification.

"Extension Functions" on page 182
: 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, the identifier of an ISO standard, 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 off the part about specifying datatypes for elements and attributes 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,” and often abbreviated as “XSD”—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 derived one 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`, but you use it for values that do not include a time value—for example, “2014-10-13” instead of “2014-10-13T12:15:00”.

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="http://www.w3.org/2001/XMLSchema#date">2011-02-14</dm:shipped>
    <dm:quantity
     rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">4</dm:quantity>
    <dm:invoiced
     rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">false</dm:invoiced>
    <dm:costPerItem
     rdf:datatype="http://www.w3.org/2001/XMLSchema#decimal">3.50</dm:costPerItem>
  </rdf:Description>

</rdf:RDF>
```

To review a few things that 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:datatype` attribute.
- The name of the datatype can be a full URI, as shown in the `dm:shipped` value in the ex033.ttl example.
- It can also be a prefixed name: 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 above.
- Just like any other prefixes in RDF triples, the `xsd:` prefix on a datatype must 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 prior examples:

```
# 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    false .
d:item342 dm:costPerItem 3.50 .
```

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 the following 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.

When run with ARQ, 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" . }
```

ARQ treats them both as equal to "two" because, when working with data completely stored in memory, the Jena framework underpinning ARQ infers that an untyped literal and a literal with the same value typed as `xsd:string` are equal. (It does this based on the W3C's "RDF Semantics" Recommendation, which provides some of the foundation for the eventual RDF 1.1 upgrade, in which it will become official on a more widespread basis to treat `xsd:string` as a default datatype.) You may find that other SPARQL processors, such as the ones used with Fuseki and Sesame, do not make this inference when querying ex217.ttl with ex220.rq and return the single value that exactly matches the one shown in the triple pattern.

This next query returns the same two results when run with ARQ, but may return the single value that exactly matches the one shown in the triple pattern with other current SPARQL processors:

```
# 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 . }
```

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 has 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 164.)

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 those 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 153:

```
# 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 96.)

Comparing Values and Doing Arithmetic

"FILTERing Data Based on Conditions" on page 75 in Chapter 3 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 14, 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"                             |
-------------------------------------------------------------------
```

In "Combining Values and Assigning Values to Variables" on page 88, we saw some arithmetic being performed in the following query, which does 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 whitespace 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, 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 that manipulated input values and returned new values.

Because of this, most of the original 1.0 functions 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 of them, 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 makes no 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. The next example 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, RDF 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 whitespace you want before and after parentheses and the commas that separate parameters. This makes complex expressions easier to read.

After the query binds each dm:location value to the ?locationValue variable in the first triple pattern, it 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 .
```

I created a URI for Heidelberg in the *http://learningsparql.com/ns/data#* namespace and not the *http://dbpedia.org/resource/* namespace because new URIs should use namespaces that you have control over, not namespaces built around someone else's domain.

As you learn about more functions to use in SPARQL queries, remember that you can use any of them—including additional 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 55 in Chapter 3 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 expressions qualify as URIs, literals, numeric literals, or blank nodes. 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 123 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 146 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()`** function 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.

Two more useful functions for checking on variables are **`bound()`** and **`sameTerm()`**. The `bound()` function 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 59. The `sameTerm()` function returns a boolean value telling us whether the two terms are the same or not. We'll see some examples of its use in Chapter 7.

The `sameTerm()` function returns a true value if the two arguments are the *same term*, not the same value. So, while `sameTerm(4,4)` will return true, `sameTerm(4,4.0)` will not.

Th `sameTerm()` function's greatest usefulness is in comparing two variables that are storing URIs. A call like `sameTerm(?var1,?var2)` is essentially the same as the expression `(?var1 = ?var2)`. If it does the same thing as the more obvious expression, why bother with it? Because with most SPARQL engines, `sameTerm()` is more efficient, so your query will run a little faster.

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 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, xsd:string 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 of 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 146 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 in "String Functions" on page 171. 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 following revision of ex251.rq 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 were 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 for all the data in ex241.ttl:

```
@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          | 1.0               | "1.0E0"^^xsd:float | 1.0E0           |
| 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>`).

The SPARQL 1.1 Query Language specification includes a table that shows which type conversions are always allowed (for example, integer to string), which are never allowed (for example, dateTime to boolean) and which conversions are "dependent on the lexical value." As examples of this last case, the string "4" can be converted to an integer, but the string "four" cannot.

You may find that different SPARQL processors handle examples of this last case differently. For example, while `xsd:decimal(1.0e5)` didn't return anything when the above query was run with ARQ 2.10, with Sesame 2.6.4, it returns 10000.

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 when running query ex251.rq with dataset ex241.ttl, 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 ?boolTest
WHERE
{
  ?s ?p ?o .
  BIND (xsd:string(?o) as ?stringTest)
  BIND (xsd:boolean(?o) as ?boolTest)
}
```

The results show that conversion to a string works for everything but the blank node, and that conversion to boolean is pickier:

```
------------------------------------------------------------------------------------
| o                                     | stringTest                           | boolTest |
====================================================================================
| 3                                     | "3"^^xsd:string                      | true     |
| "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"                                | "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                  | true     |
------------------------------------------------------------------------------------
```

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 171 will be handy.) The `xsd:boolean()` function converts the number 0 to a boolean false and all other numbers, as you can see above, to a boolean true; this is consistent with the behavior of several popular programming languages.

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/demo#* namespace with customized datatypes from the *http://learningsparql.com/ns/units#* namespace. The following query converts them using `STRDT()` to assign the custom datatypes:

```
# filename: ex267.rq

PREFIX im: <http://learningsparql.com/ns/importedData#>
PREFIX dm: <http://learningsparql.com/ns/demo#>
PREFIX u:  <http://learningsparql.com/ns/units#>

CONSTRUCT { ?s dm: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 dm:      <http://learningsparql.com/ns/demo#> .
@prefix im:      <http://learningsparql.com/ns/importedData#> .

d:item2
      dm:amount     "10"^^u:squareMeters .

d:item1
      dm:amount     "14"^^u:liters .
```

```
d:item3
      dm:amount     "30"^^u:centimeters .
```

The CONCAT() function concatenates the ?units value onto a string version of the URI 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 150 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 123 for more on SPARQL rules.)

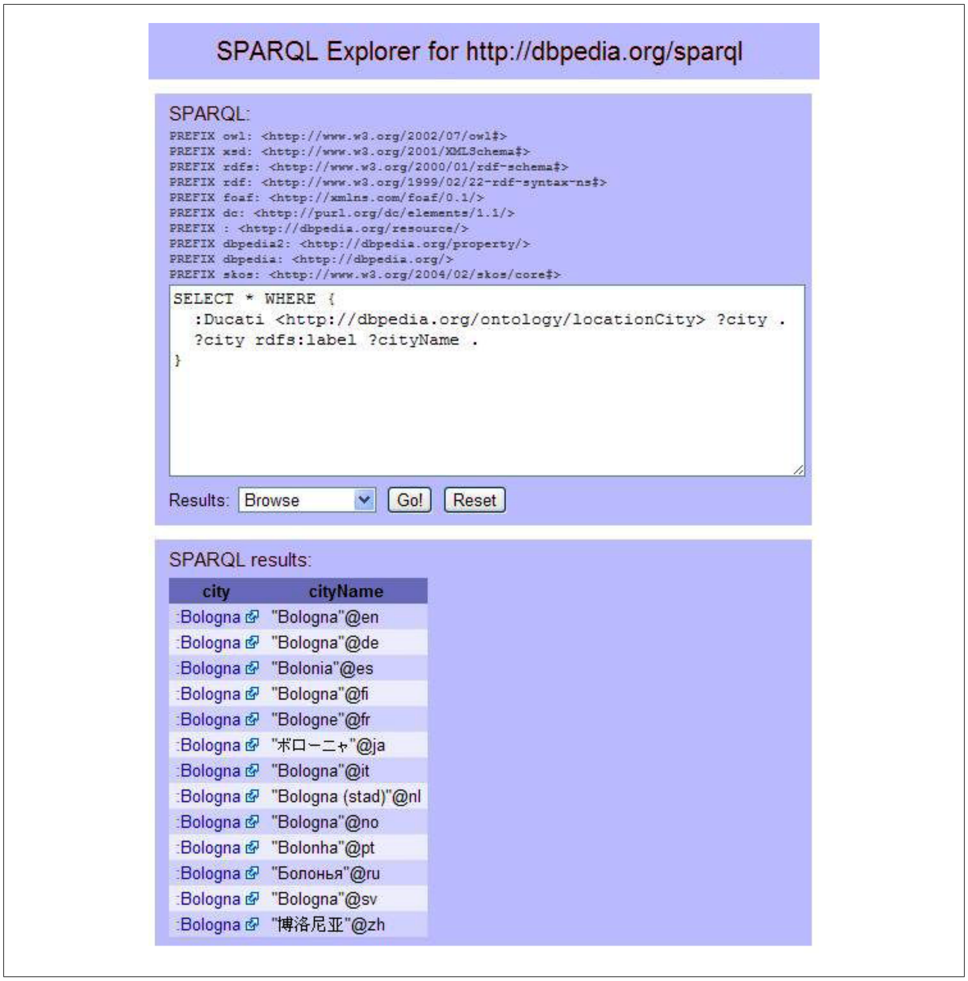
Checking, Adding, and Removing Spoken Language Tags

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. It also shows the 13 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*:


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 {
  :Ducati <http://dbpedia.org/ontology/locationCity> ?city .
  ?city rdfs:label ?cityName .
}
```

SPARQL results:

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

```
# 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 .}
```

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 153 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 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 uppercase 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 lack 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 two 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. Each returns a boolean value that tells you whether or not 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 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` value 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. It's 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 three 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 in Chapter 1 we saw that the following query retrieves triples where the string "yahoo", in any combination of uppercase 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 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.learnsparql.com%2Fcgi%2Ffunc1%26color%3Dred" |
| "My%20String"                                                   |
-------------------------------------------------------------------
```

This is especially useful, as we'll see in Chapter 10, 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 100 in Chapter 3 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, 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 14, 2011, at 12 noon five time zones 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 time zone and fractions of a second are not required, so that if you wanted to say that a meeting begins or a flight leaves at `"2011-10-14 T12:00:00"^^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 time zone, so because `meeting1` takes place at 12:30 five time zones 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 time zone 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 time zone 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 time zone values out of the `d:meeting1` time. 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, which we've seen before in Chapters 1 and 4 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, 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 built into 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 was 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 let you update their data. You may still see this with older triplestores.

In this chapter, we'll learn about:

"Getting Started with Fuseki" on page 186
: Just enough about Fuseki to try out all the key parts of SPARQL Update

"Adding Data to a Dataset" on page 188
: How to add triples specified in your query and triples from remote sources to the dataset's default graph

"Deleting Data" on page 194
: How to delete data from the default graph

"Changing Existing Data" on page 196
: How to replace existing triples in the dataset's default graph

"Named Graphs" on page 201
: 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 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; in "SPARQL and HTTP" on page 295, we'll learn about using its HTTP interface.

As of this writing, the latest official release of Fuseki is release 0.2.6. 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 this chapter describes.

To download Fuseki, follow the Downloads link from its home page to get the binary ZIP file whose name has the format jena-fuseki-*-distribution.zip. Once you unzip this, you're ready to go. It includes a shell script called `fuseki-server` that will start it up under Linux or on a Mac and a `fuseki-server.bat` batch file that will do the same under Windows. You can find out about Fuseki's command-line options with the following command:

```
fuseki-server --help
```

Before starting up the Fuseki server, I created a subdirectory of the distribution's root directory and called it `dataDir`. Then, to start up Fuseki as a server on a Windows or Linux machine, I ran the fuseki-server script with these parameters:

```
fuseki-server --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`
: The dataset path name, which must begin with a slash. (This is a Fuseki detail unrelated to the SPARQL spec.)

> The examples in this book always use the dataset named `/myDataset`, and the instructions sometimes have you erase everything stored there before proceeding with the next steps. If you're building an application unrelated to this book's examples, store its data in a different dataset.

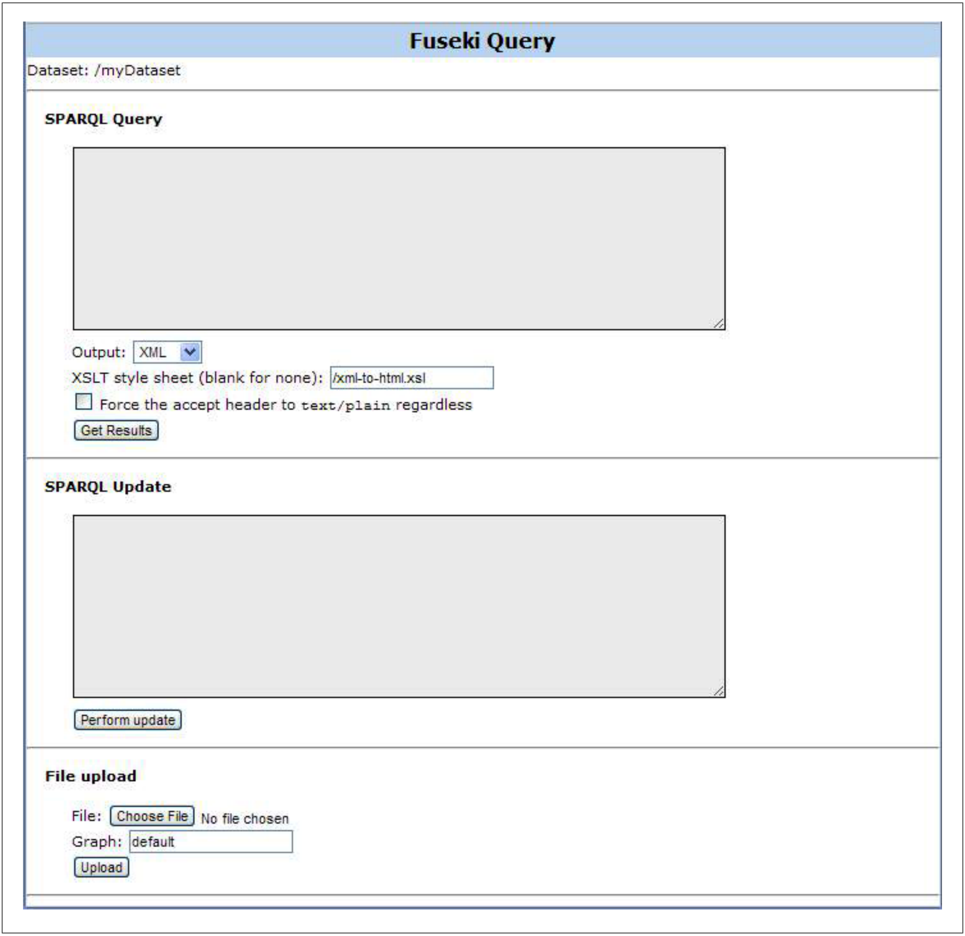
As the server starts up, a few status messages will scroll up in the window where you entered the command to start it. (Later, when you've finished using Fuseki and you're ready to shut it down, press `Ctrl+C` in this window.)

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

SPARQL Update

File upload

File: Choose File No file chosen

Graph: default

Figure 6-1. Fuseki'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 ex012.ttl, 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 (you'll want to set the form's Output field to either XML or Text first), 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"

The Fuseki Query screen shows that output returned as XML gets formatted with an XSLT stylesheet included with Fuseki 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 201 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 ex312.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 creates 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, not triple patterns, between the curly braces with an INSERT DATA operation.

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 previous examples, 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 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 finds 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 exactly 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. This is because (as we’ll see in the section “Named Graphs” on page 201) a request may consist of multiple individual updates—or as the spec calls them, operations.

Paste this into the SPARQL Update panel of the Fuseki Query form and click the Perform update button 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 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 Tim Berners-Lee’s book *Weaving the Web* from the OCLC’s WorldCat enormous collection of data about published works:

```
# filename: ex546.ru

LOAD <http://worldcat.org/oclc/41238513.ttl>
```

After you paste this update request into Fuseki’s SPARQL Update panel and click the Perform update button, the ex311.rq List All Default Graph Triples query will show that you loaded over 100 triples into your dataset.

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: ex547.rq

SELECT * WHERE
{<http://www.worldcat.org/isbn/0062515861> ?p ?o }
```

When you run it, you'll see the predicates and objects from the 6 (at this writing) triples that describe the hardcover version of the book. This includes the predicates and objects from the following two triples, which show:

- That it's the "cloth" edition, the industry term for hardcover
- The fact that this URI represents the same resource as the URN shown

(Remember, URNs are URIs too, but they're rarely used apart from identifying the ISBN numbers of books.)

```
<http://www.worldcat.org/isbn/0062515861>
  <http://schema.org/description>
  "cloth" .

<http://www.worldcat.org/isbn/0062515861>
  <http://www.w3.org/2002/07/owl#sameAs>
  <urn:isbn:0062515861> .
```

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 ex548.ru

PREFIX wci: <http://www.worldcat.org/isbn/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

DELETE DATA
{
  wci:0062515861 <http://schema.org/description> "cloth" ;
                 owl:sameAs <urn:isbn:0062515861> .
}
```

After executing it, try the ex547.rq SELECT query, and you'll see two less triples in the result, because you just deleted them.

The following DELETE update request would do the same thing as the DELETE DATA update request in ex548.ru:

```
# filename ex549.ru

PREFIX wci: <http://www.worldcat.org/isbn/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

DELETE
{
  wci:0062515861 <http://schema.org/description> "cloth" ;
                 owl:sameAs <urn:isbn:0062515861> .
}
WHERE {}
```

Like the comparable INSERT DATA and INSERT queries, the DELETE DATA version can be faster if you're working 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 *http://www.worldcat.org/isbn/0062515861* as a subject, 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, you'll see at least three remaining triples that fit this pattern. Then, run this DELETE update request:

```
# filename: ex550.ru

DELETE { <http://www.worldcat.org/isbn/0062515861> ?p ?o }
WHERE  { <http://www.worldcat.org/isbn/0062515861> ?p ?o }
```

Running the ex311.rq List All Default Graph Triples SELECT query again will show that these triples are gone. (You may see at least one triple where this URI is the object, but this query was only deleting those where it was the subject.)

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 ex550.ru update request:

```
# filename: ex551.ru

DELETE WHERE  { <http://www.worldcat.org/isbn/0062515861> ?p ?o }
```

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 201):

```
# 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 preceding INSERT/DELETE update request:

   ```
   # 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 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 only 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 with Fuseki's web-based interface, 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" 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, 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. (On the down side, the extra complexity of SKOS-XL has prevented it from getting much support or traction.)

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>
<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>

s	p	o
<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 about graphs between a client and a SPARQL processor. 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. See "SPARQL and HTTP" on page 295 for more on this specification.

To get a feel for creating, adding to, deleting from, and replacing triples in named 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 graph names 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.

This next update request does the same thing as ex331.ru, but instead of the keyword DATA after INSERT, it has a WHERE clause:

```
# filename: ex543.ru

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

INSERT
{ GRAPH d:g1
  { d:x dm:tag "three" }
}
WHERE {}
```

The syntax difference between ex331.ru and ex543.ru is just like the difference between ex312.ru and ex313.ru, which used these two techniques to insert triples into the default graph instead of a named one. The first uses INSERT DATA because there's no need for a WHERE clause to specify conditions. The second has INSERT without DATA an empty WHERE clause where, if you wanted, you could add the kind of triple patterns that you might put in any other SPARQL query to control the logic of what got created.

After you run ex331.ru or ex543.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 still 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 0.2.6 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 0.2.6 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" .
}
```

You can use the USING and WITH keywords together in the same update, but make sure you understand which parts of your graph each applies to. It's often simpler to just avoid using them together.

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”. Because of the USING keyword, it looks in graph *dg:g2* for these triples. 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 with a subject of *d:w* that you inserted with update request ex343.ru), the USING keyword specifically told the SPARQL processor to look in the *d:g2* graph for triples that matched 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 update request 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#>

INSERT
{ GRAPH d:g4
  { ?s dm:tag "five", "six" . }
}
USING NAMED d:g2
WHERE
{ GRAPH d:g2
  {
    ?s dm:tag  "five" .
    ?s dm:tag  "six" .
  }
}
```

Copying and Moving Entire Graphs

SPARQL Update's COPY and MOVE operations let you copy and move triples between named graphs or between the default graph and a named graph.

We can set up some sample data to see their effects by first running the DROP ALL command shown in ex337.ru in "Dropping Graphs" on page 204, and then running the ex338.ru update that follows it. This inserts two triples in the default graph, two in the named graph `d:g1`, and two in the named graph `d:g2`. After running it, running the ex332.rq List All Triples SELECT query shows the following triples and the graphs they belong to (with prefixes substituted for the original base URIs to more easily fit the output on this page):

```
---------------------------------
| g    | s   | p      | o       |
=================================
|      | d:x | dm:tag | "one"   |
|      | d:x | dm:tag | "two"   |
| d:g1 | d:x | dm:tag | "three" |
| d:g1 | d:x | dm:tag | "four"  |
| d:g2 | d:x | dm:tag | "five"  |
| d:g2 | d:x | dm:tag | "six"   |
---------------------------------
```

The COPY operation copies triples from one graph into another, replacing any existing triples in the destination graph. (To quote the spec, "If the destination graph does not exist, it will be created.") The following update request copies the triples from the default graph to graph `d:g2`:

```
# filename: ex503.ru

PREFIX d: <http://learningsparql.com/ns/data#>
COPY DEFAULT TO d:g2
```

Pretty simple. After running it, running the query that lists all graphs and triples shows that the "five" and "six" triples that were in the `d:g2` graph are no longer there and that the "one" and "two" triples are there and still in the default graph as well:

```
---------------------------------
| g    | s   | p      | o       |
=================================
|      | d:x | dm:tag | "one"   |
|      | d:x | dm:tag | "two"   |
| d:g1 | d:x | dm:tag | "three" |
| d:g1 | d:x | dm:tag | "four"  |
| d:g2 | d:x | dm:tag | "one"   |
| d:g2 | d:x | dm:tag | "two"   |
---------------------------------
```

The MOVE operation moves triples from one graph to another, also replacing existing triples in the destination graph. As with COPY, if the destination graph doesn't exist, it will be created. The following update request moves the triples in graph `d:g2` to graph `d:g1`:

```
# filename: ex505.ru

PREFIX d: <http://learningsparql.com/ns/data#>
MOVE d:g2 TO d:g1
```

When run against the result of the COPY update request, the result shows that there's nothing left in `d:g2` and that `d:g1` has the triples that used to be in `d:g2`:

```
-------------------------------
| g    | s   | p      | o     |
===============================
|      | d:x | dm:tag | "one" |
|      | d:x | dm:tag | "two" |
| d:g1 | d:x | dm:tag | "one" |
| d:g1 | d:x | dm:tag | "two" |
-------------------------------
```

You can see more variations on these COPY and MOVE operations in the SPARQL Working Group's SPARQL 1.1 Test Suite, but the tests are basically different combinations of moving triples between default and named graphs, pre-existing or otherwise.

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 also 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 in the query. While the ex347.ru update request 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"}
```

Replacement of triples in named graphs is a combination of deleting and inserting them, just like 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 predicate but "cinco" as an object. Go ahead and run it, then run the ex332.rq List All Triples query 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 enroll in 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 10.)

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. And, 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 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

Query Efficiency and Debugging

A query asks for a set of information. Sometimes, there are different ways that a query can ask for the same set of information, and some ways are more efficient than others. When you keep in mind the amount of work that each part of your query asks a SPARQL processor to perform (or, in computer science jargon, how “expensive” each part is in processor cycles), it helps you create queries that run faster. Debugging techniques and tools can also help you tune your query as well as fix a query that isn’t doing what you want it to.

In this chapter, we’ll learn about:

“Efficiency Inside the WHERE Clause”
: The WHERE clause is the heart of any query, and the ordering of its components and the choice of functions it calls can speed things up or slow things down.

“Efficiency Outside the WHERE Clause” on page 226
: Once a WHERE clause has returned values from a dataset, there are several things that a query can do with those queries, and some are more expensive than others.

“Debugging” on page 227
: Debugging of SPARQL queries starts with classic techniques that you’d use with any development language and can also take advantage of specialized tool features.

Efficiency Inside the WHERE Clause

Before a SPARQL processor can list, sort, delete, or insert the data described in your query or update request, it usually must first find the data you’re interested in by matching the triple patterns in your query’s WHERE clause against the triples in the dataset that you’re querying. While the order of a graph pattern’s triple patterns should not affect the eventual query results, the ordering can have a big effect on the speed of the query’s execution. As some background for why ordering matters, let’s look at a simplified overview of what the SPARQL processor does as it moves through a WHERE clause’s triple patterns:

1. It looks for triples in the dataset that match the first triple pattern. If it finds none, it gives up the search—unless the triple pattern is in an OPTIONAL clause, in which case the processor keeps trying to match further triple patterns.
2. If it did find at least one triple that matched the triple pattern in step 1, and the triple pattern had any variables, it stores the appropriate values in (or, to use the technical term, binds them to) those variables. BIND statements also assign values to variables.
3. If the next triple pattern in the WHERE clause uses any variables that were bound during a match against an earlier triple pattern, the query processor substitutes those values into those parts of the triple pattern and then goes looking for triples that match that pattern.
4. If it doesn't find any and the triple pattern isn't in an OPTIONAL clause, it gives up. If it does find some, it resumes at step 2 with this triple pattern.

If the query never gave up because of a lack of matches, it repeats the second and third steps until it's matched the full graph pattern of the WHERE clause against the dataset. Then, it applies any FILTER statements, which can use simple or complex expressions to specify triples to filter out of the result set. Finally, it passes along the remaining variable values to the rest of your query outside of the WHERE clause.

Keeping these steps in mind, remember this general principle: a fast query is one that, if there are no matches for the graph pattern, gives up quickly, and if there are matches, quickly returns the variable values that it bound. Let's look at some ways to encourage this.

Reduce the Search Space

Imagine that you've lost your sunglasses in a two-story house that has 10 rooms on each floor. Just before you set off to look through those 20 rooms, your friend coming down the stairs tells you that she just searched every room on the upper floor for her keys and definitely did not see your sunglasses up there. She has reduced your search space, making your job easier by ruling out half of the places that you might have otherwise searched for your sunglasses.

Whether you're searching a relational database for a part supplier name or searching Google for the name of a song you heard on the radio, many of the engineering techniques that went into speeding up your searches were designed around reducing the search space as much as possible as early as possible. The same idea applies to a SPARQL engine's search through a collection of triples.

We'll see this as a theme throughout our discussion of efficient SPARQL queries. We'll start with a simple application of this idea: take advantage of data typing. RDF doesn't require you to declare classes and then identify which resources are members of which classes, but triples that do this can be easier to search quickly.

For example, let's say that you're looking for information about a particular book in a dataset that has information about books, plays, movies, paintings, and music, and the dataset uses DBpedia classes such as `dbpedia-owl:Book` and `dbpedia-owl:Film` to identify the class of each work. If the triple pattern `{?work rdf:type dbpedia-owl:Book}` appears early in your query's graph pattern and later triple patterns use the same `?work` variable, that triple pattern has narrowed the search space for these later triples and set the stage for the search engine's job to go more quickly.

When your data identifies class membership for the resources being described, it can help speed up queries that people make of your data.

OPTIONAL Is Very Optional

The OPTIONAL keyword is handy for retrieving values that may or may not be available without aborting your query's execution, but the fact that an OPTIONAL clause gives the query processor work to do without reducing the search space means that it gives the processor no help in completing its work. Academic papers on SPARQL query optimization agree: OPTIONAL is the guiltiest party in slowing down queries, adding the most complexity to the job that the SPARQL processor must do to find the relevant data and return it. Nested OPTIONAL clauses compound the problem.

Because nonoptional triple patterns can help a SPARQL engine decide whether it's going to find an answer set or not, moving your OPTIONAL clauses after them can mitigate its effects.

Query engines usually optimize for you, so moving parts of your graph pattern earlier or later may have no effect on query performance. Different query processors have different approaches to this. Ask your SPARQL engine's developers about their optimization strategies. Some may consider their approach to be an advantage of their SPARQL processor over others, and may therefore not want to discuss the topic in detail, but for open source SPARQL processors, it's a classic discussion topic.

The best optimization is to just avoid using OPTIONAL whenever possible. In an ASK query, which we first learned about in Chapter 4, an OPTIONAL clause has no effect on the answer—either the required triples are there or not—so it won't do anything but slow the query down. (The combination of OPTIONAL with a FILTER or MINUS clause can affect the result of an ASK query, so this advice only applies to simple uses of OPTIONAL.)

Triple Pattern Order Matters

OPTIONAL clauses are not the only parts of a graph pattern whose order can affect execution time; even the order of simple triple patterns can do so. The fewer triples that a triple pattern matches against, the more it narrows down the search space, and the faster the query processor can finish its job. You know that a triple pattern with three unbound variables will match against all the triples in your datastore, and a triple pattern with no unbound variables will try to match against only one. While it is possible for a triple pattern with only one unbound variable to match more triples than a triple pattern with two, usually the one with more unbound variables will match against more triples because it's more flexible, so it won't narrow the search space as much.

A variable's position in the triple (that is, whether it's in the subject, predicate, or object position) can also provide a clue about the relative number of triples that it may match. Knowing this gives you more opportunities to identify triples that reduce the search space quickly and deserve to be earlier in the query.

For example, a given dataset is more likely to have the same property in the predicate position of a large number of triples than that dataset is to have a particular resource in the subject position of a similarly large number of triples. In a database of 10,000 triples that uses 10 properties to describe each of 1000 resources, each resource will be the subject of an average of 10 triples—in relational terms, you can think of those triples collectively as a record with 10 fields—but if each property is used to describe each resource, then each of the 10 properties will be used as the predicate of 1000 triples. Matching against this data, a triple pattern that only has a specific value in the predicate position and variables in the other two will match against many more triples than a triple pattern that has a specific value in the subject position and variables in the others.

The documentation for the open source C# dotNetRDF library provides some details about how its SPARQL engine's query optimizer reorders triple patterns based on where each has variables. dotNetRDF tries to evaluate triple patterns with higher selectivity earlier to reduce the search space. It does this by ranking the triple patterns in the order shown by the following list, which describes which parts of each triple are have fixed values instead of being variables:

1. Subject-Predicate-Object
2. Subject-Predicate
3. Subject-Object
4. Predicate-Object
5. Subject
6. Predicate
7. Object

This list is for dotNetRDF release 1.0.0 and may change. Remember that it provides general heuristics based on typical patterns and won't necessarily optimize any dataset-query combination that you throw at it; this is why query engines often apply additional optimization techniques such as analysis of data statistics. Still, the list provides a great example of issues to think about when comparing the potential effect of different triple patterns on query execution time.

A triple pattern with a subject, a predicate, an object, and no variables is the most selective of all, because it will only match one triple. The least selective (besides, of course, a triple pattern with three variables) is one that has variables in the subject and predicate position and a specific value in the object position. The ranking of a triple pattern with a known subject at position 5 in the dotNetRDF list above the position of a triple pattern with a known predicate at position 6 confirms the preceding discussion about searching 10,000 triples: a triple pattern with a known subject and a variable in the predicate position usually narrows the search space faster than the reverse.

The discussion above does not take the potential role of named graphs into account, but the same principles would apply: if your query engine knows that it's trying to match triple patterns within specific named graphs, it has a narrower search space and can do its job faster. On the other hand, if you have the GRAPH keyword followed by a variable in place of a graph name, because you want to know which named graph or graphs contain certain triples, you're giving the query engine more work to do.

FILTERs: Where and What

Along with moving triple patterns around in a graph pattern, moving a FILTER statement earlier can also reduce the search space for subsequent triples, as long as all of the variables that it references have been bound before the FILTER statement.

Sometimes, careful use of triple patterns can let you omit a FILTER statement and make a query faster. For example, the following query works, but could use some improvement. When executed against DBpedia's SPARQL endpoint, it successfully lists all films that starred both Al Pacino and Robert De Niro:

```
#filename: ex508.rq

PREFIX db: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?pacinoFilm
WHERE
{
  ?pacinoFilm dbo:starring db:Al_Pacino .
  ?deNiroFilm dbo:starring db:Robert_De_Niro .
  FILTER(?pacinoFilm = ?deNiroFilm)
}
```

The first triple pattern finds all the films that Pacino has starred in, the second finds De Niro's, and the FILTER statement indicates that we're only interested in the values that appeared in both lists.

Let's review what the bolded line in that query does: it looks for *all* triples with a predicate of `dbo:starring` and an object of `db:Robert_De_Niro`. If the goal of the query is to find De Niro films that also starred Al Pacino, and we already know at that point in the query what films Pacino starred in, we should use that information to reduce the search space to check for De Niro films, and we no longer need the FILTER statement:

```
#filename: ex509.rq

PREFIX db: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?pacinoFilm
WHERE
{
  ?pacinoFilm dbo:starring db:Al_Pacino .
  ?pacinoFilm dbo:starring db:Robert_De_Niro .
}
```

This shorter query will get the same result and be faster.

When you do use a FILTER keyword, the work that you ask it to do can also affect query performance. The boolean expression that specifies which triples to pass along can be a very simple one, but it can also take advantage of (and combine) all the functions that your SPARQL processor supports—and some functions ask more of the processor than others.

If we review the first query we saw in Chapter 1 that used the FILTER keyword, it looks pretty simple, asking for all triples but then using the `regex()` function in the FILTER statement to indicate that we only want the triples whose objects have the string "yahoo" in them, regardless of whether it's in uppercase or lowercase:

```
# filename: ex021.rq

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

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

If you've worked with regular expressions before, you know that in addition to a simple string like "yahoo", the `regex()` function can look for patterns. For example, giving this function a second argument of "^\\d{3}\\-\\d{2}\\-\\d{4}$" tells it to look for values in its first argument that fit the pattern of a U.S. Social Security number: three digits followed by a hyphen, two more digits, another hyphen, and then four final digits.

All this power and flexibility, though, comes at a cost. Accounting for all the options that `regex()` can handle while it looks for a match is a lot of work for the processor. If you look over the functions described in "String Functions" on page 171, you'll see some simpler, more specific ones that can be faster than `regex()`.

The following revision of the last query uses two SPARQL 1.1 functions to do the same thing in the FILTER statement: the `CONTAINS()` function looks for "yahoo" in the lowercase version of `?o` returned by the `LCASE()` function:

```
# filename: ex510.rq

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

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

The `CONTAINS()` function, being more limited (or, you could say, more specialized) than the `regex()` function, will probably do its job faster than the `regex()` function.

If we wanted to find Yahoo email addresses instead of just the string "yahoo", another SPARQL 1.1 function can be even more efficient. In this revision of the above queries, SPARQL 1.1's `STRENDS()` function ("string ends") checks whether the value stored in the `?o` variable ends with the string "@yahoo.com":

```
# filename: ex511.rq

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

SELECT *
WHERE
{
  ?s ?p ?o .
  FILTER (STRENDS(?o, "@yahoo.com"))
}
```

We saw earlier that the principle of reducing the search space in order to speed things up applies to relational databases, web searches, and triples. Using `STRENDS()` instead of `CONTAINS()` takes this principle down to the string level: the more focused function only has to check at the end of the string being searched instead of looking all the way through it, so it will know sooner whether the target "@yahoo.com" string is there or not.

Another specialized SPARQL function that can speed things up is `sameTerm()`, which returns a boolean true value if the two arguments passed to it are the same term. With most SPARQL processors, this function is more efficient than using the equals operator.

For example, the following revision of ex508.rq from earlier in this chapter has its original FILTER statement commented out and replaced with a FILTER statement that uses `sameTerm()`, making it more efficient:

```
#filename: ex522.rq

PREFIX db: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?pacinoFilm
WHERE
{
  ?pacinoFilm dbo:starring db:Al_Pacino .
  ?deNiroFilm dbo:starring db:Robert_De_Niro .
#  FILTER(?pacinoFilm = ?deNiroFilm)
  FILTER(sameTerm(?pacinoFilm,?deNiroFilm))
}
```

The ex509.rq revision of ex508.rq is still more efficient than either, but it's worth seeing this example because the practice of comparing two resources to see if they're equal is so common when deciding which data to display or triples to create.

Let's look at another example: in Chapter 4, ex190.rq created triples about who was the aunt of who by checking for the grandparents' female children but filtering out parents, because we don't want to list someone's mother as his or her aunt. The following revision of ex190.rq does the same thing, but comments out the original FILTER statement and adds a new one that uses the `sameTerm()` function to be more efficient:

```
# filename: ex523.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)
  FILTER (!sameTerm(?parent,?aunt))
}
```

Even when running the two variations of this query against the little bit of data in ex187.ttl, using the ARQ SPARQL engine with its `--time` command-line parameter (described in "Debugging" on page 227) shows that the query with `sameTerm()` FILTER statement is usually faster than the original version.

See "Node Type and Datatype Checking Functions" on page 150 for more on the `sameTerm()` function, including a warning about just how similar its two arguments need to be.

Instead of relying on a few general-purpose functions like `regex()`, get to know the full range of functions available for your SPARQL queries. More specialized ones are often more efficient—especially when manipulating strings, because string processing can be expensive. Your research should include the SPARQL extensions in your implementation as well as standard functions, since extensions are often more optimized for a particular triplestore and therefore more efficient. Virtuoso's `bif:contains()` function is one example. (Of course, using extensions reduces the portability of your code.) See Chapter 5 for more about SPARQL functions.

Property Paths Can Be Expensive

In many cases, the property paths feature added to SPARQL 1.1 (see "Searching Further in the Data" on page 61) does for triple patterns what regular expressions do for strings: they give you more sophisticated ways to describe patterns to search for. The wider choice of options in how and where property paths let you search for patterns among your data means that they often increase your query's search space.

For example, the plus sign in the following example, like the plus sign in regular expression syntax, tells the processor that once it has found the specified target, it should keep looking for additional connected ones—in this case, that if it finds papers that cite `:paperA`, it should also look for papers that cite those, and papers that cite those, and so on:

```
# filename: ex078.rq

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

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

SPARQL property path syntax offers several other ways to say "and then keep looking for more." While these ways give you more power to find data that meets certain conditions, their use obviously gives the processor more work to do, because along with the extra searching, the processor must check whether each new triple that it finds is truly new or one that found earlier via a different path. Whether it's successful or not in finding these additional connections, the fact that it has to do this extra looking and checking means that your query's execution will take longer.

Efficiency Outside the WHERE Clause

Once your SPARQL engine has matched a graph pattern against a dataset and bound values to appropriate variables, there are many things that you can do with these values, and some are more expensive than others. The variables you ask for and your options for arranging the returned data can make a difference in total execution time.

Asking the SPARQL processor to sort the returned values with ORDER BY may be asking a lot. The example shown at "Sorting Data" on page 96 sorted nine values, which is very little work, but if your query returns more than a few thousand rows, an ORDER BY instruction (especially one with multiple sort keys) can cost you some time. The DISTINCT keyword also asks the processor to do extra work after executing the WHERE clause's logic—work that requires scanning through the entire answer set.

On the other hand, the LIMIT keyword, as described in "Retrieving a Specific Number of Results" on page 78, usually tells the SPARQL processor to do less work than it might otherwise have, so it can make a query execute faster. For example, if I know that a given SPARQL endpoint has many millions of triples, there's a good chance that many of them specify `rdfs:label` values. Instead of asking to see all of these, if I just want a general sense of what the labels look like (for example, what language or languages they're in, or if they use technical terminology from a specific field such as finance or biology) I might ask to just see 20 of them like this:

```
# filename: ex512.rq

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

SELECT *
WHERE
{ ?s rdfs:label ?label }
LIMIT 20
```

Using ORDER BY with LIMIT, as described in "Finding the Smallest, the Biggest, the Count, the Average..." on page 98, is handy for tasks like finding the three highest values for a given property, but you lose the execution efficiency that a LIMIT clause can add because the processor must retrieve all the relevant results and sort them before giving you the requested amount.

Naming fewer variables in your SELECT statement can also make your query run faster. It may not seem like you're relieving the query processor of much work by asking for the values of two variables instead of five, since all five must be bound as part of the WHERE clause's logic, but in addition to reducing the size of the returned data, this gives the query processor more flexibility in how it optimizes your query.

SELECT * is a nice shorthand when you're first assembling a query, but for a query that will be used as part of a production system, asking the SPARQL processor to return all the bound variables is usually asking more of it than you really need.

The most extreme (and most efficient) application of this idea of asking for fewer variables is to not ask for any of them, which is essentially what an ASK query does. Because an ASK query can only return a boolean value indicating whether or not a graph pattern had a match in the target dataset, it's useful when you only want to know whether certain conditions exist in the data, and it gives a query processor even more optimization options. (And, as mentioned above, don't even bother with OPTIONAL clauses in an ASK query unless it's using FILTER or MINUS in a way that may affect the returned result.)

Debugging

If your query or data has syntax errors, your SPARQL processor should report them to you. When there are no syntax errors, a query that isn't behaving the way you want is returning either wrong data, more data than you expected, or more likely, less data—often, no data at all. What steps can you take to fix these problems?

Even when your query does exactly what you want, if it's taking longer than you'd like, debugging techniques can can help you to identify the parts that are slowing things down and to develop strategies for addressing the issues.

Manual Debugging

Typical good habits from any programming or query language will help you here. Comments added with the # symbol help someone reading a complex query to understand the intended logic. So does the use of extra whitespace to indent and separate blocks of code. This someone might be another person, but it might be you if you have to review some code that you haven't looked at in a while.

One development technique that is especially valuable in SPARQL is incremental coding. If you know all the details of the data that you want to retrieve and you write out the complete query before executing it for the first time, you may end up with no results. It's best to write a little, make sure that it returns something, add a little more, try again, and repeat this pattern until you have what you originally were aiming for.

For example, let's say I wrote out the following query, which is a slight variation on query ex017.rq from "More Realistic Data and Matching on Multiple Triples" on page 8:

```
# filename: ex513.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 .
}
```

To really develop this kind of query, I would start with something closer to this:

```
# filename: ex514.rq

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

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

The WHERE clause only has the first of the triple patterns that I want, and the SELECT clause asks for everything, which in this case is just the values of the `?person` variable. I won't have the final version of the query return the `?person` values because, being URIs, they're less readable than the data associated with them, but for now I only want to make sure that the pattern finds something, and a test with ex012.ttl shows that it does.

Next, I add a second triple pattern, and a test with the same data reveals that it also finds some matches:

```
# filename: ex515.rq

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

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

The ex513.rq example would be my third step. (At this point, it would still have an asterisk in the SELECT statement instead of specific variable names; I would put in the variable names either when I was convinced that the logic all worked correctly or when the number of columns crowded the display.) When that didn't return any data, I would know exactly where the problem was: in that third triple pattern that I had just added. The problem? I forgot to capitalize the "n" in the predicate's property name, and while the data has at least one resource with an `ab:homeTel` value, an `ab:firstName` value, and an `ab:lastName` value, it has none with an `ab:homeTel` value, an `ab:firstName` value, and an `ab:lastname` value, which is what the combination of the three triple patterns in ex513.rq asks for.

This approach can be done retroactively by commenting out code. The following version of ex513.rq has the SELECT list commented out, an asterisk inserted to temporarily replace them, and the last two lines commented out:

```
# filename: ex516.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 .
}
```

This is essentially the same query as ex514.rq, and can be incrementally restored to the full query by removing the hash symbols one at a time until you find out which line is causing any problem. I do this all the time.

If a query retrieves the results you want but does so more slowly than you'd like, commenting out certain parts (for example, an OPTIONAL or ORDER BY clause) can give you clues about whether those parts of the query are slowing it down.

Using temporary SELECT lists on your way to the final query is not just a technique for developing SELECT queries. Using temporary SELECT lists for early drafts of CONSTRUCT, ASK, INSERT, and DELETE requests can help you be sure that your WHERE clause finds the data that you really want it to find—something you want to be especially sure of for DELETE requests.

As you tweak a query to make it more efficient, small changes in execution time will be difficult to detect without help from your tools, and many SPARQL tools have help to offer. For example, adding the `--time` parameter to the ARQ command line adds the execution time figure, measured to the thousandth of a second, to your output. It's how I confirmed that the ex523.rq query earlier in this chapter ran faster that the ex190.rq query that it was based on.

SPARQL Algebra

Some query processors can offer additional clues about how they're executing your query if you learn a little about SPARQL algebra. To help SPARQL implementers, chapter 18 of the W3C SPARQL Query Language 1.1 specification shows how the query syntax gets converted to an abstract syntax that is described using this SPARQL algebra, which represents data structures using parenthesized lists. (The syntax will look familiar to those who've used the programming language LISP or its descendants such as Scheme or Clojure.) A SPARQL processor usually works from this, not from the query that you wrote out, when it executes your query.

A given SPARQL query may have different SPARQL algebra representations, and a single SPARQL algebra expression can be represented with different sets of query syntax.

If you don't plan to implement SPARQL yourself, there's certainly no need to learn all the details of the SPARQL algebra syntax, but just seeing how the parentheses group together the parts of your query can give you a better idea of how it's executing those parts.

For example, here's a query that deliberately ignores some of the advice on query efficiency:

```
# filename: ex517.rq

PREFIX ab: <http://example.org/ab#>

SELECT ?first ?last ?homeTel
WHERE
{
  OPTIONAL { ?s ab:homeTel ?homeTel .}

  ?s ab:firstName ?first .
  FILTER(CONTAINS(?last,"M"))
  ?s ab:lastName ?last .
}
ORDER BY ?last
```

(Mapping of prefixes to full URIs is one of the steps in the conversion of query syntax to SPARQL algebra, so to help the following SPARQL algebra example fit on the page better, the `ab:` prefix in the example above is standing in for the URI *http://example.org/ab#* instead of the longer *http://learningsparql.com/ns/addressbook#* URI that it usually represents in this book.) Here the SPARQL algebra version of this query, as output by ARQ:

```
(project (?first ?last ?homeTel)
  (order (?last)
    (filter (contains ?last "M")
      (join
        (leftjoin
          (table unit)
          (bgp (triple ?s <http://example.org/ab#homeTel> ?homeTel)))
        (bgp
          (triple ?s <http://example.org/ab#firstName> ?first)
          (triple ?s <http://example.org/ab#lastName> ?last)
        )))))
```

Because the query gets executed from the inside out, we can see that the heart of this query has two "basic graph patterns" (labeled bgp) that will get evaluated before the FILTER condition is applied to them, and then the ORDER BY statement will be applied last.

Knowing a little about the query algebra syntax also lets you take advantage of other debugging tools. For example, when you ask ARQ for verbose output by adding the -v switch to its command line, it displays log messages showing parts of the SPARQL algebra version of the query as they're executed. Here are the log messages shown when executing the previous query:

```
16:53:03 INFO  exec :: BGP :: (?s <http://example.org/ab#homeTel> ?homeTel)
16:53:03 INFO  exec :: Reorder :: (?s <http://example.org/ab#homeTel> ?homeTel)
16:53:03 INFO  exec :: BGP :: (?s <http://example.org/ab#firstName> ?first)
               (?s <http://example.org/ab#lastName> ?last)
16:53:03 INFO  exec :: Reorder :: (?s <http://example.org/ab#firstName> ?first)
               (?s <http://example.org/ab#lastName> ?last)
```

The execution of ARQ that created those INFO messages actually included another optional command-line parameter: optimize=off. The default setting for optimization is on, and here are the log messages shown for the same query when run with optimization on:

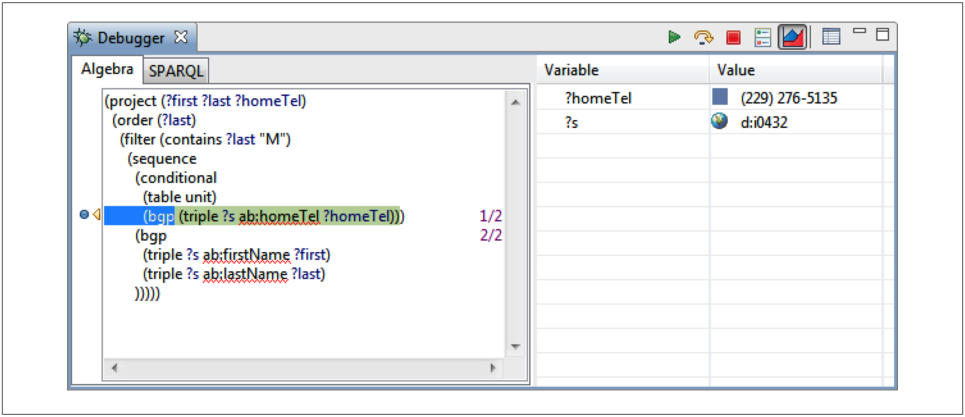
```
16:53:13 INFO  exec :: BGP :: (?s <http://example.org/ab#firstName> ?first)
                              (?s <http://example.org/ab#lastName> ?last)
16:53:13 INFO  exec :: Reorder :: (?s <http://example.org/ab#firstName> ?first)
                                  (?s <http://example.org/ab#lastName> ?last)
16:53:13 INFO  exec :: BGP :: (?s <http://example.org/ab#homeTel> ?homeTel)
16:53:13 INFO  exec :: Reorder :: (?s <http://example.org/ab#homeTel> ?homeTel)
```

The key difference is that the optimized version looks for ab:homeTel triples last instead of first. In other words, the optimizer has the query processor executing the OPTIONAL clause last in the GRAPH pattern instead of first, where it's actually located—a classic bit of query optimization. So, we've found another bonus of understanding a little of the SPARQL algebra: we can see some of the optimizations that ARQ performs.

Debugging Tools

You'll find that other SPARQL tools offer additional features for users who don't mind looking at a bit of SPARQL algebra. For example, the Maestro Edition of TopQuadrant's TopBraid Composer modeling and integrated development environment includes a SPARQL debugger and profiler. Figure 7-1 shows how it displays a SPARQL algebra version of the last query.

(The squiggly lines under the property names show that they have not been declared as part of the currently open model; Composer does this to help you coordinate your query with a schema, which is handy when creating complex applications.) The tiny circle in the left margin of the debugger view shows that the triple pattern on that line



```
(project (?first ?last ?homeTel)
 (order (?last)
  (filter (contains ?last "M")
   (sequence
    (conditional
     (table unit)
     (bgp (triple ?s ab:homeTel ?homeTel)))          1/2
    (bgp                                             2/2
     (triple ?s ab:firstName ?first)
     (triple ?s ab:lastName ?last)
    )))))
```

Variable	Value
?homeTel	(229) 276-5135
?s	d:i0432

Figure 7-1. TopBraid Composer's SPARQL debugger view

has been set as a breakpoint, and with the query executed up to that breakpoint, we can see the bindings of that line's variables on the right.

The numbers to the right of a triple pattern show how many times it has been executed and how many triples it has returned. For example, the screen shot shows that the `ab:homeTel` pattern has been executed once, returning two triples. By showing where in your query the SPARQL engine spends most of its time, this information can be very useful when analyzing query performance.

Breakpoints, the ability to inspect variable bindings, and a profiler are typical of the features that you'll find in a debugging environment for most popular programming languages, and it's nice to see them available for SPARQL as well. Dig through the documentation for your SPARQL processor to see how it can help you understand more detail about how it executes your queries.

The more you know about how your SPARQL engine processes your queries, the more opportunities you have to tune it.

Summary

In this chapter, we learned about:

- How the main job of a query processor is to return requested data or give up upon not finding it, and how reducing the search space can speed both of these tasks
- Reordering of triple patterns to help a query engine do its job faster
- The potential effects of other WHERE clause features on execution time, such as OPTIONAL and FILTER statements

- How some function calls can ask more of a processor than others, especially when processing strings
- How the extra flexibility of SPARQL 1.1 property paths also has a processing cost
- How the LIMIT keyword can speed things up, and how other keywords outside of the WHERE clause, such as ORDER BY and DISTINCT, can slow things down
- How asking for fewer variables on the SELECT line (or none, in an ASK query) can speed things up
- How old-fashioned debugging techniques such as commenting out code and displaying the values of temporary variables can give you insight into problems a query may be having
- How some query engines include extra features to tell you even more about system usage and query optimization

CHAPTER 8

Working with SPARQL Query Result Formats

SPARQL engines can usually return query results in a range of different syntaxes. The choices include offerings for everyone from professional software engineers to business analysts who are happiest with Excel spreadsheets. Fortunately, the syntaxes are part of the SPARQL family of standards, so that when you develop something to process a query result format from one particular query engine, you can use it with others.

In this chapter, we'll look at the standardized formats:

"SPARQL Query Results XML Format" on page 238
: This is a simple, straightforward XML format that makes it easy to use query results in an XML-based system such as a publishing workflow. In this section, we'll see how to turn this format into another standardized kind of XML that you can then convert into a PDF file with open source tools.

"SPARQL Query Results JSON Format" on page 244
: The JavaScript Object Notation is an increasingly popular format for passing data between both local and remote processes, giving you new opportunities to integrate SPARQL tools with other systems.

"SPARQL Query Results CSV and TSV Formats" on page 249
: You can pull data in Comma-Separated Value and Tab-Separated Value text files directly into spreadsheet programs such as Excel and OpenOffice. As we'll see, these two formats have more differences between them than just the delimiter characters that separate returned values.

An important issue with each format, apart from the actual syntax it uses to represent the query results, is the amount of metadata it provides with the results. For example, the format may or may not indicate the datatype of a returned piece of information, or whether it's a literal or a URI—and, if the former, whether there's a datatype or spoken language associated with it.

Different SPARQL engines have different ways for you to tell them that you want a particular result format. With ARQ, add a parameter of `XML`, `JSON`, `CSV`, or `TSV` after the `--results` command-line switch.

To get an idea of the possibilities, we'll look at how each format represents the result of a simple query for the subjects and objects of this dataset's triples:

```
# filename: ex409.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#> .

d:id1 rdfs:label "book" .
d:id2 rdfs:label "5 bucks"@en-US .
d:id3 dm:shipped true  .
d:id4 dm:location _:b1 .
d:id5 dm:amount 3 .
```

Asking for the subjects will show us how the query results represent URIs, and asking for the objects will show us how the various bits of metadata are represented.

See "Making RDF More Readable with Language Tags and Labels" on page 31 for background on language tags, "Blank Nodes and Why They're Useful" on page 33 for more on blank nodes like the one shown with `d:id4` in the sample data above, and Chapter 5 for more on datatypes and how Turtle can represent them.

Here is our simple query,

```
# filename: ex408.rq

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

and here is ARQ's output when we apply query ex408.rq to the ex409.ttl dataset without specifying a result format:

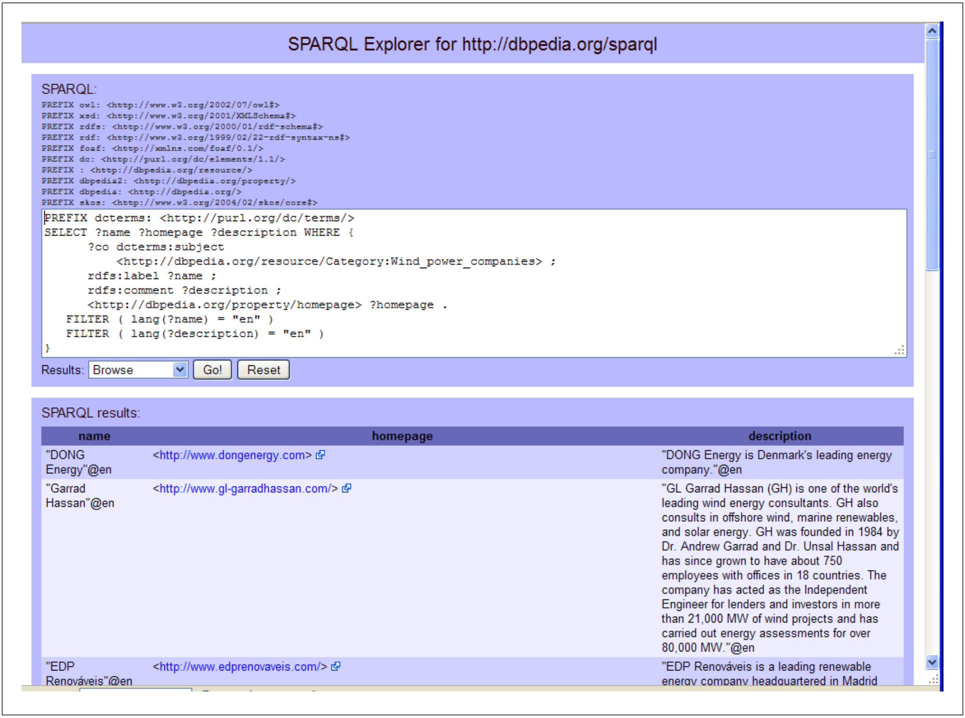
```
-------------------------------------------------------------
| s                                     | o               |
=============================================================
| <http://learningsparql.com/ns/data#id3> | true            |
| <http://learningsparql.com/ns/data#id1> | "book"          |
| <http://learningsparql.com/ns/data#id4> | _:b0            |
| <http://learningsparql.com/ns/data#id2> | "5 bucks"@en-US |
| <http://learningsparql.com/ns/data#id5> | 3               |
-------------------------------------------------------------
```

For a more practical application of each output format, we'll also look at how each represents DBpedia's result for the following query. It asks for the name, home page

URL, and description of energy companies involved in wind power, filtered to return only English language names and descriptions:

```
# filename: ex406.rq

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

SELECT ?name ?homepage ?description
WHERE
{
   ?co dcterms:subject
       <http://dbpedia.org/resource/Category:Wind_power_companies> ;
   rdfs:label ?name ;
   rdfs:comment ?description ;
   <http://dbpedia.org/property/homepage> ?homepage .
   FILTER ( lang(?name) = "en" )
   FILTER ( lang(?description) = "en" )
}
```

The bottom half of Figure 8-1 shows how DBpedia's SNORQL interface displays the beginning of this query's results.



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 dcterms: <http://purl.org/dc/terms/>
SELECT ?name ?homepage ?description WHERE {
      ?co dcterms:subject
          <http://dbpedia.org/resource/Category:Wind_power_companies> ;
      rdfs:label ?name ;
      rdfs:comment ?description ;
      <http://dbpedia.org/property/homepage> ?homepage .
   FILTER ( lang(?name) = "en" )
   FILTER ( lang(?description) = "en" )
}
```

SPARQL results:

name	homepage	description
"DONG Energy"@en	<http://www.dongenergy.com>	"DONG Energy is Denmark's leading energy company."@en
"Garrad Hassan"@en	<http://www.gl-garradhassan.com/>	"GL Garrad Hassan (GH) is one of the world's leading wind energy consultants. GH also consults in offshore wind, marine renewables, and solar energy. GH was founded in 1984 by Dr. Andrew Garrad and Dr. Unsal Hassan and has since grown to have about 750 employees with offices in 18 countries. The company has acted as the Independent Engineer for lenders and investors in more than 21,000 MW of wind projects and has carried out energy assessments for over 80,000 MW."@en
"EDP Renováveis"@en	<http://www.edprenovaveis.com/>	"EDP Renováveis is a leading renewable energy company headquartered in Madrid

Figure 8-1. Using DBpedia's SNORQL interface to list wind power companies

A SPARQL processor can only return the XML and JSON formats when you send it SELECT or ASK queries, and the "Query Results CSV and TSV Formats" Recommendation only describes the result of SELECT queries. For CONSTRUCT queries, which return triples, you would want the result in an RDF serialization such as Turtle or RDF/XML so that RDF-related tools like triplestores and other SPARQL engines can read that data directly.

The section "SPARQL and Web Application Development" on page 282 demonstrates how programs that you write and utilities such as *wget* and *curl* can send queries whose results will be delivered in these result formats if you set the HTTP Accept header to the MIME type indicated by the result format's specification.

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 gives you a great opportunity to let SPARQL queries and results play a role in those applications.

A SPARQL ASK query (described further in the section "Defining Rules with SPARQL" on page 124) can only return a boolean true or false value. Because of this, there are only two possible XML documents, not counting optional metadata, that a query engine might return to you when you send it an ASK query and request a result in SPARQL Query Results XML Format. Here's one of them:

```
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head/>
  <boolean>true</boolean>
</sparql>
```

As with any SPARQL Query Results XML Format result, all of this little document's elements are in the *http://www.w3.org/2005/sparql-results#* namespace. The `head` element is required, but none of its children are. The only remaining result of an ASK query is the `boolean` element, which tells you whether the query returned a true or false value.

You're much more likely to use this query result format with a SELECT query. When you run the simple ex408.rq query shown at the beginning of this chapter against the ex409.ttl dataset shown with it and ask for an XML result set, you get this:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="s"/>
    <variable name="o"/>
  </head>
  <results>
    <result>
      <binding name="s">
        <uri>http://learningsparql.com/ns/data#id3</uri>
      </binding>
      <binding name="o">
        <literal datatype="http://www.w3.org/2001/XMLSchema#boolean">true</literal>
      </binding>
    </result>
    <result>
      <binding name="s">
        <uri>http://learningsparql.com/ns/data#id1</uri>
      </binding>
      <binding name="o">
        <literal>book</literal>
      </binding>
    </result>
    <result>
      <binding name="s">
        <uri>http://learningsparql.com/ns/data#id4</uri>
      </binding>
      <binding name="o">
        <bnode>b0</bnode>
      </binding>
    </result>
    <result>
      <binding name="s">
        <uri>http://learningsparql.com/ns/data#id2</uri>
      </binding>
      <binding name="o">
        <literal xml:lang="en-US">5 bucks</literal>
      </binding>
    </result>
    <result>
      <binding name="s">
        <uri>http://learningsparql.com/ns/data#id5</uri>
      </binding>
      <binding name="o">
        <literal datatype="http://www.w3.org/2001/XMLSchema#integer">3</literal>
      </binding>
    </result>
  </results>
</sparql>
```

The SPARQL Query Results XML Recommendation explains every element and attribute that may come up in one of these, but when you look at this sample XML document and compare it with ARQ's default output when run with the same query dataset, its basic structure is pretty clear:

- The document element (which, like all the other elements, is in the *http://www.w3.org/2005/sparql-results#* namespace declared as the document's default) is called `sparql`, and it has two child elements: the `head` element lists the selected variable names and the `results` element contains the actual results. (After the `variable` elements, the `head` may also contain a `link` element with an `href` attribute pointing to additional metadata about the query results.)
- Each row of the returned 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 a blank node is stored in a `bnode` element whose value may or may not reflect what you saw in the input. (Remember, any blank node's name is just a temporary placeholder and not guaranteed to survive the next processing step. See "Blank Nodes and Why They're Useful" on page 33 for more on these.) If the value is a literal, it's in a `literal` element. A language tag for a literal is stored in an `xml:lang` attribute, and a datatype is stored as a URI—usually from the W3C Schema namespace for defining basic datatypes—in a `datatype` attribute.

The query results XML format spec includes links to RELAX NG and W3C schemas of the result format.

If you write a program to process some SPARQL Query Results XML, it may be able to ignore the `head` element because variable names are also provided with the `binding` elements inside of the `results` element.

At the beginning of this chapter, Figure 8-1 showed a screen shot of DBpedia's SNORQL interface with a query about wind power companies and the first few results displayed below that. If you change that form's "Results:" field from "Browse" to "XML" and click the Go! button, your browser will request a plain XML version of the result.

The following shows a sample of the XML version of the wind power company query results:

```
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="name"/>
    <variable name="homepage"/>
    <variable name="description"/>
  </head>
  <results>
    <result>
      <binding name="name">
        <literal xml:lang="en">DONG Energy</literal>
      </binding>
      <binding name="homepage">
        <uri>http://www.dongenergy.com</uri>
      </binding>
      <binding name="description">
        <literal xml:lang="en">DONG Energy is Denmark's leading energy
        company.</literal>
      </binding>
    </result>
    <result>
      <binding name="name">
        <literal xml:lang="en">Garrad Hassan</literal>
      </binding>
      <binding name="homepage">
        <uri>http://www.gl-garradhassan.com/</uri>
      </binding>
      <binding name="description">
        <literal xml:lang="en">GL Garrad Hassan (GH) is one of the world's
leading wind energy consultants. GH also consults in offshore wind, marine
renewables, and solar energy. GH was founded in 1984 by Dr. Andrew Garrad
and Dr. Unsal Hassan and has since grown to have about 750 employees with
offices in 18 countries. The company has acted as the Independent Engineer
for lenders and investors in more than 21,000 MW of wind projects and has
carried out energy assessments for over 80,000 MW.</literal>
      </binding>
    </result>

   <!-- Seven more result elements with the same structure -->

  </results>
</sparql>
```

Processing XML Query Results

Some programming languages include built-in support for parsing XML into their native data structures. For those that don't, XML parsing libraries are usually easy to find. One of the most popular languages for processing XML is XSLT, a W3C companion specification to XML. An XSLT stylesheet has a series of template rules that each describe what to do with a particular class of input elements or attributes.

The SPARQL Query Results XML Format specification links to an XSLT stylesheet that converts XML in this format to HTML. Fuseki includes a similar stylesheet in its `pages` subdirectory, and you can download the one that SNORQL uses from *http://dbpedia.org/snorql/xml-to-html.xsl*. Using any of these XSLT stylesheets as a starting point, you can create another stylesheet pretty quickly that outputs HTML with the look and feel that you want.

Let's look at another XSLT stylesheet. This one converts the SPARQL Query Results XML from the wind power query to Darwin Information Typing Architecture XML (DITA), an OASIS standard that, like DocBook, is popular for technical documentation. One advantage of using DITA is the open source DITA Open Toolkit, which (among other things) converts DITA XML to the W3C XSL-FO standard—another companion to the XML specification—for representing printed layout of content. You can then convert your XSL-FO file to PDF using XSL-FO processors such as the open source Apache FOP program. Putting all of these pieces together, we'll see how to create a formatted PDF file from SPARQL Query Results XML.

This stylesheet converts the query result XML to the DITA representation of a table. Because SPARQL query results are already pretty tabular, most of the stylesheet's template rules just map a single SPARQL result element to the appropriate DITA table element. The last one gets a little fancier, converting any `uri` element it finds to the DITA equivalent of a hypertext link:

```
<!--
     filename: ex402.xsl
     Convert XML SPARQL query results to a DITA Concept document.
-->
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:s="http://www.w3.org/2005/sparql-results#"
                exclude-result-prefixes="s">

  <xsl:output doctype-public="-//OASIS//DTD DITA Concept//EN"
doctype-system="C:\usr\local\dita\DITA-OT1.5\dtd\technicalContent\dtd\concept.dtd"/>

  <xsl:template match="s:sparql">
    <concept id="id1">
      <title>Wind Power Companies</title>
      <conbody>
        <table>
          <tgroup cols="{count(s:head/s:variable)}">
            <xsl:apply-templates/>
          </tgroup>
        </table>
      </conbody>
    </concept>
  </xsl:template>

  <xsl:template match="s:head">
    <thead>
      <row>
        <xsl:apply-templates/>
      </row>
    </thead>
  </xsl:template>

  <xsl:template match="s:variable">
    <entry><xsl:value-of select="@name"/></entry>
  </xsl:template>

  <xsl:template match="s:results">
    <tbody><xsl:apply-templates/></tbody>
  </xsl:template>
  <xsl:template match="s:result">
    <row><xsl:apply-templates/></row>
  </xsl:template>

  <xsl:template match="s:binding">
    <entry><xsl:apply-templates/></entry>
  </xsl:template>

  <xsl:template match="s:literal">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="s:uri">
    <xref format="html" href="{.}">
      <xsl:apply-templates/>
    </xref>
  </xsl:template>

</xsl:stylesheet>
```

Except for the first template rule's addition of the "Wind Power Companies" title and the last template rule's assumption that all `uri` values are valid web addresses, this spreadsheet is generic enough to use with the XML query results of just about any SPARQL query.

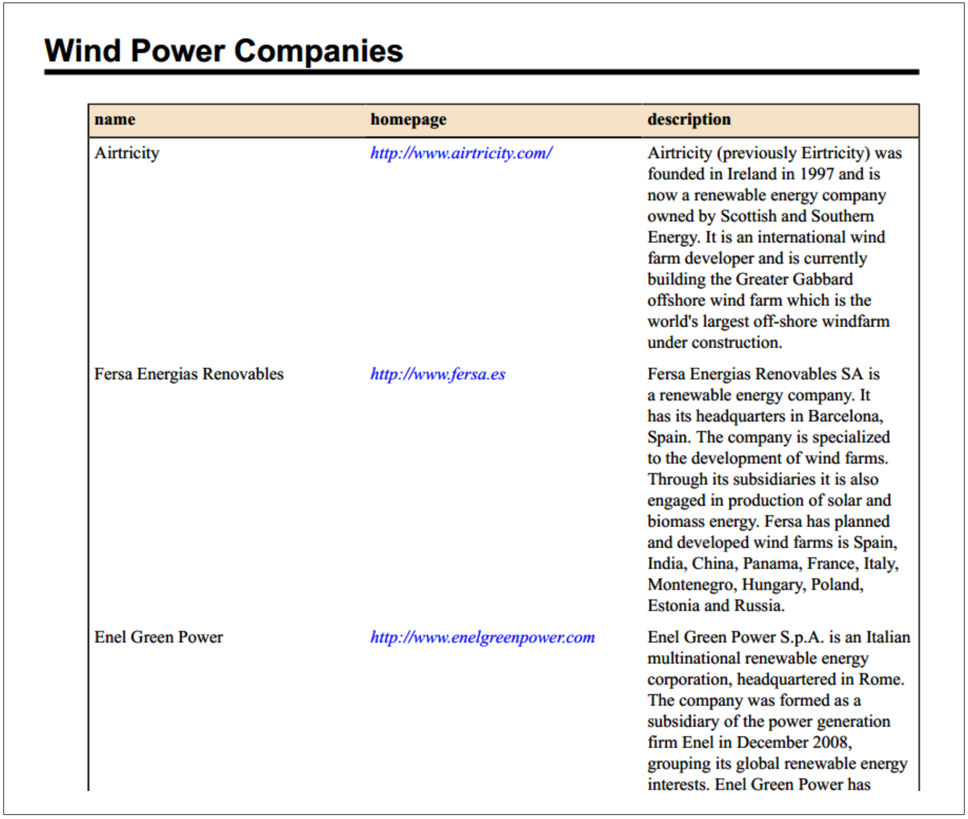
After running this XSLT stylesheet on the wind power company XML query results and then sending the stylesheet output through the DITA Open Toolkit and FOP, the beginning of the resulting PDF will look like Figure 8-2.

Because of the limited range of elements and attributes that may appear in the SPARQL Query Results XML format, the quickest way to create this kind of XSLT stylesheet is often to take another one, such as one of the ones mentioned earlier that create HTML output, and to just revise what it outputs to include the markup you want. That's how I created the preceding stylesheet, and it's also how I created a stylesheet to prepare the tables shown in Chapter 6.

The output of one of these stylesheets doesn't even have to be XML. With a `method="text"` attribute setting in your XSLT stylesheet's `xsl:output` element, you can create almost any kind of text output you want.

If you do plan to create something that's very different from XML, you should consider working with JSON query results, as described in the next section, instead of XML query results.

XML is what flows through the plumbing of most automated publishing systems these days, and the ability to retrieve SPARQL query results in a straightforward XML format makes it simple to incorporate the power of RDF technology and the Linked Data cloud into these publishing systems. If you regularly work with XML, you're going to love SPARQL Query Results XML format.


Wind Power Companies

name	homepage	description
Airtricity	http://www.airtricity.com/	Airtricity (previously Eirtricity) was founded in Ireland in 1997 and is now a renewable energy company owned by Scottish and Southern Energy. It is an international wind farm developer and is currently building the Greater Gabbard offshore wind farm which is the world's largest off-shore windfarm under construction.
Fersa Energias Renovables	http://www.fersa.es	Fersa Energias Renovables SA is a renewable energy company. It has its headquarters in Barcelona, Spain. The company is specialized to the development of wind farms. Through its subsidiaries it is also engaged in production of solar and biomass energy. Fersa has planned and developed wind farms is Spain, India, China, Panama, France, Italy, Montenegro, Hungary, Poland, Estonia and Russia.
Enel Green Power	http://www.enelgreenpower.com	Enel Green Power S.p.A. is an Italian multinational renewable energy corporation, headquartered in Rome. The company was formed as a subsidiary of the power generation firm Enel in December 2008, grouping its global renewable energy interests. Enel Green Power has

Figure 8-2. Beginning of PDF created from result of wind power companies query using XSLT, the DITA Open Toolkit, and Apache FOP

SPARQL Query Results JSON Format

1.1 Alert

The “Serializing SPARQL Query Results in JSON” document was one of the SPARQL 1.0 specifications, but because the W3C released it as a Note and not a Recommendation, it was not an official standard. (In standards-speak, it was not “normative.”) This specification was easy enough to implement that many SPARQL 1.0 engines supported it anyway, and the newer “Query Results JSON Format” specification is an official Recommendation.

The W3C specification “SPARQL 1.1 Query Results JSON Format” describes how SPARQL processors can return a JavaScript Object Notation object instead of XML. The IETF document that defines the JSON standard, RFC 4627, defines a JSON object as “an unordered collection of zero or more name-value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array." The syntax for representing all of this is pretty simple:

- Objects are enclosed with curly braces.
- A name-value pair is separated by a colon, and a name must be unique within an object.
- A list of name-value pairs is delimited by commas.
- Arrays are enclosed by square braces.

There are other potential bits of syntax, but this much will get you pretty far.

Let's start by looking at the JSON results of a SPARQL ASK query, (described further in "Defining Rules with SPARQL" on page 124) which can only return a true or false value:

```
{
  "head" : { } ,
  "boolean" : true
}
```

This object has two name-value pairs: the first is named `head`, and its value is an empty JSON object because the query engine returned none of the optional head metadata. The second is named `boolean` and shows the returned value.

When a SPARQL engine returns the results of a SELECT query as a JSON object, there's more to it, but it's still pretty straightforward because the results are a table and therefore easy to represent in a simple data structure. The following shows the JSON query result of running ex408.rq with the ex409.ttl dataset from the beginning of the chapter:

```
{
  "head": {
    "vars": [ "s" , "o" ]
  } ,
  "results": {
    "bindings": [
      {
        "s": { "type": "uri" , "value": "http://learningsparql.com/ns/data#id3" } ,
        "o": { "datatype": "http://www.w3.org/2001/XMLSchema#boolean" ,
               "type": "typed-literal" , "value": "true" }
      } ,
      {
        "s": { "type": "uri" , "value": "http://learningsparql.com/ns/data#id1" } ,
        "o": { "type": "literal" , "value": "book" }
      } ,
      {
        "s": { "type": "uri" , "value": "http://learningsparql.com/ns/data#id4" } ,
        "o": { "type": "bnode" , "value": "b0" }
      } ,
      {
        "s": { "type": "uri" , "value": "http://learningsparql.com/ns/data#id2" } ,
        "o": { "type": "literal" , "xml:lang": "en-US" , "value": "5 bucks" }
      } ,
      {
        "s": { "type": "uri" , "value": "http://learningsparql.com/ns/data#id5" } ,
        "o": { "datatype": "http://www.w3.org/2001/XMLSchema#integer" ,
               "type": "typed-literal" , "value": "3" }
      }
    ]
  }
}
```

Remember that JSON is unordered, so any code you use to process this shouldn't assume that the head object appears before results object.

The Query Results JSON Format Recommendation explains everything that might come up in one of these JSON objects, but we can see the most important ones in this example. The top level JSON object has two name-value pairs: `head` and `results`, each of which has an object as its value. The key part of the `head` object is its `vars` value: an array listing the variables that the SELECT query asked for and that are returned in the query results.

Your program that processes JSON SPARQL query results may be able to ignore the `head` object, because each query result includes variable names in the `bindings` object.

The most important part of the `results` element is its `bindings` object, which is an array of objects corresponding to each row of the result set. Each object in the `bindings` array has a name-value pair for each requested variable. In the prior example, you can see that each has an `s` and an `o` object. Each of these objects has metadata about its value, such as its type and whether any spoken language tag was assigned to it; the most important part here is the `value` object, a string showing the value assigned to the variable. You can see that although the sample JSON query results enclose the `s` values (and all the others) in quotes, the `type` values with them show that the `s` values are not strings but URIs. As with the XML result format, a blank node is called a `bnode` and may or may not have the same name (in this case, "b0") that it did in the original dataset.

When we send the query about wind power companies shown at the beginning of this chapter to DBpedia and ask for a JSON response, we get something like the following (only the first two of the nine result rows are shown here, and the description of the second energy company is trimmed to fit on the page):

```
{ "head": { "link": [], "vars": ["name", "homepage", "description"] },
  "results":
    { "distinct": false,
      "ordered": true,
      "bindings":

      [
        { "name": { "type": "literal", "xml:lang": "en", "value": "DONG Energy" }    ,
          "homepage": { "type": "uri", "value": "http://www.dongenergy.com" }    ,
          "description": { "type": "literal",
                           "xml:lang": "en",
                           "value": "DONG Energy is Denmark's leading energy company."
                         }
        },

        { "name": { "type": "literal", "xml:lang": "en", "value": "Garrad Hassan" },
          "homepage": { "type": "uri", "value": "http://www.gl-garradhassan.com/" },
          "description": { "type": "literal",
                           "xml:lang":
                           "en", "value": "GL Garrad Hassan (GH) is one of..." }
        }
      ]
    }
}
```

Processing JSON Query Results

Libraries to read and write JSON objects are available in most popular programming languages; "SPARQL and Web Application Development" on page 282 shows two Python scripts that use such a library. Because the JS in JSON stands for JavaScript, we'll take a look at a JavaScript program that converts the JSON query results about wind power companies above into HTML.

I ran the following script with the open-source Rhino JavaScript engine that is part of the Mozilla project. (You may already have the Rhino js.jar file on your computer without even knowing it, because it's sometimes included as part of larger programs.) With the script stored in ex407.js, the following command line worked in both Linux and Windows to create HTML from the example JSON object shown above, which I had stored in the file ex405.js:

```
java -jar js.jar ex407.js < ex405.js
```

The ex407.js script reads the standard input sent to it, which it assumes is a JSON SPARQL query result object, and then iterates through the pieces of that object, outputting an HTML representation of the results with a little CSS in the output file's `head` element:

```
// filename: ex407.js: Convert JSON results of SPARQL query about
// wind powercompanies to HTML. Reading of disk file based on case
// conversion demo at http://en.wikipedia.org/wiki/Rhino_(JavaScript_engine)

importPackage(java.io);      // for BufferedReader
importPackage(java.lang);    // for System[]

// Read the file into the string s

var reader = new BufferedReader( new InputStreamReader(System['in']) );
```

```
    var s = true;
    var result = "";

    while (s) {
        s = reader.readLine();
        if (s != null) {
            result = result + s;
        }
    };

    // Parse the string into a JSON object

    var JSONObject = JSON.parse(result);

    // Output the values stored in the JSON object as an HTML table.

    print("<html><head>");
    print("<style type='text/css'>* { font-family: arial,helvetica; }</style>");
    print("</head><body>");
    print("<table border='1' style='border: 1px solid; border-collapse: collapse;'>");
    print("<tr><th>Name</th><th>Description</th></td>");

    // Make each company name a link to their homepage.

    for (i in JSONObject.results.bindings) {
        print("<tr><td><a href='");
        print(JSONObject.results.bindings[i].homepage.value);
        print("'>");
        print(JSONObject.results.bindings[i].name.value);
        print("</a></td><td>");
        print(JSONObject.results.bindings[i].description.value);
        print("</td></tr>");
    }

    print("</table></body></html>");
```

Some of the input and output parts of this script are specific to how Rhino handles disk files, but the navigation of the JSON object would be similar with any JavaScript program: you iterate through the `bindings` array, pulling what you need out of each array member to use as you see fit.

JSON is becoming an increasingly common way for processes on different machines to pass data back and forth, and the ability of SPARQL engines to give you results in JSON makes it much easier to take advantage of RDF technology when working with processes that can do this.

SPARQL Query Results CSV and TSV Formats

1.1 Alert

CSV and TSV output were not mentioned in any of the SPARQL 1.0 specifications, but several SPARQL 1.0 engines supported them anyway. "SPARQL 1.1 Query Results CSV and TSV Formats" is a W3C Recommendation and therefore an official standard.

The SPARQL Query Results CSV and TSV Formats Recommendation document is very brief—once you get past the table of contents, you're about half done with it. The CSV format is the simplest of the two, so we'll look at that first.

If you thought that the CSV and TSV formats would be the same except for the character used as a delimiter, there's more to it. Make sure to compare the following sample output for the two formats when the ex408.rq query was run on the ex409.ttl data from the beginning of the chapter.

The first line in a Query Results CSV SPARQL result set is a list of the variable names in the query without the question marks that they would have in the actual query. After that, each line shows a result set row with values separated by commas. All values, including URIs, are output as if they had first passed through SPARQL's `str()` function, which converts each to a plain string with no associated metadata. (See "Node Type Conversion Functions" on page 153 for more about this function.)

When running the ex408.rq query with the ex409.ttl data from the beginning of the chapter, here's how the result looks in CSV result format:

```
s,o
http://learningsparql.com/ns/data#id3,true
http://learningsparql.com/ns/data#id1,book
http://learningsparql.com/ns/data#id4,6cc1b60b:13608e5e580:-7ffe
http://learningsparql.com/ns/data#id2,5 bucks
http://learningsparql.com/ns/data#id5,3
```

If a value has any commas, carriage returns, or single or double quote characters, it will be enclosed in quotes. If the value has any double quote characters within it, an extra one is inserted before each to escape it, so that `'5 "great" bucks'` would be written as `"5 ""great"" bucks"`. (Practices like this, and corresponding ones for the TSV results set, are not something that the SPARQL Working Group made up, but are based on IETF RFC specifications for both CSV and TSV.)

There's no way to know from these CSV values that "6cc1b60b:13608e5e580:-7ffe" represents a blank node, but a long string of hexadecimal digits with one or more colons in them is a typical internal name for a blank node, because the processing system had to come up with some sort of unique name for each one.

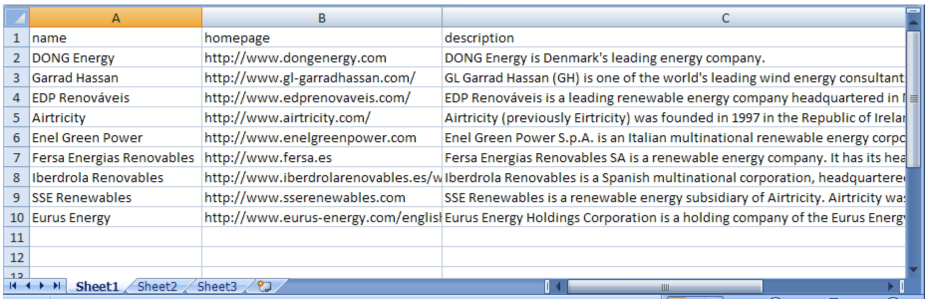
Blank node values are rarely useful in SPARQL query output, especially in the CSV flavor, because there's no clear indication that they were blank nodes. So, there's not much point in having your SELECT query output them if you can help it.

Using CSV Query Results

Most programming languages include a function call that lets you split a comma-delimited list into an array. Splitting the values at every comma in a line of text can lead to trouble if any of the values have commas within them, so check whether the programming language you're using has a library to account for this possibility such as Perl's Text::CSV module.

Frankly, if you're writing a program to process the results of a SPARQL query, you're better off using some other result format besides CSV, because all the other formats provide more information for your program to work with. CSV has the most to offer for nonprogrammers, because it's so easy to just pull it into a spreadsheet program. Microsoft Excel can open up a CSV file directly, and OpenOffice will open one after first displaying an import configuration dialog box that's already filled out for CSV import. (If you want to set such parameters when importing into Excel, use the Text Import wizard, available by picking From Text on the Get External Data section of Excel 2010's Data tab.) If you need to send data to someone with minimal technical background, CSV is often the simplest option—or, to save them the import step, you can import it into a spreadsheet yourself and send them that.

Figure 8-3 shows how the CSV result of the wind power company query from the beginning of the chapter looks after being imported into Excel.



	A	B	C
1	name	homepage	description
2	DONG Energy	http://www.dongenergy.com	DONG Energy is Denmark's leading energy company.
3	Garrad Hassan	http://www.gl-garradhassan.com/	GL Garrad Hassan (GH) is one of the world's leading wind energy consultant
4	EDP Renováveis	http://www.edprenovaveis.com/	EDP Renováveis is a leading renewable energy company headquartered in
5	Airtricity	http://www.airtricity.com/	Airtricity (previously Eirtricity) was founded in 1997 in the Republic of Irela
6	Enel Green Power	http://www.enelgreenpower.com	Enel Green Power S.p.A. is an Italian multinational renewable energy corpo
7	Fersa Energias Renovables	http://www.fersa.es	Fersa Energias Renovables SA is a renewable energy company. It has its hea
8	Iberdrola Renovables	http://www.iberdrolarenovables.es/w	Iberdrola Renovables is a Spanish multinational corporation, headquartere
9	SSE Renewables	http://www.sserenewables.com	SSE Renewables is a renewable energy subsidiary of Airtricity. Airtricity wa
10	Eurus Energy	http://www.eurus-energy.com/englis	Eurus Energy Holdings Corporation is a holding company of the Eurus Energ
11			
12			

Figure 8-3. CSV of result of wind power companies query, imported into Microsoft Excel

If your CSV file's data has any non-ASCII characters, they're probably in the UTF-8 encoding, and Excel and OpenOffice may think that they're in Latin 1 (or, as Microsoft puts it, "Windows (ANSI)") and display them incorrectly. To correct this with either spreadsheet program, set the encoding to UTF-8 on the dialog box that lets you configure the import process. I had to set the File Origin field of Excel's Text Import Wizard to UTF-8 to make it properly display the "á" characters in the fourth row of Figure 8-3's spreadsheet. With OpenOffice, it's the Character set field.

TSV Query Results

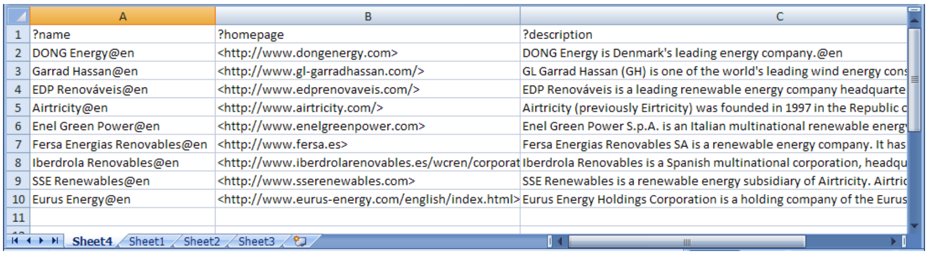
When a SPARQL engine returns results as tab-separated values, in addition to using tab characters as delimiters instead of commas, it provides more information by representing URIs, datatypes, and language tags the same way that SPARQL and Turtle do. For example, here's the TSV result of running the ex408.rq query on the ex409.ttl data from the beginning of the chapter:

```
?s    ?o
<http://learningsparql.com/ns/data#id3>    true
<http://learningsparql.com/ns/data#id1>    "book"
<http://learningsparql.com/ns/data#id4>    _:b0
<http://learningsparql.com/ns/data#id2>    "5 bucks"@en-US
<http://learningsparql.com/ns/data#id5>    3
```

The space after each > character is a tab character. We can see the following differences from the CSV version of this query result:

- Variable names include question marks at the beginning.
- URIs are enclosed with <> characters.
- Literals like "book" are enclosed in quotation marks.
- Typed literals include an indication of the type—for example, if "book" was an `xsd:string`, the TSV output would be `"book"^^<http://www.w3.org/2001/XMLSchema#string>`. The output uses abbreviated representations of types when possible, so `3` without quotes is the same as `"3"^^xsd:integer` and `true` without quotes represents `"true"^^xsd:boolean`. (See "Data Typing" on page 30 for more on these shortcuts.)
- Language tags are appended to quoted strings the same way that datatype indicators are.
- A blank node is represented as an underscore followed by a colon and an identifier that is assigned to it at output time.

Figure 8-4 shows the TSV version of the wind power company query results imported into Excel. (As with the Excel import of the CSV version of this data, I had to set the File Origin field of the Text Import Wizard to UTF-8 to make the "á" characters on the fourth row display properly.) The extra information described in the bulleted list above



	A	B	C
1	?name	?homepage	?description
2	DONG Energy@en	<http://www.dongenergy.com>	DONG Energy is Denmark's leading energy company.@en
3	Garrad Hassan@en	<http://www.gl-garradhassan.com/>	GL Garrad Hassan (GH) is one of the world's leading wind energy cons
4	EDP Renováveis@en	<http://www.edprenovaveis.com/>	EDP Renováveis is a leading renewable energy company headquarte
5	Airtricity@en	<http://www.airtricity.com/>	Airtricity (previously Eirtricity) was founded in 1997 in the Republic c
6	Enel Green Power@en	<http://www.enelgreenpower.com>	Enel Green Power S.p.A. is an Italian multinational renewable energ
7	Fersa Energias Renovables@en	<http://www.fersa.es>	Fersa Energias Renovables SA is a renewable energy company. It has
8	Iberdrola Renovables@en	<http://www.iberdrolarenovables.es/wcren/corporat	Iberdrola Renovables is a Spanish multinational corporation, headqu
9	SSE Renewables@en	<http://www.sserenewables.com>	SSE Renewables is a renewable energy subsidiary of Airtricity. Airtric
10	Eurus Energy@en	<http://www.eurus-energy.com/english/index.html>	Eurus Energy Holdings Corporation is a holding company of the Eurus
11			

Figure 8-4. TSV of result of wind power companies query, imported into Microsoft Excel

may not be especially useful when manipulating this as spreadsheet data, but it could be valuable if using this data to generate Turtle triples at some later point.

TSV files can also be a useful way to store *quads*, or groupings of subjects, predicates, objects, and, if the triple is part of a named graph, the graph's name. TSV output of the ex332.rq List All Triples query would show you an example.

Summary

In this chapter we learned about:

- The SPARQL Query Results XML format and how XSLT and other XML-processing tools can use it
- The SPARQL Query Results JSON format, which is also easy to process with most modern programming languages
- The SPARQL Query Results CSV format, which represents all values as plain strings and is handy for nonprogrammers to use in spreadsheets
- SPARQL Query Results TSV format, which uses Turtle and SPARQL conventions to include some extra metadata about the values and can be imported into any spreadsheet program

CHAPTER 9

RDF Schema, OWL, and Inferencing

In Chapter 2, we learned the basics of the roles that RDF Schema (RDFS) and the Web Ontology Language (OWL) can play in RDF applications:

- RDFS and OWL are W3C standard vocabularies that let you define and describe classes and properties that a dataset's triples might use. These do not function as templates that the data must conform to, as schemas often do in other data modeling systems, but instead as additional metadata to help you get more out of your data.
- RDFS and OWL statements themselves are expressed using triples, so you can query them with SPARQL.
- Properties and classes from the RDFS and OWL vocabularies let you describe your own properties and classes in ways that let certain applications infer new information from your dataset. For example, if you have one triple that specifies that the `ab:spouse` property is an `owl:SymmetricProperty`, and you have another triple that tells us that Richard has an `ab:spouse` value of Cindy, then an application that understands what `owl:SymmetricProperty` means will know that Cindy has an `ab:spouse` value of Richard. This is a classic example of the value of metadata: it adds information about your data so that you can get more out of it.

What kind of applications understand what `owl:SymmetricProperty` means, or what RDFS properties such as `rdfs:domain` and `rdfs:range` mean? And how does this extend the power of what you can do with your SPARQL queries? We'll learn the answers to these questions in this chapter, which covers:

"What Is Inferencing?" on page 254
: The word "inferencing" can mean many things, and has a very specific meaning when we talk about RDF.

"SPARQL and RDFS Inferencing" on page 258
: How can you use your SPARQL skills with applications that can perform RDFS inferencing (that is, applications that understand RDFS vocabulary terms such as `rdfs:domain`) to get more out of your data?

"SPARQL and OWL Inferencing" on page 263
: How can you use your SPARQL skills with applications that can perform OWL inferencing to get even more out of your data?

"Using SPARQL to Do Your Inferencing" on page 269
: SPARQL itself can do some inferencing, which adds flexibility to how you work with your data.

"Querying Schemas" on page 271
: How SPARQL queries can tell you more about RDFS and OWL schemas and ontologies.

What Is Inferencing?

Webster's New World College Dictionary defines "infer" as "to conclude or decide from something known or assumed." When you do RDF inferencing, your existing triples are the "something known," and your inference tools will infer new triples from them. (These new triples may let you come to some sort of conclusion or decision, depending on what your application does with them.)

In Chapter 2 we saw an example of two files that worked together to allow this kind of inferencing. The first had some triples of data about Richard Mutt:

```
# 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 .
```

The second had triples that told us more about the `ab:playsInstrument` property used in this data:

```
# 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 .
```

If I said "playsInstrument is a property of the Musician class," in a prescriptive schema—that is, one that defines the required structure of some data, such as an XML schema or one from a relational database or object-oriented system—it would mean that if something is a member of the Musician class, then it can (or perhaps must) have a value for the playsInstrument property. (And, if I didn't say that it was a property of that class, then a member of that class couldn't have a playsInstrument value.) In a descriptive schema language like RDFS, though, it works the other way around: this statement is extra metadata that tells us more about our resources, instead of being a rule for them to follow. It means that if someone plays an instrument, then he or she is a member of the Musician class.

The following query asks for the `ab:firstName` and `ab:lastName` values of any resources that are members of the `ab:Musician` class:

```
# filename: ex415.rq

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

SELECT ?firstName ?lastName
WHERE
{
  ?person a ab:Musician ;
          ab:firstName ?firstName ;
          ab:lastName ?lastName .
}
```

Running the ARQ SPARQL processor with this query and a combination of the two preceding data files won't get you any results because neither file has any triples that fit the triple pattern `{?person a ab:Musician}`, and ARQ doesn't know how to do RDFS inferencing—that is, it doesn't understand what's special about properties like `rdfs:range` and `rdfs:domain`, which ex044.ttl uses to to specify metadata about the `ab:playsInstrument` property.

According to the RDFS specification, an `rdfs:domain` value of `ab:Musician` means that for any triple that has `ab:playsInstrument` as a predicate, the subject of that triple is a member of the class `ab:Musician`. In other words, the new triple that can be inferred from this use of `rdfs:domain` is `{ab:i0432 rdf:type ab:Musician}`. The `rdfs:range` value of `ab:MusicalInstrument` in ex044.ttl means that, for any triple that has `ab:playsInstrument` as a predicate, the object of that triple is a member of the class `ab:MusicalInstrument`, so the triple to infer is `{ab:vacuumCleaner rdf:type ab:MusicalInstrument}`.

The OWL standard builds on the RDFS standard, so an OWL reasoner such as the open source program Pellet does understand what `rdfs:domain` and `rdfs:range` properties are saying. It will use this information to infer that resource `ab:i0432` is an `ab:Musician`, and it supports some of SPARQL, so when the query above asks for the first and last names of any musicians expressed by the triples in the two data files above, running the query against this data with Pellet's command-line query tool does give us an answer:

```
firstName | lastName
====================
"Richard" | "Mutt"
```

Being an OWL reasoner, Pellet also understands the meaning of special classes and properties from the OWL namespace such as `owl:SymmetricProperty` and `owl:inverseOf`, which gives even more power to SPARQL queries executed with Pellet.

The open source command-line version of Pellet only supports SPARQL 1.0, but the version of Pellet incorporated into Clark & Parsia's Stardog triplestore supports SPARQL 1.1.

Later sections of this chapter will show how your SPARQL queries can take further advantage of the metadata that the RDFS and OWL standards can add to your data.

Inferred Triples and Your Query

For a SPARQL query to take advantage of RDFS or OWL, it needs access to both the original triples (in the prior example, those in ex044.ttl and ex045.ttl) and any that should be inferred (or, to use the technical term, any that are entailed—in the prior example, `{ab:i0432 rdf:type ab:Musician}`) by the use of terms from the RDFS and OWL vocabularies in the original triples. Pellet understands the RDFS and OWL vocabularies and supports enough of SPARQL to execute the above query, so for that example, the inferencing and querying happened in one step.

Inferencing and querying of the results is a two-step process that may or may not be automated with the set of inferencing and querying tools that you are using.

The combinations of inferencing and SPARQL query execution that we'll see in this chapter usually fall into one of two categories:

- Running a tool that combines the steps of inferencing and querying for you. Some of these tools let you configure whether they should perform any relevant inferencing before running your SPARQL query; for example, Sesame includes an "Include inferred statements" checkbox on the screen where you enter SPARQL queries, and TopBraid Composer offers a similar button on its SPARQL query panel.
- The use of one tool to do the inferencing and another to execute the SPARQL query against the combination of the input and the output from the inferencing step. The

process of creating inferred triples so that we can use them with another tool is known as materializing them.

Matching a SPARQL query's triple patterns against entailed triples is not always a simple task. As a reference for implementers, the W3C Recommendation "SPARQL 1.1 Entailment Regimes" describes a specific approach for doing this.

Storage of your newly inferred triples with the set used as input to the inferencing operation is sometimes an option as well. As with the denormalization of a relational database, this can speed up the execution of similar queries at later points in time because the software will have less work to do, but it can also increase data maintenance work. For example, let's say that Richard has an `ab:spouse` value of Cindy, so you infer that Cindy has an `ab:spouse` of Richard and then add this inferred triple back to the original set. If Richard and Cindy get a divorce, you have two triples to remove instead of one if you want to keep your data consistent.

More than RDFS, Less than Full OWL

When I describe OWL as a superset of RDFS that specifies additional kinds of metadata that you can add to your data, I'm simplifying things a bit. When OWL first became a set of W3C Recommendations in 2004, the specification listed three "sublanguages" that it labeled OWL Lite, OWL DL, and OWL Full. The section on "SPARQL and OWL Inferencing" on page 263 describes these a bit more, but you can think of them as small, medium, and large in terms of their expressive power and the processing power required to implement them.

Five years later, when OWL 2 became a standard, it included three profiles called OWL 2 EL, OWL 2 QL, and OWL 2 RL. Instead of building on one another the way that DL built on Lite and Full built on DL, the three new profiles were optimized for different kinds of domains that the OWL Working Group had seen people addressing with OWL. We'll also learn more about these three profiles later in this chapter.

To make things a little more complicated (but ultimately, to make useful inferencing easier) some in the RDF community have defined supersets of RDFS that add their favorite bits of OWL to RDFS to create something more powerful than RDFS but easier to implement than any of the OWL flavors described. In Dean Allemang and Jim Hendler's book *Semantic Web for the Working Ontologist* (Morgan Kaufmann, 2011), they describe a superset that they call RDFS+, a spec that has been implemented in TopQuadrant's TopBraid Suite using the kind of SPARQL-based inferencing described in "Using SPARQL to Do Your Inferencing" on page 269. In Franz, Inc.'s AllegroGraph triplestore, they have implemented a slightly different superset of RDFS that they call RDFS++.

As with extensions to the SPARQL query language, the important thing is to be aware of exactly which features are supported by the tools that you're using, and to also be aware of the implications of using tools that don't align exactly with the standards when porting your work to use different standards-based tools. (Because RDFS+ and RDFS++ are both subsets of OWL, though, you should be fine processing either with a typical OWL processor.)

SPARQL and RDFS Inferencing

We've already seen the two most popular RDFS properties several times in this book: `rdfs:label` and `rdfs:comment`. The first is the most common way to add a human-readable label to an otherwise cryptic resource URI; for example, a URI of *http://my-company.com/ns/e04329* that represents an employee in a triplestore might have an `rdfs:label` value of "Richard Mutt". Reports drawing on this triplestore's data would display this name instead of the URI because the URI on its own makes little sense to human readers—as with the unique ID in a relational table, its job is to tie other bits of related information together, not to show up in a report that people have to read.

The `rdfs:comment` property usually describes the resource it's associated with. For example, the DBpedia resource that has an English `rdfs:label` of "Dog" has an English `rdfs:comment` value of three sentences, beginning with "The domestic dog (Canis lupus familiaris), is a subspecies of the gray wolf (Canis lupus), a member of the Canidae family of the mammilian order Carnivora." (It also has `rdfs:label` and `rdfs:comment` values in many other languages.)

These two properties play no role in telling an RDFS inferencing engine about things that it can infer, but inferencing can tell us `rdfs:label` and `rdfs:comment` values of resources that seemingly don't use these properties. For example, if a concept in a SKOS taxonomy has a `skos:prefLabel` of "customer", we can infer that it also has an `rdfs:label` value of "customer", because the SKOS standard defines `skos:prefLabel` as a subproperty of `rdfs:label`.

This might be a bit confusing, so it's worth reviewing: RDF triples let you use properties to describe any resources you want, and properties are themselves resources, so sometimes we use triples to tell us more about the properties being used by specifying properties of those properties. The SKOS standard describes the `skos:prefLabel` property by using the `rdfs:subPropertyOf` property from the RDFS standard to tell us that `skos:prefLabel` is a subproperty of another property from the RDFS standard: `rdfs:label`. The OWL ontology that is included as part of the W3C SKOS standard does this with the triple `{skos:prefLabel rdfs:subPropertyOf rdfs:label}`, so an application that understands RDFS but doesn't know anything about SKOS knows that it can treat any `skos:prefLabel` value that it finds as an `rdfs:label` value—for example, if the application creates a report with a list of resources that uses their human-friendly names instead of their URIs.

Let's look at another example. The popular Dublin Core metadata standard includes a `dc:creator` property. Let's say I have music files on my computer, and I have metadata about them that uses a `dm:composer` property for the classical music files. I also have image files with metadata that uses a `dm:photographer` property. After aggregating my music metadata and image metadata with triples that specify that these two properties are subproperties of `dc:creator`, I end up with a file that includes these triples:

```
# filename: ex417.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#> .
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix nfo:  <http://www.semanticdesktop.org/ontologies/2007/03/22/nfo#> .

dm:composer rdfs:subPropertyOf dc:creator .
dm:photographer rdfs:subPropertyOf dc:creator .

d:file02924 nfo:fileName "9894397.mp3" ;
            dm:composer "Charles Ives" .

d:file74395 nfo:fileName "884930.mp3" ;
            dm:composer "Eric Satie" .

d:file69383 nfo:fileName "119933.mp3" ;
            dm:photographer "Diane Arbus" .

d:file54839 nfo:fileName "204364.mp3" ;
            dm:photographer "Henry Fox Talbot" .
```

For a production application, I wouldn't have defined these properties in my own `http://learningsparql.com/ns/demo#` namespace so quickly because I'll bet that there are vocabularies out there that already define them. I just did them in my own namespace to keep the example simple. Note that I did use a property from the NEPOMUK project to store each file's name.

Imagine an application that knows about the Dublin Core metadata standard, but doesn't know about these `dm:composer` and `dm:photographer` properties that I've defined.

When I run the following SPARQL query with a query engine that understands RDFS (in this case, Pellet), I'll have such an application, because the query can look for uses of the well-known Dublin Core property without having any mentions of the `dm:composer` and `dm:photographer` properties that I used in the data:

```
# filename: ex418.rq

PREFIX nfo: <http://www.semanticdesktop.org/ontologies/2007/03/22/nfo#>
PREFIX dc:  <http://purl.org/dc/elements/1.1/>

SELECT ?filename ?creator
WHERE
{
  ?resource nfo:fileName ?filename ;
            dc:creator ?creator .
}
```

Using ARQ to run this query against the ex417.ttl would get no results because ARQ would look for triples that use the `dc:creator` property, and it wouldn't find any. However, if you use a SPARQL processor that understands RDFS, or if you run a separate RDFS inferencing engine on the data to materialize the inferred triples and then run a SPARQL processor such as ARQ on the combined input and output of the inferencing engine, you'll get output like this:

```
filename      | creator
==================================
"204364.mp3"  | "Henry Fox Talbot"
"119933.mp3"  | "Diane Arbus"
"9894397.mp3" | "Charles Ives"
"884930.mp3"  | "Eric Satie"
```

Now that we have some background on how inferencing works, let's review two RDFS properties that we've seen a few times before:

`rdfs:domain`
: Tells you that if a given property is used as the predicate of a triple, then that triple's subject belongs to a particular class.

`rdfs:range`
: Tells you that if a given property is used as the predicate of a triple, then that triple's object is a member of a particular class.

In the example of these in "What Is Inferencing?" on page 254, we saw how an RDFS-aware processor could infer that if `ab:playsInstrument` has an `rdfs:domain` of `ab:Musician` and an `rdfs:range` of `ab:MusicalInstrument`, and Richard Mutt has an `ab:playsInstrument` value of `ab:vacuumCleaner`, then Richard (or rather, the resource `ab:i0432`) is a member of the class `ab:Musician` and `ab:vacuumCleaner` is an `ab:MusicalInstrument`. With an inferencing engine aiding the SPARQL query engine, we could then query for the first and last name of any musicians in the data, even if that data had no triples saying explicitly that any resources were members of the class `ab:Musician`.

We've seen how `rdfs:domain` and `rdfs:range` can indicate that certain resources are members of particular classes, but we haven't seen many ways of describing the classes themselves. The `rdfs:Class` class and the `rdfs:subClassOf` property give you a way to do this.

We haven't talked about these yet because, unlike most programming languages and modeling systems you may have used, RDF lets you say that something is a member of a class without first declaring that that class exists. When the example earlier in this chapter inferred that Richard Mutt was a member of the class `ab:Musician`, there was no need to first say "we have a class called ab:Musician that may have members." RDFS does offer a way to do this, though, and it's a good practice to use it, because data interoperability is much easier when everyone describes the kind of data that they're working with in a machine-readable way.

Let's look at an example. The first triple in the following Turtle file tells us that `ab:Musician` is a class. (The comment on that line shows how we could have substituted the keyword `a` for the property `rdf:type`, which is what the second-to-last line of the Turtle file does in a similar triple.) In English, this first triple says "`ab:Musician` is a member of the `rdfs:Class` class"—in other words, that `ab:Musician` is itself a class:

```
# filename: ex421.ttl

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

ab:Musician rdf:type rdfs:Class . # could have said a instead of rdf:type

d:i8301 rdf:type ab:Musician ;
        ab:firstName "Craig" ;
        ab:lastName  "Ellis" .

ab:Person a rdfs:Class .          # could have said rdf:type instead of a
ab:Musician rdfs:subClassOf ab:Person .
```

The next nonblank line in ex421.ttl tells us that resource `d:i8301` is a member of that class, and the two lines after that give us more information about this resource: its `ab:firstName` and `ab:lastName` values. The line before the file's last line tells us that, like `ab:Musician`, `ab:Person` is also a class.

The last key RDFS property for inferencing is `rdfs:subClassOf`, which does for classes what `rdfs:subPropertyOf` does for properties: it tells the inference engine to treat a given class as a specialized version of another. The last line in the Turtle file above tells us that the `ab:Musician` class is a subclass of `ab:Person`. This means that any member of the `ab:Musician` class is also a member of the `ab:Person` class—at least to an application that knows how to do RDFS inferencing.

To see this subclass reasoning in action, we'll run the following query against the ex421.ttl data with a SPARQL engine that understands RDFS:

```
# filename: ex422.rq

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

```
SELECT ?first ?last
WHERE
{
  ?person a ab:Person ;
          ab:firstName ?first ;
          ab:lastName ?last .
}
```

The query asks for the `ab:firstName` and `ab:lastName` values of any members of the `ab:Person` class, and while the RDFS-aware query engine doesn't see any explicit members of this class in the ex421.ttl data, it does see that `d:i8301` is a member of the `ab:Musician` class and that `ab:Musician` is a subclass of `ab:Person`, so it outputs that resource's first and last name values:

```
first   | last
=================
"Craig" | "Ellis"
```

Let's run the same query on the same data and add in some additional data from Chapter 2: ex045.ttl (which includes a triple telling us that Richard Mutt `ab:playsInstrument ab:vacuumCleaner`), and ex044.ttl (which tells us that the `ab:playsInstrument` property has an `rdfs:domain` of `ab:Musician`). The combination of data from these three files has no triples telling us that Richard is a member of any class, but when we run the query asking the combined triples from the three files about members of the Person class, it lists Richard along with Craig:

```
first     | last
===================
"Richard" | "Mutt"
"Craig"   | "Ellis"
```

The RDFS-aware SPARQL engine (in this case, Pellet) saw one triple saying that Richard plays the vacuum cleaner, another saying that this makes him a musician, and another saying that all musicians are persons, so he appeared on the list of persons.

I actually created the data about `ab:Musician` being a subclass of `ab:Person` about two years after I created the other data files, but this new metadata helped me to get more out of the original data files by making it possible to infer new information about Richard. My data "integration" was nothing more than the addition of two filenames to the same Pellet command line that I used when running the previous example.

This demonstrates some of the most important lessons of RDF technology:

- Integration of different sets of data is really, really easy, and if the triples from the different sources reference any of the same namespaces, you may find connections between the data that tell you new things about it.
- An inferencing engine is a great tool for finding these connections.
- Metadata doesn't have to be created with the data it describes; if you create it after the fact (perhaps years later!) it can still add value to that data retroactively.

SPARQL and OWL Inferencing

As we saw in the previous section, RDFS gives you properties to describe your own properties and classes, making it possible for an inferencing engine to create new triples based on those descriptions. OWL also does some of its job with specialized properties such as `owl:inverseOf` and `owl:sameAs`, but it's more common for OWL models to describe things by asserting that they're members of one or more (usually, more) classes.

For example, the following Turtle file, which we first saw in Chapter 2, includes a triple that says that spouse is a symmetric property by telling us that `ab:spouse` is a member of the class `owl:SymmetricProperty`:

```
# 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 .
```

As we also saw in that chapter, the file only includes two pieces of information about resource `ab:i9771`: an `ab:firstName` value and an `ab:lastName` value. An OWL processor, though, knows that if `ab:spouse` is an `owl:SymmetricProperty` and resource `ab:i0432` (Mr. Mutt) has an `ab:spouse` value of `ab:i9771`, then `ab:i9771` has an `ab:spouse` value of `ab:i0432`—or, in plain English, if Richard's spouse is Cindy, then Cindy's spouse is Richard.

Similarly, if resource `ab:i8301` (Craig Ellis) has an `ab:patient` value of Cindy and the `ab:patient` property is the `owl:inverseOf` the `ab:doctor` property, then Cindy's doctor is Craig.

How can we get a SPARQL query to retrieve the inferred information? If we use ARQ to run this next query with the ex046.ttl Turtle file, it won't find what this query is looking for because the resource that has an `ab:firstName` value of "Cindy" and an `ab:lastName` value of "Marshall" has no `ab:doctor` or `ab:spouse` values:

```
# filename: ex047.rq

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

SELECT ?doctorFirst ?doctorLast ?spouseFirst ?spouseLast
WHERE
{
   ?s ab:firstName "Cindy" ;
      ab:lastName "Marshall" ;
      ab:doctor ?doctor ;
      ab:spouse ?spouse .

?doctor ab:firstName ?doctorFirst ;
        ab:lastName ?doctorLast .

?spouse ab:firstName ?spouseFirst ;
        ab:lastName ?spouseLast .
}
```

If we run that query with the SPARQL engine built into the Pellet OWL reasoner, though, we get this result, because Pellet understands the meaning (the semantics!) of `owl:symmetricProperty` and `owl:inverseOf`:

```
doctorFirst | doctorLast | spouseFirst | spouseLast
===================================================
"Craig"     | "Ellis"    | "Richard"   | "Mutt"
```

In addition to `owl:SymmetricProperty`, classes such as `owl:TransitiveProperty` and `owl:InverseFunctionalProperty` are core parts of OWL that let you describe your own properties in ways that let you get more out of those properties and the resources that they describe.

OWL's heavyweight power comes from its ability to let you define classes whose purpose is to describe conditions about their members. Classes known as *restriction classes* essentially say that to be a member of one of these classes, a resource must meet certain conditions. If those conditions are true for a particular resource, then you know that the resource is a member of the class described by those conditions, and vice versa: if a resource is a member of a particular restriction class, then you know that any conditions used to define that class must be true about that resource.

It's common in OWL to describe lots of details about a particular resource by saying that the resource is a member of multiple restriction classes. For people coming from an object-oriented background, seeing a resource defined as a member of many different classes can be confusing, but remember: in OWL, many of these classes ultimately serve the purpose of describing resources, in effect serving a role played by properties or attributes in more traditional systems. The `owl:SymmetricProperty` class in ex046.ttl is one example of such a class, and as we'll see, you can define your own classes to perform a similar role.

For example, the next Turtle file names a few states of the United States and describes several musicians, the instrument they play, and the state that they're from. It also defines three classes: the first, `dm:Guitarist`, is defined using a restriction class as the set of all resources that have a `dm:plays` value of `d:Guitar`. With the data shown, an OWL-aware processor will say that there are three members of that class in the dataset.

The class `dm:Texan` is similar, defining its members as resources that have `d:TX` as their `dm:stateOfBirth` value. The third class, `dm:TexasGuitarPlayer`, is defined as the intersection of the first two sets:

```
# filename: ex424.ttl

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

d:CA rdfs:label "California" .
d:TX rdfs:label "Texas" .
d:NY rdfs:label "New York" .

d:m1 rdfs:label "Bonnie Raitt" ;
     dm:plays d:Guitar ;
     dm:stateOfBirth d:CA .

d:m2 rdfs:label "Charlie Christian" ;
     dm:plays d:Guitar ;
     dm:stateOfBirth d:TX .

d:m3 rdfs:label "Dusty Hill" ;
     dm:plays d:Bass ;
     dm:stateOfBirth d:TX .

d:m4 rdfs:label "Kim Gordon" ;
     dm:plays d:Bass ;
     dm:stateOfBirth d:NY .

d:m5 rdfs:label "Red Garland" ;
     dm:plays d:Piano ;
     dm:stateOfBirth d:TX .
```

```
d:m6 rdfs:label "Roky Erickson" ;
     dm:plays d:Guitar ;
     dm:stateOfBirth d:TX .

dm:Guitarist
   owl:equivalentClass
           [ rdf:type owl:Restriction ;
             owl:hasValue d:Guitar ;
             owl:onProperty dm:plays
           ] .

dm:Texan
   owl:equivalentClass
           [ rdf:type owl:Restriction ;
             owl:hasValue d:TX ;
             owl:onProperty dm:stateOfBirth
           ] .

dm:TexasGuitarPlayer
   owl:equivalentClass
        [ rdf:type owl:Class ;
          owl:intersectionOf (dm:Texan dm:Guitarist)
        ] .
```

The restriction class definitions use square braces because the details describing the restrictions are grouped together by a blank node. They don't have to be, but this is a common practice in OWL modeling.

There are no triples that say that any resource has an `rdf:type` of `dm:TexasGuitar Player`, so telling ARQ to run the following query against this data will get no results:

```
# filename: ex425.rq

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

SELECT ?name
WHERE
{
  ?musician a dm:TexasGuitarPlayer ;
            rdfs:label ?name .
}
```

Running an RDFS-aware processor that doesn't know about OWL with the same query and data won't do any better. An OWL-aware processor, though, will understand that the data defines `dm:TexasGuitarPlayer` as the equivalent of a class that is the intersection of two other classes: the class of all resources that have a `dm:plays` value of `dm:Guitar` and the class of all resources that have a `dm:stateOfBirth` value of "TX".

The syntax used to define OWL restriction classes can get complicated, so it's best to use a tool like TopBraid Composer or Protégé for this. Being standards-based tools, they'll still save your OWL class definitions using standard RDF/XML or Turtle syntax.

Using the OWL processor Pellet to run the `ex425.rq` query with the `ex424.ttl` data gives us this result:

```
name
==================
"Charlie Christian"
"Roky Erickson"
```

In addition to `owl:hasValue` and `owl:intersectionOf`, other available OWL properties for defining restriction classes include `owl:allValuesFrom` and `owl:someValuesFrom`, which indicate that a resource must have all or some values from a particular class, and `owl:minCardinality` and `owl:maxCardinality`, which specify the minimum and maximum number of values for a given property that a class's members must have. (For example, you could use `owl:minCardinality` to specify that members of the restriction class `dm:BasketballTeam` must have a minimum number of five `dm:player` property values.)

The combination of the `dm:Guitarist` and `dm:Texan` restriction classes into the `dm:TexasGuitarPlayer` class is a simple example of how such classes can be combined into new ones. Advanced OWL users, especially in areas like life sciences, make much more extensive use of this ability to combine existing restriction classes into new ones, to the point where the inferencing about which resources belong in these specialized new classes may take a computer several hours. It's worth it, though, because no other standardized modeling technology makes this possible. It's nice to know that SPARQL can take advantage of this.

This power, and its potential demand on computing resources, led the W3C to define three profiles of the original OWL specification so that OWL could satisfy the sometimes competing requirements of Working Group members who wanted formal rigor in their automated reasoning and members who wanted to write applications that could get their work done in a reasonable amount of time:

OWL Lite
: Doesn't allow as many ways to describe your data as DL and Full, but it's easier to implement. It never became very popular.

OWL DL
: Much more powerful than OWL Lite, while omitting a few bits of OWL Full to prevent OWL engines from getting stuck when computing all the potential inferences of a given set of triples. DL stands for Description Logics, an alternative to the network- and frame-based approaches to knowledge representation that became popular in the early heyday of artificial intelligence in the 1970s. Description

Logics focused more often on specific domains, making the job of modeling knowledge less open-ended and therefore easier to accomplish. Pellet is an OWL DL reasoner.

OWL Full
: The most expressive of the three options, letting you model things that developers of OWL processors were hoping not to see because the processor could theoretically be cranking away forever.

OWL DL became the most popular. When the W3C published OWL 2, in order to make it easier for OWL engine implementers to accommodate certain sets of customers, the OWL Working Group came up with an alternative to the Lite/DL/Full distinction, publishing three profiles optimized for specific classes of applications:

OWL 2 RL
: The subset of OWL that can be implemented by a rules language. These languages include Prolog, a grand old programming language of the artificial intelligence world, and—as we'll see later—SPARQL, which was heavily influenced by Prolog. This makes RL easier to implement than the other two OWL 2 profiles and lets it cover most typical OWL needs.

OWL 2 EL
: Popular for large-scale versions of the `dm:TexasGuitarPlayer` example: complex combinations of class definitions that let you find out which resources fall into which classes. According to the W3C OWL 2 Web Ontology Language Profiles document, EL "is particularly suitable for applications employing ontologies that define very large numbers of classes and/or properties."

  Remember, with RDF modeling, a given instance can fall into multiple classes, so if your model describes the properties and functions of various chemical compounds, and you've used restriction classes to define a new class of potential solutions to a given problem, an OWL reasoner can help find which compounds may be solutions to the specified problem. This kind of application is popular with drug discovery and other biomedical fields.

OWL 2 QL
: Designed to query datasets with large amounts of instance data, especially when stored in relational databases, because the OWL features chosen for QL were selected to more easily automate the generation of SQL queries. According to the Web Ontology Language Profiles document, QL provides "many of the main features necessary to express conceptual models such as UML class diagrams and ER diagrams," which will also appeal to developers working with large relational systems.

Which is best for you? The short answer is, don't worry about it until your application grows large enough that performance is an issue. Find out which profile or profiles your OWL reasoner supports and what its providers say about why; this can provide some further guidelines.

Of course, the actual W3C OWL specification documents are the ultimate guide to what each profile does and doesn't support. These are usually described in terms of what OWL properties each profile does not support (for example, OWL 2 EL does not allow the use of `owl:maxCardinality`) and what limitations exist for certain properties. Generally, though, the specific restrictions on each profile are about more obscure OWL features that won't be an issue to anyone but advanced OWL modelers.

Using SPARQL to Do Your Inferencing

What do you think the following query would do with the ex046.ttl dataset that we saw earlier?

```
# filename: ex427.rq

PREFIX  owl:  <http://www.w3.org/2002/07/owl#>

CONSTRUCT
{ ?resource2 ?property1 ?resource1 . }
WHERE
{
   ?property1 owl:inverseOf ?property2 .
   ?resource1 ?property2 ?resource2 .
}
```

The only `owl:inverseOf` triple in that data tells us that the `ab:doctor` property is the inverse of the `ab:patient` property, so the WHERE clause here will bind `ab:doctor` to the `?property1` variable and `ab:patient` to the `?property2` variable.

Because of that last binding, the predicate of the second triple pattern will be `ab:patient`. Because resource `ab:i8301` (Craig Ellis) has an `ab:patient` value of `ab:i9771` (Cindy Marshall), the query's WHERE clause will bind `ab:i8301` to the `?resource1` variable and `ab:i9771` to the `?resource2` variable. The CONSTRUCT clause will take the bindings in the second triple pattern and creates a new triple saying that resource `ab:i9771` has an `ab:doctor` value of `ab:i8301`.

Or, to make a long story short, if `ab:patient` is the inverse of `ab:doctor` and Craig has Cindy as an `ab:patient`, the query creates a triple saying that Cindy has Craig as an `ab:doctor`. The query has implemented some inferencing, in the RDF sense of the term: based on the patterns it found, it created a new triple that made the implicit `ab:doctor` relationship between Cindy and Craig explicit.

The SPARQL Inferencing Notation (SPIN) specification submitted to the W3C by TopQuadrant, OpenLink Software, and Rensselaer Polytechnic Institute takes this idea much further by making it possible to store SPARQL-based inferencing rules like this with class definitions. TopQuadrant products (including the free version of their modeling software, TopBraid Composer) include rule sets that use SPIN to implement their RDFS+ superset of RDF, OWL 2 RL, and several other sets of modeling rules.

So, if you're using a toolset that includes a SPIN implementation, this is another option for performing inferencing, but it brings up another point about a big contribution that SPARQL can make to your application development: if you need certain kinds of specialized inferencing (once again, the creation of new triples based on existing patterns), you may not even need RDFS or OWL. You may be able to use one or more SPARQL queries to generate exactly what you need using a simple SPARQL engine that has no knowledge of the special instructions defined by these metadata standards.

As an example, look at the following simple query. If a resource has a `dm:stateOfBirth` value of `d:TX` and a `dm:plays` value of `d:Guitar`, the CONSTRUCT clause creates a triple saying that this resource is a member of the `dm:TexasGuitarPlayer` class:

```
# filename: ex428.rq

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

CONSTRUCT
{ ?resource a dm:TexasGuitarPlayer }
WHERE
{
  ?resource dm:stateOfBirth d:TX ;
            dm:plays d:Guitar .
}
```

When you use ARQ to run this query with the ex424.ttl dataset that we saw in "SPARQL and OWL Inferencing" on page 263, it creates triples saying that resources `d:m2` (Charlie Christian) and `d:m6` (Roky Erickson) are members of this class:

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

d:m2
      rdf:type      dm:TexasGuitarPlayer .

d:m6
      rdf:type      dm:TexasGuitarPlayer .
```

When we used an OWL processor to run the ex425.rq query against this data earlier, it used the restriction classes defined in this data to infer that these two musicians were members of the `dm:TexasGuitarPlayer` class. This time, we used ARQ and a query that ignored the restriction class definitions. (The ex425.rq query didn't explicitly reference all the class definitions, but the OWL processor saw that the query asked about members of the `dm:TexasGuitarPlayer` class and had to look at all three restriction class definitions to see which resources would qualify as members of that class. When running query ex428.rq, ARQ had no idea what to do with those class definitions.)

So, we've seen one way to find the guitar-playing Texans by using OWL expressions (which, as we also saw, can get pretty complicated) and a processor that understands OWL, and we've seen another way to find these musicians using a simple SPARQL 1.0 query and a SPARQL engine that knows nothing of OWL. The latter case didn't use a SPARQL query to implement an OWL keyword's logic, as the ex427.rq query did for the `owl:inverseOf` property; it just used a SPARQL query to do something that might otherwise have been done with OWL.

While RDFS and OWL open up many new opportunities to get value out of your data, SPARQL alone can often do the pattern matching necessary to infer the triples that your application needs. This can give you more flexibility in your application architecture, because the choice of available processors that understand RDFS and OWL is much more limited than the choice of available SPARQL engines.

A SPARQL-based alternative to using RDFS or OWL for inferencing isn't always best. These two standards provide a foundation for data modeling that can scale way up, making them the better choice for many large-scale applications. It's good to know, though, that some of the things they let you do can often be done faster, more quickly, and with a wider choice of commercial and free software when you do your pattern matching with pure SPARQL.

Querying Schemas

No discussion of SPARQL, RDFS, and OWL would be complete without mentioning one particular advantage of how all of these RDF-related standards fit so well together: because the schemas are expressed with the same syntax and data model as the data itself, we can use the same query language with the schemas that we use with the data. This makes it easier to get to know a schema and to plan future queries and applications.

For example, the Friend of a Friend (FOAF) ontology includes a `foaf:page` property that people use to name a web page about a particular subject. If I know that FOAF also defines a few specialized versions of this property, but I don't know exactly what they are, I can apply the following query to the FOAF ontology (which can be downloaded from the FOAF project's home page) to list these subproperties:

```
# filename: ex430.rq

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

SELECT *
WHERE
{ ?subproperty rdfs:subPropertyOf foaf:page . }
```

The answer shows us four:

```
-------------------------
| subproperty           |
=========================
| foaf:isPrimaryTopicOf |
| foaf:tipjar           |
| foaf:weblog           |
| foaf:homepage         |
-------------------------
```

Do any of the subproperties have their own subproperties? Using the SPARQL 1.1 property paths feature, we can add a single plus sign to the above query to find all the descendant subproperties of `foaf:page`. Adding one more triple pattern tells it to include the direct parent of each property in the output, giving us a better idea of that property's place in the subproperty tree:

```
# filename: ex432.rq

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

SELECT *
WHERE
{
  ?subproperty rdfs:subPropertyOf+ foaf:page ;
               rdfs:subPropertyOf ?parentProperty .
}
```

Here is the result:

```
-------------------------------------------------
| subproperty           | parentProperty        |
=================================================
| foaf:isPrimaryTopicOf | foaf:page             |
| foaf:openid           | foaf:isPrimaryTopicOf |
| foaf:homepage         | foaf:page             |
| foaf:homepage         | foaf:isPrimaryTopicOf |
| foaf:tipjar           | foaf:page             |
| foaf:weblog           | foaf:page             |
| foaf:homepage         | foaf:page             |
| foaf:homepage         | foaf:isPrimaryTopicOf |
-------------------------------------------------
```

FOAF defines a `foaf:Person` class and uses the `rdfs:domain` property to associate several properties with it. Which properties?

```
# filename: ex434.rq

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

SELECT ?property
WHERE
{ ?property rdfs:domain foaf:Person . }
```

This simple query shows us an interesting range of properties that are associated with the `foaf:Person` class:

```
--------------------------
| property               |
==========================
| foaf:schoolHomepage    |
| foaf:plan              |
| foaf:myersBriggs       |
| foaf:workplaceHomepage |
| foaf:family_name       |
| foaf:lastName          |
| foaf:geekcode          |
| foaf:workInfoHomepage  |
| foaf:firstName         |
| foaf:surname           |
| foaf:familyName        |
| foaf:knows             |
| foaf:pastProject       |
| foaf:publications      |
| foaf:currentProject    |
| foaf:img               |
--------------------------
```

The more you know about RDFS and OWL features for adding metadata to RDF datasets, the more you can use SPARQL to learn about a given schema or ontology.

Chapter 11 has several good queries for exploring schemas and ontologies.

Summary

In this chapter we learned:

- How inferencing with RDF applications is about deriving new triples from existing triples and the metadata that we know about those triples (especially metadata about the triples' predicates), and how this can add to your applications
- How RDFS inferencing lets you infer resource class membership based on `rdfs:domain` and `rdfs:range` values
- How OWL offers additional properties and classes to let us infer new triples, as well as the ability to define customized metadata classes called restriction classes
- How SPARQL's pattern-matching abilities let queries infer new triples without any help from the specialized RDFS and OWL processors used to do inferencing earlier in the chapter
- How SPARQL lets you learn more about specific schemas and ontologies because, since schemas and ontologies are themselves triples, you can use all of your SPARQL skills to query them

CHAPTER 10

Building Applications with SPARQL

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 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 rows of results are sorted according to the values of one or more of the columns. Upon receiving these results, the client processes use this 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 reserve a hotel room both fit into this scenario. Perhaps the clothing and hotel data are stored in relational databases, and SQL queries are being sent between the processes assembling the information that 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 browsing for T-shirts or reserving hotel rooms don't (and shouldn't have to) care. They just want the right 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 and update information. One benefit of this approach is the flexibility you gain from the ability to treat different data formats as triples even if they weren't designed that way—for example, by using a middleware layer that lets you send SPARQL queries to relational databases. Because of this, there doesn't necessarily need to be a triplestore in the 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.

Chapter 8 has additional helpful information about application development with SPARQL. It explains ways to incorporate the standardized XML, JSON, CSV, and TSV versions of SPARQL query results into applications that can read these formats—and that's a lot of applications.

All that being said, SPARQL can bring more to application development than its value as a standardized syntax for retrieving data that conforms to a flexible, standardized data model. Some RDF-based applications (and application development tools) can bring extra power when they see certain kinds of RDF, so we'll look at examples of those as well.

In this chapter, we'll take a look at how several SPARQL-related technologies can contribute to different parts of this picture:

"Applications and Triples" on page 277
: Sometimes predicates are more than just descriptions of relationships; they can be instructions to perform certain jobs.

"SPARQL and Web Application Development" on page 282
: A program or web form can send a query to a SPARQL endpoint via HTTP and use the returned results with very little code.

"SPARQL Processors" on page 291
: 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 HTTP" on page 295
: The SPARQL Graph Store HTTP Protocol protocol describes a RESTful way for applications to retrieve, insert, add to, and delete named graphs of triples in a triplestore. This method doesn't use queries, but it is part of the SPARQL family of standards.

Applications and Triples

In Chapter 9, we learned that while an RDF parser can read the following example and know where the subjects, predicates, and objects of each triple are, an RDFS-aware processor can do more: it understands that the triple with a predicate of `rdfs:subPropertyOf` is an instruction telling it to treat any values of `skos:prefLabel` as if they were also `rdfs:label` values. (This particular triple is actually part of the SKOS standard.) In other words, it tells the processor to explicitly or implicitly generate the triple that is commented out in this example:

```
# filename: ex521.ttl

@prefix d:    <http://learningsparql.com/ns/data#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

skos:prefLabel rdfs:subPropertyOf rdfs:label .
d:i4230 skos:prefLabel "Frank" .
# d:i4230 rdfs:label "Frank" .
```

An OWL engine is an RDF parser that understands instructions in the OWL namespace. Like `rdfs:subPropertyOf`, OWL instructions such as `owl:SymmetricProperty` let us describe properties to tell us more about them, and an OWL engine typically acts on those instructions by generating more triples. Because RDFS and OWL engines can do this, they are themselves RDF applications, acting on instructions within a set of triples to help you get more out of the data in those triples. These are sometimes combined with SPARQL processors (for example, in the OWL engine Pellet), adding this inferencing power to your queries, which can add a lot to the applications you develop.

You can also think of your own SPARQL CONSTRUCT queries and UPDATE and DELETE requests as applications (perhaps not complete applications, but potentially very valuable components of larger applications) because they take actions based on the presence of specific subject, predicate and object values.

Property Functions

An interesting variation on the theme of applications taking specific actions when they see certain predicates is the idea of *property functions*, also known as functional predicates or magic properties. While the "Extension Functions" on page 182 are implementation-specific functions that can be used in place of or along with the functions defined as part of the SPARQL standard (for example, in FILTER statements, or when building values to assign to a variable with a BIND statement), a property function is used as part of a triple pattern, taking the place of the predicate and often of the object as well. Instead of matching against triples in the data the way a normal predicate would, they instruct applications that understand them to execute specific tasks.

Sometimes, property functions use passed parameters or values matched against the subject or object as input, and sometimes they bind values to variables in these positions.

For example, the following uses an ARQ property function to find out which release of ARQ is being used:

```
#filename: ex524.rq

PREFIX apf: <http://jena.hpl.hp.com/ARQ/property#>

SELECT ?o
WHERE
{ ?s apf:versionARQ ?o }
```

With this property function, regardless of the input data used with the query, the result binds the variable in the triple pattern's object position to a string describing the release of ARQ used to run the query:

```
--------------------
| o                |
====================
| "2.9.4-SNAPSHOT" |
--------------------
```

Another example of an implementation-specific property function is the TopBraid `tops:for` function, which creates integers in the range specified. In the following query, the property function tells the query engine to generate integers from 3 to 6, and the BIND statement after it multiples each value by 100 and stores the result in the `?mult` variable:

```
# filename: ex526.rq

PREFIX tops: <http://www.topbraid.org/tops#>
SELECT *
WHERE
{
  ?index tops:for(3 6)
  BIND(?index*100 AS ?mult)
}
```

The result has four rows, one for each integer generated:

```
[index] mult
3       300
4       400
5       500
6       600
```

An interesting application that relies heavily on property functions is geosparql.org, which lets you query the geospatial Linked Data stored at that site. GeoSPARQL is a standard for representing this kind of data, and the site (whose owners KONA are not affiliated with the group behind the standard) lets you enter SPARQL queries that can use several property functions designed specifically for this application.

For example, the following query asks for the name and population values of places near Stonehenge, which has the latitude and longitude coordinates 51.1789 and -1.8264:

```
# filename: ex528.rq

PREFIX co: <http://www.geonames.org/countries/#>

SELECT  ?name ?population
WHERE
{
  ?place gs:nearby(51.1789 -1.8264 10) ;
         gn:name ?name ;
         gn:population ?population .
}
```

Running this query on the geosparql.org website gives us this result:

```
----------------------------
| name         | population |
============================
| "Salisbury"  | "45600"    |
| "Andover"    | "39951"    |
| "Warminster" | "17875"    |
| "Romsey"     | "17773"    |
| "Trowbridge" | "36922"    |
| "Chippenham" | "36890"    |
----------------------------
```

(The third parameter of `gs:nearby` is the "limit" parameter, a number that tells the searching algorithm how far to look. This number doesn't represent miles or kilometers, but something specific to the search algorithm. Still, the bigger the number, the further the function will look, and the more results you'll get.)

When querying geosparql.org with the `gs:nearby` property function, asking for the `owl:sameAs` value of each resource place returned gives you its DBpedia URI—for example, *http://dbpedia.org/resource/Trowbridge* for the town of Trowbridge. This lets you look up lots of additional information about each place, making it a nice example of Linked Data in action.

Model-Driven Development

Model-driven development has been a growing trend in application development in recent years, especially among developers who understand the advantages of RDF and the related standards. Before discussing it, let's take a look at what it improves on.

For about as long as computers have been around, software has been developed by first modeling a system's components and their relationships (in the early days, by drawing flowcharts on graph paper using pencils and plastic templates; later, using software tools built around standards like UML) and then using those plans as guidelines for the actual coding. Once the software was built and deployed, the original plans became an artifact of the early stages of the system's development. If someone revised or updated the system, the plans may have been updated to reflect this, but only as a documentation step, and often not at all.

In model-driven development, the original model becomes a part of the deployed system. To change the application, you often only need to change the model, and the change is reflected in the application—or applications, because the model may play a role in multiple applications.

Let's look at an example that demonstrates why the RDF Schema language makes this easier. In the section "Reusing and Creating Vocabularies: RDF Schema and OWL" on page 36, we learned how we can use the `rdfs:domain` property from the RDF Schema vocabulary to say that if a given property is used as the predicate of a triple, then that triple's subject belongs to a particular class. In "What Is Inferencing?" on page 254, we saw that if the `ab:playsInstrument` property has a domain of `ab:Musician` (which we can state with the triple `{ab:playsInstrument rdfs:domain ab:Musician}`) and resource `ab:i0432` has an `ab:playsInstrument` value of `ab:vacuumCleaner`, then we can infer that resource `ab:i0432` is a member of the class `ab:Musician`. We also saw that if a property has an `rdfs:range` value, then the object of a triple using that property belongs to a certain class. For example, if we have the triple `{ab:playsInstrument rdfs:range ab:MusicalInstrument}`, we know from the earlier triple about `ab:80432` that `ab:vacuumCleaner` is a member of the class `ab:musicalInstrument`.

The `rdfs:domain` and `rdfs:range` properties can also play a role in a model-driven system. For example, let's say your model includes the following information about the `ab:Person` and `ab:MusicalInstrument` classes:

```
# filename: ex536.ttl

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

ab:Person a owl:Class .
ab:MusicalInstrument a owl:Class .

ab:firstName rdfs:domain ab:Person ;
             rdfs:label "first name" .
ab:lastName  rdfs:domain ab:Person ;
             rdfs:label "last name" .
ab:homeTel   rdfs:domain ab:Person ;
             rdfs:label "home telephone" .
ab:email     rdfs:domain ab:Person ;
             rdfs:label "email address" .

ab:plays rdfs:domain ab:Person .
ab:plays rdfs:range ab:MusicalInstrument .
```

```
ab:guitar a ab:MusicalInstrument ;
          rdfs:label "guitar" .
ab:bass   a ab:MusicalInstrument ;
          rdfs:label "bass" .
ab:drums  a ab:MusicalInstrument ;
           rdfs:label "drums" .
```

Your application is going to pop up a form where users can enter data about new instances of the `ab:Person` class. You could just create this form in an HTML file or some other form display tool, but what happens when the `ab:Person` model is revised to include an `ab:mobileTel` property? You would have to go back and edit the HTML file to account for the new property, and probably update a lot of other program logic as well.

On the other hand, you could dynamically generate the form from the results of this SPARQL query, which asks for the properties associated with the `ab:Person` class:

```
#filename: ex537.rq

PREFIX ab:   <http://learningsparql.com/ns/addressbook#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?property ?propName
WHERE
{
  ?property rdfs:domain ab:Person ;
            rdfs:label ?propName .
}
```

If you asked your SPARQL engine for the results of this query in XML (described in "SPARQL Query Results XML Format"on page 238), a little XSLT could then generate the form where your application's users enter new instances of `ab:Person`. (If your favorite tools make JSON easier to use than XML, you can ask for the query results in that format.) If the model is later changed to associate the new `ab:mobileTel` property with the `ab:Person` class, the new version of the dynamically generated data input form will include the new property, and no one needs to edit any form files.

The example model includes `rdfs:label` values, and the SPARQL query retrieves them, because human-readable names are always important for applications that use this data and metadata. Unless your end users are RDF geeks, you don't want them to see prefixed names or URIs on your application's screens. In this case, the application would use the `rdfs:label` values to label the fields on the data entry form.

The `rdfs:range` property can also drive application logic. Because `ab:plays` has a `rdfs:domain` of `ab:Person`, a data entry form generated using the query above will include a "plays" field, and we can take it a little further. While your form could let users enter any value they want there, it could also limit them to a specific set of values, and the set is already part of the model: `ab:plays` has an `ab:range` value of ab:MusicalInstrument, and the ab:MusicalInstrument class has three instances. Your form could include these values in a drop-down list, and if someone eventually adds ab:piano as a new instance of this class, the SPARQL query that generates that part of the form will pass the new instrument name (most likely, the rdfs:label value associated with ab:piano) along to be added to that drop-down list.

While these examples describe the use of rdfs:domain and rdfs:range values to generate a data entry form, the dynamic use of model values can go far beyond that. Using similar techniques, this metadata can help your application format reports; it can even play a role in the business logic and rules that drive your application. Whatever you do with it, SPARQL is the simplest way to get those values from your model to your application. The use of RDFS and OWL can let you take these ideas even further.

This idea of model-driven development and deployment is not limited to the use of the RDF family of specifications, but the fact that these specifications are standards with a range of commercial and open source implementations makes it an attractive choice for many developers using this methodology.

SPARQL and Web Application Development

In "Applications and Triples" on page 277, we learned about SPARQL engines and related RDF processors that perform special instructions based on the subjects, predicates, and objects (mostly, predicates) that they find in a set of triples. While these can be valuable components of an application, in this section we'll look at the bigger picture of application development using SPARQL: tying together such components with the other parts you need to create a complete application.

"Web application development" typically means two things: using web-based technologies to create an interface and taking advantage of web-based data sources. Later in this section, we'll see an example of how web forms and dynamically generated HTML can build an interface around data retrieved using SPARQL; first, though, lets look more closely at how such an application can use SPARQL to retrieve that data.

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 by sending 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 that answer 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 URL 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
```

Pasting that URL 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 URL as a parameter and save the returned results in a file. (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.) Because some programs that read input from files will also let you specify the data to read with a URL instead of a local file name, you may be able to specify the data for these programs to read with a URL like the one above that sends a query to an endpoint. For example, you can use a URL that retrieves SPARQL Query Results XML Format 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.

Here's a Python script that escapes the Elvis birthday query, builds the URI above, and then uses it to retrieve the result:

```
# 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 described in Chapter 8:

```
<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—for example, the addition of code to recover gracefully if the remote endpoint is down. These developers 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 native data structures for 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, the dotNetRDF library for C# , and ARQ 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.) The script requests the results in the 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
```

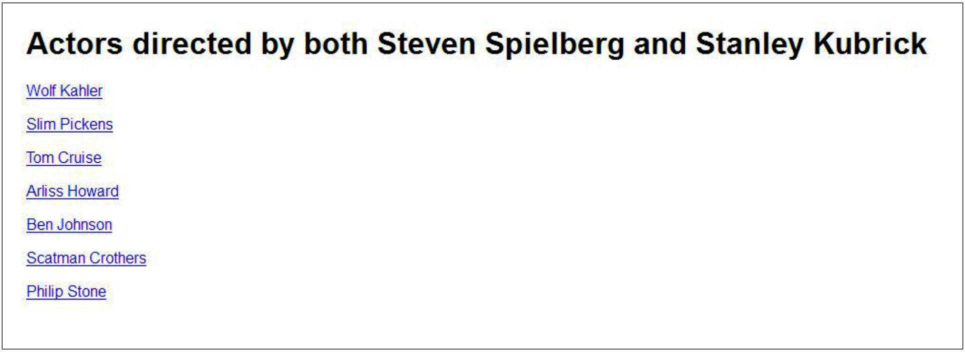
Creating this kind of list would be very time-consuming without SPARQL and a collection of data that let you search the connections between actors, films, and directors. If you want to create a similar actor list for other pairs of directors, just replace Spielberg and Kubrick's names and run the script 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 simply clicking a button?

With a few revisions to the Python script, you can create this application. 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 10-1.


Actors directed by both Steven Spielberg and Stanley Kubrick

Wolf Kahler

Slim Pickens

Tom Cruise

Arliss Howard

Ben Johnson

Scatman Crothers

Philip Stone

Figure 10-1. Web page created by ex363.py Python script as displayed by browser

The use of something like the `replace()` function can make some applications vulnerable to the injection attacks that are sometimes used to hack SQL-driven websites. It's less of an issue here, because this isn't an update query, but it's worth checking your SPARQL programming language library for functions or classes such as ARQ's ParameterizedSparqlString class that make this easier to avoid.

For most of the Web's history, CGI (Common Gateway Interface) scripts have been the most common way to dynamically parse and generate content. The final version of our movie director Python script adds the following to ex363.py to make it a CGI script:

- 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 script in a hosted environment.
- It takes values passed to it in `dir1` and `dir2` variables and stores them in the Python variables `director1` and `director2`.
- After preparing the SPARQL query for delivery to the endpoint 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 plus signs for any spaces (or `%20`, which I used in the URL version of the query about Elvis's birthday) as shown:

```
http://learningsparql.com/examples/ex364.cgi?dir1=John+Ford&dir2=Howard+Hawks
```

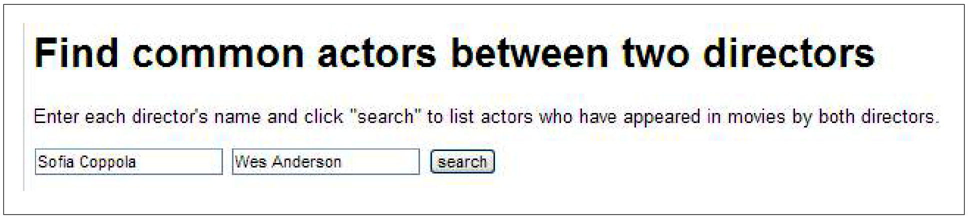
The final step is to create the web form that your film buff friends will fill out to list the actors that two directors have in common. This is all you need:

While there is a copy of ex364.cgi in *http://www.learningsparql.com/examples*, because that's not configured as a CGI directory, the URL will not execute that copy.

```
<!-- 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 someone clicks the search button, 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 10-2. Figure 10-3 shows the result of clicking the search button in Figure 10-2.

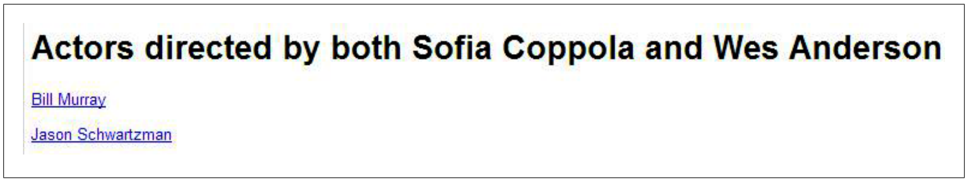


Find common actors between two directors

Enter each director's name and click "search" to list actors who have appeared in movies by both directors.

Sofia Coppola | Wes Anderson | search

Figure 10-2. 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 10-3. Results of clicking "search" in Figure 10-2

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 take advantage of 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 a client process that sends a request to the script. With the results of your SPARQL query loaded into the typical data structures of your favorite development language or returned as JSON or XML, you can 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.

SPARQL Processors

The previous section discussed ways to send a query off to a SPARQL endpoint, wherever it may be, and then use the result in an application. In this section, we'll look more closely at the roles that three classes of SPARQL processors might play in your applications: processors that function as a standalone program, those that are a built-in feature of a triplestore, and those that provide a middleware layer to let 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 immediately see the results of running the query against that data. There are no setup or configuration steps necessary to get a server up and running.

When you do set up a server that can accept SPARQL requests and return the results, the extra steps are often worth it when you're assembling a more complex application, but before discussing these, it's worth saying more about how you can use ARQ 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, though, you can still use the standalone ARQ binary used with this book's examples as part of a simple application. For example, we saw in Chapter 8 how it can deliver your query result in SPARQL Query Results XML Format, JSON, and comma- and tab-separated output formats, so you can use ARQ to feed data to application components that understand these 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 take advantage of ARQ's ability to output comma- and tab-separated values.

This approach may not result in the fastest, most robust and scalable application, but it's great for prototyping when you're evaluating data that you may be working with.

Triplestore SPARQL Support

The ARQ command-line tool must read all of the data that you're going to query into memory before it can run your query against it. You're more likely to be querying a larger set of data that is more sensibly stored using 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 other languages in order to let other application components communicate with it efficiently, but for standards support and quick development, it should support not only the SPARQL query language, but the other relevant SPARQL specifications as well: SPARQL Update, the SPARQL Protocol, the Graph Store HTTP Protocol, and the various query results formats. (Whether the triplestore exposes its support for the

Update language and data-altering parts of the Graph Store HTTP Protocol to users is a decision for its site administrator.)

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 URLs. 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 URL 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 one of the SPARQL query results formats, as shown in Figure 10-4.

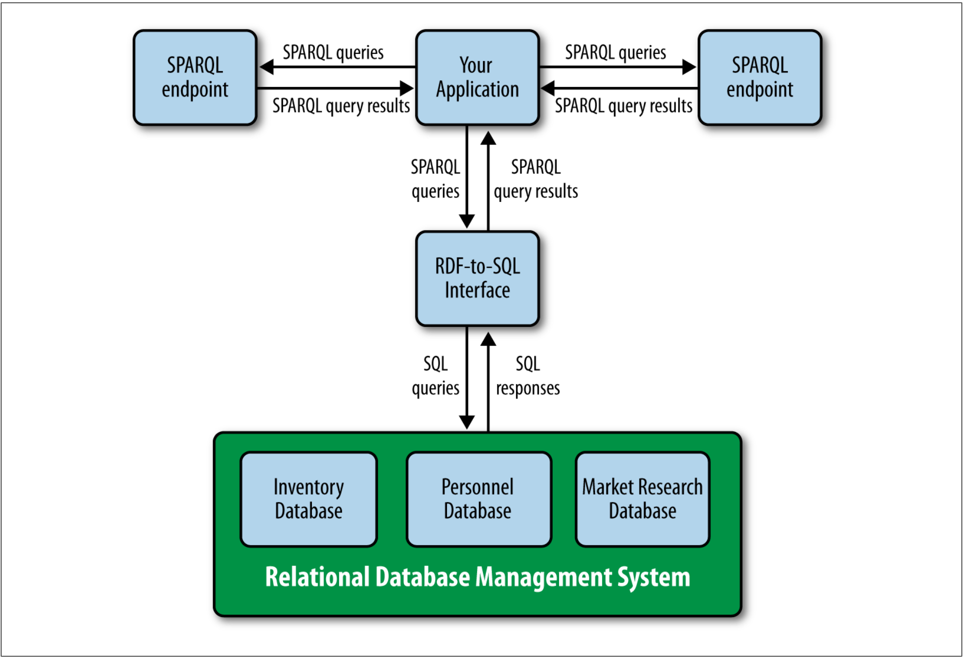


Figure 10-4. Relational database SPARQL endpoint middleware lets your application communicate with relational databases as if they were SPARQL endpoints

If you've followed along with the examples in this chapter, then you've already used one of these combinations of SPARQL middleware and an RDBMS without knowing it: the Linked Movie 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 that database's 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—for example, one stored using MySQL and another using Oracle—a single SPARQL query can ask for data from these 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. Since then, the W3C (with input from the D2RQ team) has released R2RML, a standardized mapping to improve consistency between the use of different relational systems in different RDF application environments. Several implementations are already available.

The Oracle Corporation's most well-known products are relational database managers, and plenty of RDF 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 the Oracle Spatial add-on, which optimizes processing of large amounts of location data.) The more recent Oracle NoSQL Database product, which is separate from 11g, includes an "RDF Graph" feature with SPARQL 1.1 support.

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, XML, HTML, and other formats. They also typically coordinate multiple sources in different formats to appear as a single source to someone (or something) sending them 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 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 host these applications on the same servers used for those 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 a single source is making this technology 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.

A simple, low-cost way to mix and match data from different silos (whether this data is stored as triples or not) opens up great new possibilities in any enterprise, especially when you can retrieve 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.

SPARQL and HTTP

We can't talk about using SPARQL in application development without talking about a new specification in SPARQL 1.1: the Graph Store HTTP Protocol, which describes ways to add, delete, and modify graphs of triples with HTTP commands.

When Tim Berners-Lee invented the World Wide Web, along with writing the first web browser and web server, he and his team wrote early drafts of three specifications that would let the browsers and servers work with each other: URLs, which let browsers express which documents they wanted to retrieve; HTML, a markup language that let browsers know which blocks of text were titles, subtitles, lists, and regular paragraphs; and HTTP, the Hypertext Transfer Protocol.

A computer communications protocol is a language that lets computers say things to each other like "hey, send this resource to me" and "OK, here you go" or "I have no idea what you're talking about." Of course, the language is terser—the HTTP way to say "I have no idea what you're talking about" is the error code 404, which we've all seen in our browsers after misspelling a web address.

These days, with web servers serving up much more than web pages, we can think of them more broadly as HTTP servers, because they deliver a range of resources using the HTTP protocol. Other client programs besides web browsers (for example, the wget and curl utilities mentioned earlier in this chapter, not to mention programs you can write with just about any programming or scripting language) can make HTTP requests, giving you some nice application development options.

Of the HTTP requests that clients can make to servers, the four most important are PUT, GET, POST, and DELETE, which correspond to the four "CRUD" operations that people need to execute with databases: Create new data, Read existing data, Update data, and Delete data. (For example, SQL offers the commands INSERT, SELECT, UPDATE, and DELETE to perform these tasks.) Much of the popular application architecture style known as REST is built around the PUT, GET, POST, and DELETE operations, and the SPARQL Working Group defined a way to let us use these with graphs of triples. According to the abstract of an earlier draft of the SPARQL 1.1 Graph Store HTTP Protocol Recommendation, it "describes the use of HTTP operations for the purpose of managing a collection of graphs in the REST architectural style."

The final line of the the Recommendation version of this specification, under "Changes since Last Call," says "Removed reference to REST." Why? My guess is that they wanted to steer clear of the sometimes energetic arguments over what qualifies as a proper example of a Representational State Transfer architecture and what doesn't.

As this Recommendation points out, in addition to the techniques described there, you can perform most if not all of the operations that it describes with a query sent to a SPARQL endpoint instead of with a specialized HTTP request. Hardcore RESTafarians who might not consider the SPARQL HTTP Graph Store Protocol to be 100% RESTful will still prefer it, with its use of basic HTTP operations with URIs that name resources (in this case, graphs of triples), over the more implementation-detail-oriented practice of embedding complete SPARQL queries in URLs.

Adding and deleting triples at the named graph level of granularity (as opposed to the triple level) also makes more sense for data publishing workflows in which sets of data—probably with their own metadata, covering topics such as provenance—are added and deleted as a unit. For example, if you're a data publisher and I'm one of your providers, I would send you a set of data to replace the current set that you're offering from my organization, which you may have distinguished from the other data offerings in your triplestore by keeping my organization's data in its own named graph.

We're going to see how a triplestore that supports the SPARQL Graph Store HTTP Protocol lets you do the following with the four central HTTP commands:

- GET the triples from the default graph or a named graph
- PUT a set of triples into a new graph in the triplestore, replacing any existing graph with the same name if it exists
- POST some triples to an existing graph, adding them to any existing triples there
- DELETE a graph of triples

We'll send these commands to the open source Fuseki SPARQL server (See Chapter 6 for information on getting and installing Fuseki) using the curl command-line utility mentioned earlier in this chapter, but variations on these commands should also work with the similar wget utility. More importantly, variations should work with the

HTTP GET, PUT, POST, and DELETE capabilities available in most programming languages, either natively or with the inclusion of an appropriate library.

When you send HTTP requests to a server, the first thing you need to know is what URL that server offers to support these requests, so check the documentation of the triplestore or middleware where you'll send your SPARQL HTTP requests. For the following Fuseki examples, I'll send requests to *http://localhost:3030/myDataset/data*, where *myDataset* is a dataset I created on Fuseki for these experiments.

The idea of named datasets are a Fuseki detail and not part of the SPARQL standard. The Sesame triplestore, which is described a bit in "Querying Named Graphs" on page 80, has a similar concept called repositories. If I sent SPARQL HTTP requests to the Sesame triplestore, the URL might look more like *http://localhost:8080/openrdf-sesame/repositories/myRepository/rdf-graphs/service*, where *myRepository* was the name of a repository I had created within Sesame. Like Fuseki, Sesame offers other options in addition to what I describe here for accessing data with the SPARQL Graph Store HTTP Protocol; see their documentation for more information.

Although we'll be sending requests for triples to a server, we're not sending SPARQL queries to a SPARQL endpoint, but to a different service offered by the same server at a different URL—in other words, don't confuse support for the SPARQL Graph Store HTTP Protocol with a SPARQL endpoint. For example, I would send a SPARQL query of the myDataset dataset on Fuseki to its endpoint at *http://localhost:3030/myDataset/query*, but I would send a SPARQL HTTP update request for the same dataset to *http://localhost:3030/myDataset/data*.

If you have a dataset called myDataset in your Fuseki installation and it has data that you want to keep, you'll want to use a different one for these experiments—especially when we try the HTTP DELETE command.

To set up some data to play with in Fuseki, repeat these steps from Chapter 6: create and select a myDataset dataset, clear its existing data with the DROP ALL update request (ex337.ru), and then run the ex338.ru update request to add a pair of triples to the dataset's default graph and a few more to each of 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" .
  }
}
```

Run the ex332.rq List All Triples query from that chapter to see what you've got, which should be the six triples inserted by ex338.ru:

```
# filename: ex332.rq

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

The same chapter suggested that you bookmark the results of the ex332.rq List All Triples query so that you can run it by simply going to that bookmark; that will be handy on the upcoming pages as well. To make it even simpler, because the query is embedded in the URL that retrieves the results, you can just reload the "page" of your browser that displays the search results after each HTTP PUT, POST, and DELETE request below so that you can see the effect of the request.

GET a Graph of Triples

The simplest HTTP operation is GET. It's what a browser does when you send it to a typical web page, and it's the default action of curl. The following asks for the triples in the default graph of the myDataset dataset:

```
curl http://localhost:3030/myDataset/data?graph=default
```

Fuseki responds to this request with an RDF/XML version of the two triples that ex338.ru had put into the default graph:

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:j.0="http://learningsparql.com/ns/demo#" >
  <rdf:Description rdf:about="http://learningsparql.com/ns/data#x">
```

```
    <j.0:tag>one</j.0:tag>
    <j.0:tag>two</j.0:tag>
  </rdf:Description>
</rdf:RDF>
```

When you specify an HTTP header indicating that you want the returned data in a different format (in this case, using the media type designator for Turtle),

```
curl -H "Accept: text/turtle" http://localhost:3030/myDataset/data?graph=default
```

a server that supports the SPARQL Graph Store HTTP Protocol returns the requested data in that format:

```
<http://learningsparql.com/ns/data#x>
      <http://learningsparql.com/ns/demo#tag>
                  "two" ;
      <http://learningsparql.com/ns/demo#tag>
                  "one" .
```

Check your server's documentation to see which media types it supports, and in particular, whether it supports the standard designator for the Turtle format used here or one of the older designators used before Turtle became standardized, such as "application/x-turtle".

We saw above that when we add "?graph=default" to the URL supporting this request, we're asking for the default graph's triples. To ask Fuseki for the triples from a specific named graph, include the URI of that graph's name as a `graph` parameter in the request. Because the graph name is a parameter being passed as part of the request and not part of the address where the request is being sent, the graph name should be escaped, so you'll see some of the trickier URL characters in the graph name *http://learningsparql.com/ns/data#g1* escaped in this request:

```
curl http://localhost:3030/myDataset/data\
?graph=http%3A%2F%2Flearningsparql.com%2Fns%2Fdata%23g1
```

When you pack this much information into a URL, it's too long to fit the width of one of these pages, so I've split them here. Consider each curl command in this chapter to be one line. Remove the backslash that you see at the end of any lines and join up the command there; be careful not to add or remove any spaces shown just before the backslash. For example, the middle of the one-line version of the command above would say `myDataset/data?graph=http%3A%2F` with no space where the two pieces were rejoined.

This request retrieves the "three" and "four" triples that ex338.ru added to the g1 graph earlier.

PUT a Graph of Triples

To demonstrate the use of the SPARQL Graph Store HTTP Protocol to send triples to a triplestore, we'll have curl read the following file from disk and PUT it to Fuseki (remember that ex338.ru added triples with objects of "one" through "six" to the triplestore):

```
# filename: ex530.ttl

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

d:x dm:tag "seven" .
d:x dm:tag "eight" .
```

The HTTP PUT command sends a resource to a server to be stored using a particular URI. In SPARQL Graph Store HTTP terms, a PUT command sends a set of triples to be stored in the graph named by that URI. The following command sends the contents of ex530.ttl (the two triples with "seven" and "eight" in them) for Fuseki to store with the name *http://learningsparql.com/ns/data#g3*. After running it, try the ex332.rq List All Triples query to see where the triples ended up:

```
curl -X PUT --data-binary @ex530.ttl -H "Content-Type: text/turtle" \
http://localhost:3030/myDataset/data\
?graph=http%3A%2F%2Flearningsparql.com%2Fns%2Fdata%23g3
```

Although ex530.ttl is a text file, my curl command here uses the `--data-binary` parameter because without it, curl may not respect the file's line-ending characters. These characters don't affect the validity of the Turtle syntax, but if Fuseki's parser doesn't see those line endings and treats the file as one long line, the # character that starts this long line will make it look like one long line that's been commented out.

Because PUT essentially says "here is the resource to store at this URI," if there's something already at that URI, it gets replaced. The following command sends the "seven" and "eight" triples from ex530.ttl to be stored in the default graph. After you execute this command and list all the stored triples, you'll see that the "seven" and "eight" triples replaced the "one" and "two" triples that ex338.ru originally put there:

```
curl -X PUT --data-binary @ex530.ttl -H "Content-Type: text/turtle" \
http://localhost:3030/myDataset/data?graph=default
```

POST a Graph of Triples

An HTTP POST command requests that the server accept the entity being sent as a subordinate resource of the existing resource named by the URI in the request. In SPARQL Graph Store HTTP terms, a POST commands sends a set of triples to be added to the named graph. The following command sends the “seven” and “eight” triples from ex530.ttl to be added to the *http://learningsparql.com/ns/data#g2* named graph:

```
curl -X POST --data-binary @ex530.ttl -H "Content-Type: text/turtle" \
http://localhost:3030/myDataset/data?graph=http%3A%2F%2F\
learningsparql.com%2Fns%2Fdata%23g2
```

After executing this command and running the ex332.rq List All Triples query, you’ll see that the command added the triples to named graph g2 and that g2 still has its original “five” and “six” triples.

DELETE a Graph of Triples

This one is pretty self-explanatory. You give the server the URI of a named graph to delete, and the server deletes that graph’s triples. Try the following, which deletes the triples in the *http://learningsparql.com/ns/data#g3* graph that was created in “PUT a Graph of Triples” on page 300:

```
curl -X DELETE -H "Content-Type: text/turtle" \
http://localhost:3030/myDataset/data?\
graph=http%3A%2F%2Flearningsparql.com%2Fns%2Fdata%23g3
```

Using this command to delete the default graph deletes only the triples in that graph. After running the command, running this next one deletes the “seven” and “eight” triples from the default graph, leaving the remaining triples in the g1 and g2 graphs alone:

```
curl -X DELETE -H "Content-Type: text/turtle" \
http://localhost:3030/myDataset/data?graph=default
```

Summary

In this chapter, we learned:

- How some applications (or application components) can execute business logic based on the presence of specific values—especially predicates—in the triples that they see
- How to send a SPARQL query to an endpoint from popular scripting languages and from a web form, and how applications that use these can make use of the returned data
- The role that standalone, triplestore, and middleware SPARQL processors can play to make RDF and non-RDF data available to applications that use SPARQL queries
- How RESTful (or, perhaps, “REST-like”) applications can use the SPARQL Graph Store HTTP Protocol to manage named graphs of triples on a remote server

CHAPTER 11

A SPARQL Cookbook

This chapter shows some SPARQL queries that can be handy with a wide variety of datasets. Many of these queries will work with just about any set of triples out there, including those which you know nothing about. These queries are especially useful in those situations, helping you to learn more about exactly what you have to work with when exploring a new set of data.

In this chapter, we'll learn about:

"Themes and Variations"
: SPARQL offers a few simple features, covered earlier in this book, that you can use to enhance most of this chapter's queries. This section reviews them.

"Exploring the Data" on page 306
: This section has queries to see what you've got in a particular dataset—for example, how much structure has been defined there, how much of that structure is used, and, if there isn't much structure, how you can still get a feel for what kind of data is there.

"Creating and Updating Data" on page 341
: Some changes to a dataset would be tedious to make by hand and very simple with the right UPDATE query, especially when you need to perform various kinds of global replacements.

Themes and Variations

The queries shown in this chapter fall into several themes, and there are several variations than can apply to many of them. For example, sorting your output by the values in one or more columns of query output can make it easier to see patterns, especially if you get a lot of results; see "Sorting Data" on page 96 for details of how to use the ORDER BY phrase. You'll see it used here and there in this chapter when the sample input provides enough query results that sorting makes the output easier to read.

For some datasets out there, some of these queries are asking for a lot of information —maybe too much. For example, "What Properties Are Used?" on page 314 asks the SPARQL engine to list every predicate used in a dataset, which is fine for a dataset of a couple of megabytes that someone emailed to you but not reasonable to send to DBpedia. Queries in this chapter that could potentially be asking too much have the line `LIMIT 50` commented out at the end. If you see it and will be sending the query to a large dataset, uncomment that line and see how it goes. Maybe you can increase the number above 50 a few times before you decide that it's worth commenting out the line and really asking for all the triples that meet the specified patterns.

The possibility that some of these queries are more reasonable to use on some datasets than on others brings up another point: several of these queries are not very efficient, which is more of an issue as you use them on larger and larger datasets. For example, if a query has lots of OPTIONAL graph patterns or it has triple patterns with more than one variable, these can slow it down, especially when they appear early in the query. See Chapter 7 for further background on which SPARQL features can slow down queries against large datasets.

When you add data to a dataset, it's best to try an INSERT update request as a CONSTRUCT query first, just to check on which triples would be added if you really did INSERT them. Although you can combine INSERT and DELETE instructions together, you can't combine CONSTRUCT and DELETE, so when testing INSERT requests by temporarily making them CONSTRUCT queries, remember to comment out any DELETE instructions. For example, before running the query in "How Do I Globally Replace a Property Value?" on page 342 the way it's shown in this chapter, try running it like this first to see what it would create:

```
# filename: ex479.rq

PREFIX dcterms: <http://purl.org/dc/terms/>

#DELETE {?document dcterms:dateCopyrighted "2013" . }
#INSERT {?document dcterms:dateCopyrighted "2014" . }
CONSTRUCT{?document dcterms:dateCopyrighted "2014" . }
WHERE  {?document dcterms:dateCopyrighted "2013" . }
```

As another test, you can use comments to temporarily substitute a CONSTRUCT statement for DELETE to find out exactly what would be deleted. The following will list triples with a `dcterms:dateCopyrighted` value of "2013", which is what the DELETE clause would be deleting:

```
# filename: ex480.rq

PREFIX dcterms: <http://purl.org/dc/terms/>

#DELETE {?document dcterms:dateCopyrighted "2013" . }
#INSERT {?document dcterms:dateCopyrighted "2014" . }
CONSTRUCT{?document dcterms:dateCopyrighted "2013" . }
WHERE  {?document dcterms:dateCopyrighted "2013" . }
```

The queries shown in this chapter are only starting points that you can refine as you learn more about the dataset that you're working with.

Three datasets are used as samples for most of this chapter: ex012.ttl, the VoID RDF Schema for describing datasets, and the Good Relations ecommerce vocabulary.

The ex012.ttl dataset will look familiar from other examples in this book:

```
# 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 Vocabulary of Interlinked Datasets (VoID) RDF Schema is specified in a W3C Interest Group Note and lets you describe metadata about a dataset such as the vocabularies that the dataset uses and any associated SPARQL endpoints. The sample files that accompany this book include VoID's definition file in a file called void.ttl.

The Good Relations ecommerce vocabulary is popular for describing products and services for sale. More and more large and small ecommerce sites are using RDFa and schema.org markup to embed metadata that uses this vocabulary into their web pages, making it easier for search engines and other applications to use data about these sites' products.

As the comment at the beginning of the goodrelations-v1.owl file included with this book's sample data tells you, the file was originally called v1.owl and downloaded from the URL *http://www.heppnetz.de/ontologies/goodrelations/v1.owl*. As you read this, a more recent version of the Good Relations ontology may be available on that site.

If you've never dug into the VoID or Good Relations vocabularies, so much the better. One great advantage of RDF technology is that schemas and ontologies are stored using the same data model as the data that they describe, and many of this chapter's queries are aimed specifically at helping you to explore this metadata so that you can learn more about a given dataset. You'll see that these queries can help you explore and learn about schemas like VoID and Good Relations as easily as they can help you to navigate inventory or publishing metadata.

When this chapter's queries ask about specific classes, properties, or values (for example, the query about the `gr:hasNext` property in "What Values Does a Given Property Have?" on page 326), it uses examples from these three datasets, but you can plug in other classes, properties or values from datasets that you're working with. If these include any namespace prefixes, don't forget to include the appropriate declarations for these prefixes in your query.

It's always best for variable names to give a clue about the data they store, but because many of this chapter's queries are generalized so that you can explore data that you may not yet know much about, these queries often use the generic variable names `?s`, `?p`, and `?o` to represent subjects, predicates, and objects.

A SPARQL query that plays a specific role in an application will probably SELECT specific variables from the query's graph pattern. Because many of this chapter's queries are quick and dirty short ones that don't use many variables, the queries often use the asterisk wildcard to SELECT all the variables for output (`SELECT *`). As described in Chapter 7, naming specific variables can make the query more efficient.

Exploring the Data

This section has queries that, as shown or with slight modifications, can help you explore a wide range of datasets.

How Do I Look at All the Data at Once?

Problem

When you have a new dataset, the first question on your mind is usually, "What do we have here?" A simple way to find out is to just ask for all the data.

This warning applies to all the queries in this chapter that include the line `# LIMIT 50` at the end, but it goes double for the two queries here: asking for all the data can be asking for too much.

Solution

The first query is a very common one, and it appears to be asking for all the triples in a dataset:

```
# filename: ex436.rq

SELECT *
WHERE
{
  ?s ?p ?o .
}
#LIMIT 50
```

This query actually just asks for all the triples in the default graph, although some triplestores may return triples in named graphs as well. If they do, they won't return the names of the named graphs, because the query doesn't ask for it. This next one does:

```
# filename: ex437.rq

SELECT ?g ?s ?p ?o
WHERE
{
  { ?s ?p ?o }
  UNION
  { GRAPH ?g { ?s ?p ?o } }
}
# LIMIT 50
```

This query really does ask for all the triples in a dataset, whether they're in the default graph or in any named graphs, and the `?g` in the WHERE clause and SELECT statement shows that the query is asking for the graph names as well.

Discussion

If you've loaded a file from disk and you know how big it is, you'll have some sense of whether it would make sense to try this query. If you're querying a remote SPARQL endpoint, other queries in this chapter give you more sensible ways to start investigating that data than to just ask for all the triples outright.

As you can see in Chapter 6, I use both of these queries often when I know that I'm working with a small dataset. Sometimes this is because I'm trying out new SPARQL tools or techniques, so I'm just playing. For more serious projects, if someone has emailed me some data, then I know that I don't have to plan around scale issues when querying a dataset that is small enough to attach to an email, and I have complete flexibility in what I ask for. These two queries are very handy in these situations.

See Also

- "What Classes Are Declared?" on page 308
- "What Properties Are Declared?" on page 310

- "Which Classes Have Instances?" on page 313
- "What Properties Are Used?" on page 314
- "What Values Does a Given Property Have?" on page 326
- "How Do I Delete All the Data?" on page 341

What Classes Are Declared?

Problem

An RDF dataset may or may not have some structure defined as an ontology or RDF Schema. If so, these definitions can make it easier for you to work with the data, because someone explicitly modeled its structures and recorded them. A quick way to check on this is to list any classes that were declared.

Solution

The two ways to say "there is a class called Employee" in RDF are both similar to the RDF way to say "emp91234 is a member of class Employee": you could say "Employee is a member of the class owl:Class" or you could say "Employee is a member of the class rdfs:Class." The former is more common—if people only use a tiny bit of OWL in their data, it's common for that tiny bit to be class declarations. The following Turtle file makes all three of these statements:

```
# filename: ex438.ttl

@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix d:    <http://learningsparql.com/ns/data#> .

d:emp91234 a d:Employee .      # "a" a shortcut for "rdf:type"
d:Employee a rdfs:Class .
d:Employee a owl:Class .
```

The OWL specification defines `owl:Class` as a subclass of `rdfs:Class`, so anything that is an `owl:Class` is also an `rdfs:Class`.

Now, to list any declared classes, we know what to query for: triples saying that something has an `rdf:type` of `owl:Class` or an `rdf:type` of `rdfs:Class`.

The following query asks for all of these triples. The last triple pattern binds the class's type (either `owl:Class` or `rdfs:Class`) to the `?classType` variable so that we can see which one was used in the declaration:

```
# filename: ex439.rq

PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT *
WHERE
{
  { ?class a owl:Class }
  UNION
  { ?class a rdfs:Class }

  ?class a ?classType
}
```

Running this query on the void.ttl data gives us an interesting result:

```
-------------------------------------------------------------
| class                                      | classType  |
=============================================================
| <http://rdfs.org/ns/void#DatasetDescription> | owl:Class  |
| <http://rdfs.org/ns/void#DatasetDescription> | rdfs:Class |
| <http://rdfs.org/ns/void#TechnicalFeature>   | owl:Class  |
| <http://rdfs.org/ns/void#TechnicalFeature>   | rdfs:Class |
| <http://rdfs.org/ns/void#Linkset>            | owl:Class  |
| <http://rdfs.org/ns/void#Linkset>            | rdfs:Class |
| <http://rdfs.org/ns/void#Dataset>            | owl:Class  |
| <http://rdfs.org/ns/void#Dataset>            | rdfs:Class |
| <http://rdfs.org/ns/void#DatasetDescription> | owl:Class  |
| <http://rdfs.org/ns/void#DatasetDescription> | rdfs:Class |
| <http://rdfs.org/ns/void#TechnicalFeature>   | owl:Class  |
| <http://rdfs.org/ns/void#TechnicalFeature>   | rdfs:Class |
| <http://rdfs.org/ns/void#Linkset>            | owl:Class  |
| <http://rdfs.org/ns/void#Linkset>            | rdfs:Class |
| <http://rdfs.org/ns/void#Dataset>            | owl:Class  |
| <http://rdfs.org/ns/void#Dataset>            | rdfs:Class |
-------------------------------------------------------------
```

This says that the void.ttl schema declared each class as both an `rdfs:Class` and as an `owl:Class`. This excerpt from void.ttl shows that this is indeed the case:

```
# filename: ex442.ttl (excerpt from void.ttl)

void:Linkset a rdfs:Class, owl:Class;
    rdfs:label "linkset";
    rdfs:comment "A collection of RDF links between two void:Datasets.";
    rdfs:subClassOf void:Dataset .
```

Discussion

To be honest, instead of ex439.rq, I'd be more likely to type in the following much simpler query, then run it, then change `owl:Class` to `rdfs:Class` and run it again:

```
# filename: ex440.rq

PREFIX owl:  <http://www.w3.org/2002/07/owl#>
```

```
SELECT *
WHERE
{ ?class a owl:Class }
```

(Then I'd see that the query didn't work because I had used the `rdfs:` prefix without declaring it, so I'd add that prefix declaration at the top and run the query again. I told you I was being honest.)

The greater simplicity of this query means that I'd be more likely to remember all the relevant syntax off the top of my head, and by running the two slightly different versions I'd still get all the same information.

It's not unusual to declare a class by simply saying that it's a subclass of another one, so querying for triples that have a predicate of `rdfs:subClassOf` may turn up more class names.

See Also

- "What Properties Are Used?" on page 314
- "Which Classes Have Instances?" on page 313
- "How Much Was a Given Class Used?" on page 320

What Properties Are Declared?

Problem

As we saw in "What Classes Are Declared?" on page 308, an RDF dataset may have some structure defined as OWL ontology metadata or RDF Schema metadata. If so, this structural metadata makes it easier for you to work with the data because it gives you a better idea of what's there. In traditional object-oriented modeling, you wouldn't have properties (or, as they might be called in that case, attributes) declared without also having classes declared because the attributes would be details of class structures. With RDF, on the other hand, you may find that some classes have been declared, or that some properties have been declared, or maybe both, or maybe neither. So, checking for property declarations is worth it even if there are no class declarations.

A set of property declarations with no class declarations is actually quite common because, as a way to say "here are the properties that may show up in the predicates of my triples," it lets people share their vocabularies for reuse without getting involved in more structural declarations of classes and their relationships. The Dublin Core Metadata Element Set is one example.

Solution

We also saw in that same section that RDF data declares something to be a class by saying that it has an `rdf:type` of `owl:Class` or `rdfs:Class`, or perhaps by saying that it's an `rdfs:subClassOf` another class. Property declarations are similar: as the following shows, we can declare properties by saying that they have an `rdf:type` of `rdf:Property` or of one of its subclasses. The latter is especially common with OWL modeling; in the following, `owl:DatatypeProperty` is a subclass of `rdf:Property`:

```
# filename: ex443.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:playsInstrument rdf:type rdf:Property .
ab:quantity rdf:type owl:DatatypeProperty .
```

Now, to find property declarations, we know what to query for. We can ask for any instances of the `rdf:Property` class with a query like this:

```
# filename: ex444.rq

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

SELECT  ?property
WHERE {
  ?property a rdf:Property .   # could have said rdf:type instead of a
}
```

To account for the possibility that a property may be declared as a subproperty of a subclass of `rdf:Property`, the following makes a more thorough request for declared properties because the plus sign (described further in "Searching Further in the Data" on page 61) tells the query engine to look for indirect subclass relationships as well as direct ones:

```
# filename: ex445.rq

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#>

SELECT *
WHERE
{
  ?propClass rdfs:subClassOf+ rdf:Property .
  ?property a ?propClass .
}
```

If we tell ARQ to apply this query to the Good Relations ontology, though, we get nothing. Good Relations declares many properties as instances of `rdf:Property` subclasses (for example, it declares `gr:height` to be an `owl:ObjectProperty` and gr:priceType to be an owl:DatatypeProperty), but ARQ doesn't know that owl:ObjectProperty and owl:DatatypeProperty are subclasses of rdf:Property—unless we provide the appropriate metadata as part of the data to query. To do this, I retrieved the OWL file that declares OWL itself from http://www.w3.org/2002/07/owl and then, because I saw that it was a Turtle file, I renamed it as owl.ttl. I could then provide this file as an additional data file for ARQ to query with the Good Relations ontology:

```
arq -query ex445.rq --data goodrelations-v1.owl -data owl.ttl
```

This time, the query gives us over 330 results, beginning with the following lines:

```
-------------------------------------------------------------------------------
| propClass              | property                                           |
===============================================================================
| owl:OntologyProperty   | owl:versionIRI                                     |
| owl:OntologyProperty   | owl:priorVersion                                   |
| owl:OntologyProperty   | owl:incompatibleWith                               |
| owl:OntologyProperty   | owl:imports                                        |
| owl:OntologyProperty   | owl:backwardCompatibleWith                         |
| owl:AnnotationProperty | <http://purl.org/goodrelations/v1#displayPosition> |
| owl:AnnotationProperty | owl:incompatibleWith                               |
```

The URL where I retrieved the OWL file should be familiar: it's the OWL namespace URI.

Discussion

The ex445.rq query gets results when we add owl.ttl to the goodrelations-v1.owl dataset that we originally queried because owl.ttl includes triples that say things like {owl:ObjectProperty rdfs:subClassOf rdf:Property} and {owl:AsymmetricProperty rdfs:subClassOf owl:ObjectProperty}. This helps ARQ to understand more about the Good Relations ontology.

OWL and RDFS are more than just specifications: they themselves are RDF-based vocabularies, providing more metadata (more semantics!) about the terms they define, and you can use these vocabularies to do more with your own data and that of others.

See Also

- "What Properties Are Used?" on page 314
- "How Much Was a Given Property Used?" on page 317

Which Classes Have Instances?

Problem

Lots of class declarations may mean that lots of structure has been defined, but how much data actually uses that structure? In other words, of all the data instances in the data, what are they instances of?

Solution

This query lists any classes used in the dataset, whether those classes have been declared or not:

```
# filename: ex447.rq

SELECT DISTINCT ?class
WHERE {
  ?instance a ?class .
}
ORDER by ?class
```

When we take the ex439.rq query in "What Classes Are Declared?" on page 308 and run it with on goodrelations-v1.owl, it finds 98 classes declared. However, when we run the query above on the same dataset, we find that less than a sixth of those are actually used in the ontology:

```
-----------------------------------------------------------------
| class                                                         |
=================================================================
| <http://purl.org/goodrelations/v1#BusinessEntityType>        |
| <http://purl.org/goodrelations/v1#BusinessFunction>          |
| <http://purl.org/goodrelations/v1#DayOfWeek>                 |
| <http://purl.org/goodrelations/v1#DeliveryMethod>            |
| <http://purl.org/goodrelations/v1#DeliveryModeParcelService> |
| <http://purl.org/goodrelations/v1#Offering>                  |
| <http://purl.org/goodrelations/v1#PaymentMethod>             |
| <http://purl.org/goodrelations/v1#PaymentMethodCreditCard>   |
| <http://purl.org/goodrelations/v1#WarrantyScope>             |
| <http://www.w3.org/2002/07/owl#AnnotationProperty>           |
| <http://www.w3.org/2002/07/owl#Class>                        |
| <http://www.w3.org/2002/07/owl#DatatypeProperty>             |
| <http://www.w3.org/2002/07/owl#ObjectProperty>               |
| <http://www.w3.org/2002/07/owl#Ontology>                     |
| <http://www.w3.org/2002/07/owl#SymmetricProperty>            |
| <http://www.w3.org/2002/07/owl#TransitiveProperty>           |
-----------------------------------------------------------------
```

Discussion

Good Relations has a perfectly good reason to declare so many more classes than it uses: the idea of an ontology is usually to define structure for other data to use, and there are tons of datasets around the world using a broader selection of Good Relations classes than the ontology itself uses. It's probably a different story for different datasets, and that story is often an important part of what purpose that dataset serves.

Remember, `rdf:Property` and its OWL-related subclasses are themselves classes, so if properties were declared, you will see these classes show up in the list of classes that have instances. The output above includes a few of these.

See Also

- "What Properties Are Declared?" on page 310
- "What Classes Are Declared?" on page 308
- "How Much Was a Given Class Used?" on page 320
- "A Given Class Has Lots of Instances. What Are These Things?" on page 321
- "What Data Is Stored About a Class's Instances?" on page 324
- "How Do I Turn Resources into Instances of Declared Classes?" on page 347

What Properties Are Used?

Problem

This is probably the most basic query you can make of a new dataset, because when you want to know what a dataset has to tell you, this very simple query will always get a useful answer. You want to list all the properties that the dataset uses—whether they're declared or not—but if there are 10,000 triples, you don't want a query that will give you 10,000 results.

Solution

You want a query that asks for a list of all the predicates, but with no duplicates:

```
# filename: ex449.rq

SELECT DISTINCT ?property
WHERE
{ ?s ?property ?o . }
# LIMIT 50
```

Here is the result set that this query retrieves when run with the ex012.ttl dataset:

```
---------------------------------------------------------
| property                                              |
=========================================================
| <http://learningsparql.com/ns/addressbook#email>      |
| <http://learningsparql.com/ns/addressbook#homeTel>    |
| <http://learningsparql.com/ns/addressbook#lastName>   |
| <http://learningsparql.com/ns/addressbook#firstName>  |
---------------------------------------------------------
```

Discussion

This is probably my favorite SPARQL query, and it's often the first one I execute when exploring a new dataset. Class and property declarations and metadata are handy, but completely optional in RDF. Predicates are not optional, and while a list of subjects will tell you what is being described (and it won't tell you much about them—just their identifiers) and a list of objects will tell you what the values are, a list of predicates tells you what kind of information is being provided about the subjects, which is usually the first thing I want know.

The output shown from running this query with the ex012.ttl dataset is simple enough, but for a larger, more complex dataset—even the Good Relations or VoID ontologies —adding an `ORDER BY ?property` line at the end can make it easier to quickly see what you have.

If you query a dataset in a triplestore that supports named graphs, the query above will only retrieve the properties used in the default graph. The following variation on this query asks for all the predicates used in all the graphs as well as the names of the graphs containing these predicates:

```
# filename: ex538.rq

SELECT DISTINCT ?graph ?property
WHERE
{
  { ?s ?property ?o . }
  UNION
  { GRAPH ?graph { ?s ?property ?o } }
}
# LIMIT 50
```

If different properties are used in different named graphs, that may tell you something interesting about the role of those named graphs.

See Also

- "What Properties Are Declared?" on page 310
- "How Much Was a Given Property Used?" on page 317
- "What Values Does a Given Property Have?" on page 326
- "Which Classes Use a Particular Property?" on page 316

Which Classes Use a Particular Property?

Problem

You see that a dataset uses a certain property, and you'd like to know more about how or why that property is being used.

Solution

Let's say that you took the query in "What Properties Are Used?" on page 314, ran it on the Good Relations ontology, and saw a property called `hasNext`. Has next what? The following query asks what resources have a value for this property, and what class each of those resources belong to:

```
# filename: ex451.rq

PREFIX gr: <http://purl.org/goodrelations/v1#>

SELECT DISTINCT ?class
WHERE {
?s gr:hasNext ?o ;
   a ?class .
}
```

Running this query with goodrelations-v1.owl gives us a simple answer:

```
----------------
| class        |
================
| gr:DayOfWeek |
----------------
```

Discussion

When you wonder what a particular property is used for, you can learn a lot from a list of which classes have instances that use that property.

This query is also handy with broader, more generalized properties. For example, if you use this query to see what kinds of resources include `rdfs:label` or `rdfs:comment` values, you can more quickly find descriptive parts of the data that provide clues about how the data is intended to work.

Someone creating an ontology or RDF schema may also use the `rdfs:domain` property to associate a property with a class. As we saw in "Reusing and Creating Vocabularies: RDF Schema and OWL" on page 36, if one triple says that `{ab:playsInstrument rdfs:domain ab:Musician}` and another triple says that `{ab:i0432 ab:playsInstrument ab:vacuumCleaner}`, that means that resource `ab:i0432` is a member of the class `ab:Musician`. In addition to letting applications infer a resource's class membership like this, `rdfs:domain` gives an application some excellent clues about what properties to use if the application must display a form that lets users edit data for an instance of a particular class.

The following query asks what class the `gr:hasCurrency` property is associated with in the data model. The query doesn't care whether any instances of that class actually have a value for this property, but only whether the data model associates that class with that property:

```
# filename: ex533.rq

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX gr:   <http://purl.org/goodrelations/v1#>

SELECT ?class
WHERE
{ gr:hasCurrency rdfs:domain ?class }
```

When run with the Good Relations ontology, we get this result:

```
-------------------------
| class                 |
=========================
| gr:PriceSpecification |
-------------------------
```

What other properties are associated with that class? (Or, to use a realistic use case, if an application was going to display a form letting the user edit values of a `gr:PriceSpecification` instance, what fields should be on that form?) See query ex434.rq in "Querying Schemas" on page 271 for a query that you can use as a model to find out.

See Also

- "What Properties Are Used?" on page 314
- "How Much Was a Given Class Used?" on page 320
- "How Much Was a Given Property Used?" on page 317
- "What Values Does a Given Property Have?" on page 326
- "What Data Is Stored About a Class's Instances?" on page 324

How Much Was a Given Property Used?

Problem

Perhaps you're writing an application to process a dataset, and you need to write code to process its various properties. Some properties are probably used more than others; knowing which come up the most can help you prioritize your development efforts. After all, a property used in 3,000 triples is probably more important than a property used in 3 triples.

Solution

How much does the `hasNext` property come up in the Good Relations ontology? The following query tells us:

```
# filename: ex453.rq

PREFIX gr: <http://purl.org/goodrelations/v1#>

SELECT (COUNT(?o) AS ?hasNextTotal)
WHERE
{ ?s gr:hasNext ?o }
```

The answer turns out to be 7.

1.1 Alert

The `COUNT()` function was new with SPARQL 1.1.

For a large dataset, this query can be asking a lot of the query engine, and the following query asks much more—for a sorted list of how much every property was used:

```
# filename: ex454.rq

SELECT ?p (COUNT(?p) AS ?pTotal)
WHERE
{ ?s ?p ?o . }
GROUP BY ?p
ORDER BY DESC(?pTotal)
```

Running this on the Good Relations ontology gives an interesting report:

```
-----------------------------------------------------------------
| p                                                     | pTotal |
=================================================================
| <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>    | 267    |
| <http://www.w3.org/2002/07/owl#disjointWith>         | 256    |
| <http://www.w3.org/1999/02/22-rdf-syntax-ns#first>   | 213    |
| <http://www.w3.org/1999/02/22-rdf-syntax-ns#rest>    | 213    |
| <http://www.w3.org/2000/01/rdf-schema#comment>       | 177    |
| <http://www.w3.org/2000/01/rdf-schema#label>         | 177    |
| <http://www.w3.org/2000/01/rdf-schema#isDefinedBy>   | 176    |
| <http://www.w3.org/2000/01/rdf-schema#domain>        | 96     |
| <http://www.w3.org/2000/01/rdf-schema#range>         | 96     |
| <http://www.w3.org/2002/07/owl#unionOf>              | 61     |
| <http://www.w3.org/2000/01/rdf-schema#subPropertyOf> | 22     |
| <http://www.w3.org/2000/01/rdf-schema#subClassOf>    | 19     |
| <http://www.w3.org/2002/07/owl#inverseOf>            | 10     |
| <http://www.w3.org/2002/07/owl#equivalentClass>      | 9      |
| <http://purl.org/goodrelations/v1#displayPosition>   | 8      |
| <http://purl.org/goodrelations/v1#hasNext>           | 7      |
| <http://purl.org/goodrelations/v1#hasPrevious>       | 7      |
| <http://www.w3.org/2002/07/owl#deprecated>           | 6      |
| <http://www.w3.org/2002/07/owl#equivalentProperty>   | 4      |
| <http://purl.org/dc/elements/1.1/contributor>        | 1      |
| <http://purl.org/dc/elements/1.1/creator>            | 1      |
| <http://purl.org/dc/elements/1.1/rights>             | 1      |
| <http://purl.org/dc/elements/1.1/subject>            | 1      |
| <http://purl.org/dc/elements/1.1/title>              | 1      |
| <http://purl.org/dc/terms/license>                   | 1      |
| <http://purl.org/goodrelations/v1#category>          | 1      |
| <http://purl.org/goodrelations/v1#description>       | 1      |
| <http://www.w3.org/2002/07/owl#versionInfo>          | 1      |
| <http://xmlns.com/foaf/0.1/homepage>                 | 1      |
-----------------------------------------------------------------
```

It tells us, for example, that `rdf:type` is the most commonly used property. This is no surprise, because Good Relations is an ontology, declaring classes and properties for ecommerce applications to use. The output also shows that the Dublin Core properties `dc:contributor`, `dc:creator`, `dc:rights`, `dc:subject`, and `dc:title` are each only used once, so if I were writing an application to process this data, I might not put a huge amount of effort into the code that processes these particular properties.

Discussion

As the previous section noted, a request to count all the triples in a dataset that meet a certain condition can be asking a lot of a query engine if the dataset is large enough, so be careful where you run ex453.rq. Be even more careful where you run ex454.rq, which asks the SPARQL engine to do far more counting. It's probably best not to run either of these on a remote dataset—for example, a dataset exposed using a SPARQL endpoint—unless you're absolutely sure that the dataset has no more than a few thousand triples. (As I write this, DBpedia has 1.89 billion triples.)

The LIMIT keyword, which tells the SPARQL engine to return no more than a certain number of result rows, will be no help here because you'd still be asking for the SPARQL engine to do the same amount of counting work.

On the other hand, if you have a dataset on your hard disk that you're trying to learn more about, go for it. Even if it takes a few minutes—or a lot of minutes—for one of these queries to run, its results can provide valuable information about the dataset.

See Also

- "What Properties Are Declared?" on page 310
- "What Properties Are Used?" on page 314
- "What Values Does a Given Property Have?" on page 326
- "How Much Was a Given Class Used?" on page 320
- "Which Classes Use a Particular Property?" on page 316

How Much Was a Given Class Used?

Problem

The reasons for counting the instances of a particular class are similar to the reasons for asking "How Much Was a Given Property Used?"on page 317: it helps to prioritize your development time. The queries that you can use to do the counting are also similar.

Solution

Perhaps after asking "What Classes Are Declared?" on page 308 in the VoID ontology, you see that the Dublin Core Metadata Terms vocabulary's `dct:Agent` class is one of them. How many instances of this class does the VoID ontology have? The following query will tell you:

```
# filename: ex456.rq

PREFIX dct: <http://purl.org/dc/terms/>

SELECT (COUNT(?s) AS ?agentTotal)
WHERE
{ ?s a dct:Agent }
```

The VoID ontology has four instances of this class.

1.1 Alert

The `COUNT()` function was new with SPARQL 1.1.

As with its equivalent in "How Much Was a Given Property Used?" on page 317, with a large dataset this query may ask a lot of the query engine. The following query asks for much more—a sorted list of how many instances every class has:

```
# filename: ex457.rq

SELECT ?class (COUNT(?s) AS ?instanceTotal)
WHERE
{ ?s a ?class. }
GROUP BY ?class
ORDER BY DESC(?instanceTotal)
```

Running this on the VoID ontology gives these results:

```
-------------------------------------------------------------------------
| class                                                | instanceTotal |
=========================================================================
| <http://www.w3.org/1999/02/22-rdf-syntax-ns#Property> | 27            |
| <http://www.w3.org/2002/07/owl#DatatypeProperty>     | 8             |
| <http://purl.org/dc/terms/Agent>                     | 4             |
| <http://www.w3.org/2000/01/rdf-schema#Class>         | 4             |
| <http://www.w3.org/2002/07/owl#Class>                | 4             |
| <http://www.w3.org/2002/07/owl#FunctionalProperty>   | 4             |
| <http://xmlns.com/foaf/0.1/Person>                        | 4               |
| <http://www.w3.org/ns/adms#SemanticDistribution>          | 2               |
| <http://www.w3.org/2002/07/owl#Ontology>                  | 1               |
| <http://www.w3.org/ns/adms#SemanticAsset>                 | 1               |
-------------------------------------------------------------------------
```

Interestingly, this ontology has no instances of the classes that it itself declares, although this is not unusual; the purpose of this ontology is to define a model for others to use in their applications.

Discussion

All the same issues that that apply to the corresponding queries in "How Much Was a Given Property Used?" on page 317 apply to the two queries in this section.

See Also

- "What Classes Are Declared?" on page 308
- "Which Classes Have Instances?" on page 313
- "Which Classes Use a Particular Property?" on page 316
- "How Much Was a Given Property Used?" on page 317

A Given Class Has Lots of Instances. What Are These Things?

Problem

A class's instances may have a lot of data associated with them, with some values more cryptic than others. Is there a simple way to find out exactly what these instances represent?

Solution

A resource's URI isn't supposed to tell you anything about it; the main point of the `rdfs:label` property is to be the human-readable alternative name of the resource. With any luck, a resource will have an `rdfs:comment` property telling you more.

If you get frustrated that not enough resources have `rdfs:label` values, learn from their creators' mistakes: make sure to include them with your own RDF data.

Let's say I'm wondering about what the various instances of the Good Relations `gr:BusinessFunction` class are. The following query lists their `rdfs:label` values, their `rdfs:comment` values if available, and the resource URIs themselves:

```
# filename: ex459.rq

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX gr:   <http://purl.org/goodrelations/v1#>

SELECT *
WHERE
{
   ?bf a gr:BusinessFunction ;
       rdfs:label ?label .
   OPTIONAL { ?bf rdfs:comment ?comment }
}
```

When run with the Good Relations ontology, we get plenty of output. (The `rdfs:comment` values are long enough that I moved them to their own line and added carriage returns. ARQ converted the carriage returns it found in the data into the "\n" characters you see here.)

```
-------------------------------------------------------------------------
| bf                          | label
| comment
=========================================================================

| gr:Dispose                  | "Dispose (business function)"@en
| "This gr:BusinessFunction indicates that the gr:BusinessEntity offers (or
  seeks) the acceptance of the specified gr:ProductOrService for proper
  disposal, recycling, or any other kind of allowed usages, freeing the
  current owner from all rights and obligations of ownership."@en

| gr:Sell                     | "Sell (business function)"@en
| "This gr:BusinessFunction indicates that the gr:BusinessEntity offers to
  permanently transfer all property rights on the specified
  gr:ProductOrService."@en

| gr:Repair                   | "Repair (business function)"@en
| "This gr:BusinessFunction indicates that the gr:BusinessEntity offers (or
  seeks) the evaluation of the chances for repairing, and, if positive, repair
  of the specified gr:ProductOrService. Repairing means actions that restore
  the originally intended function of a product that suffers from outage or
  malfunction."@en

| gr:Maintain                 | "Maintain (business function)"@en
| "This gr:BusinessFunction indicates that the gr:BusinessEntity offers (or
  seeks) typical maintenance tasks for the specified
  gr:ProductOrService. Maintenance tasks are actions that undo or compensate
  for wear or other deterioriation caused by regular usage, in order to
  restore the originally intended function of the product, or to prevent
  outage or malfunction."@en

| gr:ProvideService           | "Provide service (business function)"@en
| "This gr:BusinessFunction indicates that the gr:BusinessEntity offers (or
  seeks) the respective type of service.\n\nNote: Maintain and Repair are also
  types of Services. However, products and services ontologies often provide
  classes for tangible products as well as for types of services. The business
  function gr:ProvideService is to be used with such goods that are services,
  while gr:Maintain and gr:Repair can be used with goods for which only the
  class of product exists in the ontology, but not the respective type of
  service.\n\nExample: Car maintenance could be expressed both as \"provide
  the service car maintenance\" or \"maintain cars\"."@en

| gr:ConstructionInstallation | "Construction / installation (business function)"@en
| "This gr:BusinessFunction indicates that the gr:BusinessEntity offers (or
  seeks) the construction and/or installation of the specified
  gr:ProductOrService at the customer's location."@en

| gr:Buy                      | "Buy (business function, DEPRECATED)"@en
| "This gr:BusinessFunction indicates that the gr:BusinessEntity is in general
  interested in purchasing the specified gr:ProductOrService.\nDEPRECATED. Use
  gr:seeks instead."@en

| gr:LeaseOut                 | "Lease Out (business function)"@en
| "This gr:BusinessFunction indicates that the gr:BusinessEntity offers (or
  seeks) the temporary right to use the specified gr:ProductOrService."@en
```

Good Relations is obviously a well-documented ontology.

Discussion

While all of the `gr:BusinessFunction` instances found by the example query have `rdfs:comment` values as well as `rdfs:label` values, the query does not assume that the `rdfs:comment` values will be there. The triple pattern for that is in an OPTIONAL graph pattern so that if a resource has an `rdfs:label` value but no `rdfs:comment` value, the `rdfs:label` value will still appear in the result set.

> Don't assume that resources with `rdfs:label` values also have `rdfs:comment` values.

What if you want to know which classes have instances with `rdfs:label` values? You can find out with a pretty simple query that asks, for each resource that has an `rdfs:label` value, what class it belongs to:

```
# filename: ex461.rq

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

SELECT DISTINCT ?class
WHERE
{
   ?s rdfs:label ?label ;
      a ?class .
}
```

Here are the Good Relations classes whose instances have `rdfs:label` values:

```
-----------------------------------------------------------------
| class                                                         |
=================================================================
| <http://www.w3.org/2002/07/owl#ObjectProperty>                |
| <http://www.w3.org/2002/07/owl#SymmetricProperty>             |
| <http://www.w3.org/2002/07/owl#TransitiveProperty>            |
| <http://purl.org/goodrelations/v1#PaymentMethod>              |
| <http://www.w3.org/2002/07/owl#DatatypeProperty>              |
| <http://purl.org/goodrelations/v1#BusinessFunction>           |
| <http://purl.org/goodrelations/v1#PaymentMethodCreditCard>    |
| <http://purl.org/goodrelations/v1#BusinessEntityType>         |
| <http://www.w3.org/2002/07/owl#Class>                         |
| <http://purl.org/goodrelations/v1#DeliveryModeParcelService> |
| <http://purl.org/goodrelations/v1#DayOfWeek>                  |
| <http://purl.org/goodrelations/v1#DeliveryMethod>             |
| <http://purl.org/goodrelations/v1#WarrantyScope>              |
| <http://www.w3.org/2002/07/owl#Ontology>                      |
| <http://www.w3.org/2002/07/owl#AnnotationProperty>            |
-----------------------------------------------------------------
```

See Also

- "What Data Is Stored About a Class's Instances?" on page 324
- "Which Classes Have Instances?" on page 313
- "A Certain Property's Values Are Resources. What Data Do We Have About Them?" on page 328

What Data Is Stored About a Class's Instances?

Problem

The flexibility of RDF technology means that we don't have to declare a class's properties before we start creating instances of that class. Even if we do add these declarations, they're just guidelines—they're not rules whose violation means a broken system, like schema declarations are with a relational database or XML-based system.

And, if lots of properties are associated with a given class using the `rdfs:domain` property, that doesn't mean that they're all used. So how do we find out which properties are used for a class's instances in a given dataset? With a simple query.

Solution

The FOAF specification lists many properties that can be associated with instances of the `foaf:Person` class. Which does the VoID ontology use? We can find out by asking, for each each instance of the `foaf:Person` class, what the predicates are in the triples describing those instances:

```
# filename: ex463.rq

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

```
SELECT DISTINCT ?property
WHERE
{
  ?person a foaf:Person ;
          ?property ?value .
}
```

(Note the use of the DISTINCT keyword to indicate that we only want each property name listed once.) The result shows that VoID only has a bit of contact information about each of the `foaf:Person` instances stored in the ontology:

```
-----------------------------------------------------
| property                                          |
=====================================================
| foaf:mbox                                         |
| foaf:homepage                                     |
| foaf:name                                         |
| <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> |
-----------------------------------------------------
```

Discussion

The last row of the results shouldn't be a surprise: asking about resources that are members of the `foaf:Person` class is the same as asking about resources that have an `rdf:type` value of `foaf:Person`, so obviously `rdf:type` is one of the properties each will have.

If you do want to know which properties have been associated with a class by a schema, ask which properties have that class as their domain. Although it uses FOAF properties, the VoID ontology won't tell us which of them have a domain of `foaf:Person` because that's the FOAF ontology's job. We can ask the VoID ontology about which properties have been associated with one of its own classes, `void:Linkset`, with the following query:

```
# filename: ex465.rq

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

SELECT *
WHERE
{ ?property rdfs:domain void:Linkset }
```

Running this query shows that there are four of these properties:

```
-----------------------
| property            |
=======================
| void:objectsTarget  |
| void:subjectsTarget |
| void:linkPredicate  |
| void:target         |
-----------------------
```

See Also

- "Which Classes Use a Particular Property?" on page 316
- "Which Classes Have Instances?" on page 313
- "A Given Class Has Lots of Instances. What Are These Things?" on page 321

What Values Does a Given Property Have?

Problem

If you're wondering what a given property is for, or how it's used, a good clue is to find out what different values it has. You can do this with a very simple query.

Solution

Let's say that after I see the `gr:hasNext` property defined in Good Relations, I wonder "has next what"? Of course, the `rdfs:comment` value for the property itself is a good place to check, but a list of the values used for this property can tell us more. The following query will tell us:

```
# filename: ex467.rq

PREFIX gr: <http://purl.org/goodrelations/v1#>

SELECT DISTINCT ?value
WHERE
{ ?s gr:hasNext ?value }
```

The query result shows that the values are the days of the week:

```
----------------
| value        |
================
| gr:Saturday  |
| gr:Sunday    |
| gr:Tuesday   |
| gr:Monday    |
| gr:Wednesday |
| gr:Friday    |
| gr:Thursday  |
----------------
```

Note that the values are prefixed names and therefore resources with their own URIs instead of being string value representations of day names like "Sunday". Doing this lets an application use strings from different languages such as "Sonntag" or "domingo" if necessary.

Discussion

The DISTINCT keyword prevents the SPARQL processor from showing duplicate results. This can be handy if more than 40 or 50 triples use this property and have many repeats.

This query can also be useful without the DISTINCT keyword, though—when you look at *all* the values stored for a given property, you might see that some are used more often than others. (In these cases, adding an ORDER BY line at the end to sort the output will make this easier to see.) Removing the DISTINCT keyword from ex467.rq won't make any difference when querying the Good Relations ontology about the `gr:hasNext` property, but it just might with the next dataset that you work with.

Querying for the `rdfs:range` setting of a particular property can also tell you a lot about that property's potential values. The following asks about the `rdfs:range` of the Good Relations `gr:hasNext` property:

```
# filename: ex469.rq

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX gr: <http://purl.org/goodrelations/v1#>

SELECT *
WHERE
{ gr:hasNext rdfs:range ?rangeValue . }
```

The result shows us that a `gr:hasNext` value is an instance of the `gr:DayOfWeek` class that the ontology declares:

```
----------------
| rangeValue   |
================
| gr:DayOfWeek |
----------------
```

For a variation on this, if you modify this query to ask the Good Relations ontology about its `gr:amountOfThisGood` property, you'll see that its values have a type of `xsd:float`—a floating point number.

See Also

- "How Much Was a Given Property Used?" on page 317
- "What Properties Are Used?" on page 314
- "A Certain Property's Values Are Resources. What Data Do We Have About Them?" on page 328
- "Which Data or Property Name Includes a Certain Substring?" on page 334

A Certain Property's Values Are Resources. What Data Do We Have About Them?

Problem

Looking at some triples, you often see that a given property has URIs for values. URIs can be pretty cryptic (as an example, see the `dc:creator` values in void.ttl) and you may well wonder what those things are.

Solution

In many cases, checking for `rdfs:label` values attached to these resources would be a good next step, but as it turns out, the void.ttl ontology's `dc:creator` resources have no `rdfs:label` values. A simple query can find out exactly which data is associated with the `dc:creator` values:

```
# filename: ex471.rq

PREFIX dc:   <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX void: <http://vocab.deri.ie/void#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?creator ?propertyName ?propertyValue
WHERE
{
  ?s dc:creator ?creator .
  ?creator ?propertyName ?propertyValue .
}
```

This query declares the `dc:` prefix because the graph pattern uses this prefix. I included the other prefix declarations because I knew that those URIs would show up in the search results, and I wanted to make the results narrow enough to fit on the page. Even then, I had to put a carriage return after each `?creator` value in the following results to make them fit; if you run this example, you'll see each result appear on one line instead of two. I also replaced the high-level domain of each result email address in the output with "xyz" because I'd hate to be responsible for the VoID creators getting any extra spam.

The result shows that each `dc:creator` has values for the `foaf:mbox`, `foaf:homepage`, `foaf:name`, and `rdf:type` properties:

```
--------------------------------------------------------
| creator
| propertyName  | propertyValue                        |
========================================================
| <http://vocab.deri.ie/void#Michael%20Hausenblas>
| foaf:mbox     | <mailto:michael.hausenblas@deri.xyz> |
| <http://vocab.deri.ie/void#Michael%20Hausenblas>
| foaf:homepage | <http://sw-app.org/about.html>       |
| <http://vocab.deri.ie/void#Michael%20Hausenblas>
| foaf:name     | "Michael Hausenblas"                 |
| <http://vocab.deri.ie/void#Michael%20Hausenblas>
| rdf:type      | foaf:Person                          |
| void:keiale
| foaf:mbox     | <mailto:Keith.Alexander@talis.xyz>   |
| void:keiale
| foaf:homepage | <http://kwijibo.talis.com/>          |
| void:keiale
| foaf:name     | "Keith Alexander"                    |
| void:keiale
| rdf:type      | foaf:Person                          |
| void:junzha
| foaf:mbox     | <mailto:jun.zhao@zoo.ox.ac.xyz>      |
| void:junzha
| foaf:homepage | <http://users.ox.ac.uk/~zool0770/>   |
| void:junzha
| foaf:name     | "Jun Zhao"                           |
| void:junzha
| rdf:type      | foaf:Person                          |
| void:cygri
| foaf:mbox     | <mailto:richard.cyganiak@deri.xyz>   |
| void:cygri
| foaf:homepage | <http://richard.cyganiak.de/>        |
| void:cygri
| foaf:name     | "Richard Cyganiak"                   |
| void:cygri
| rdf:type      | foaf:Person                          |
--------------------------------------------------------
```

Discussion

When you see URIs as the objects of triples, remember that this is one of the great things about RDF: when a given resource's URI can be the subject of some triples and the object of others (and perhaps the predicate of others!), it means that the data can more easily connect with other data, letting you and your applications find additional relevant data. When property values are resources, those resources may have data about them, and that data may point at other resources that have additional data about them, and so forth. These are the links of Linked Data. Exploring such paths with interesting data is one way to have some fun with SPARQL.

See Also

- "A Given Class Has Lots of Instances. What Are These Things?" on page 321
- "What Values Does a Given Property Have?" on page 326

How Do I Find Undeclared Properties?

Problem

"Problem" may be a strong word here—the fact that RDF doesn't require you to declare properties before using them is a great benefit because it makes it easier to store and aggregate small, granular bits of data. Because a triple's predicate is expressed as a URI, an RDF parser can work with it, whether it's heard of the property used in that predicate or not.

If the parser knows something about that predicate, though, it can do more with it. If an application knows that a particular property is sometimes used to express data for a particular class of resources, it can use that property to generate forms, reports, and other application components for instances of that class. If it knows that values of that property are always members of a particular class, or values of a specific type such as boolean or integer, it can create even better forms for instances of the class that use that property, guiding users to select appropriate values instead of letting them enter anything they want.

If an application knows that a property that it's never seen before is a subproperty of one that it knows about, it may know some useful things that it can do with it. For example, if the unrecognized `foo:glassBlower` property is a subproperty of the Dublin Core `dc:creator` property, the application knows that it can treat `foo:glassBlower` values the same way it treats `dc:creator` values, perhaps putting them in a "Creator" column of a report.

If your triples include predicates with undeclared properties, how can you find them, and then once you find them, what do you do about it?

Solution

The following query asks for a list of all predicates that haven't been declared to be of some type (remember, in SPARQL and in Turtle, "a" is a synonym for `rdf:type`):

```
# filename: ex485.rq

SELECT DISTINCT ?p
WHERE
{
  ?s ?p ?o .
  MINUS { ?p a ?someType }
}
ORDER BY ?p
```

The query essentially says, "give me a list of all the predicates... except for the ones that are declared to be of some type."

Discussion

The type of the property used as the predicate (which this query stores in the `?someType` variable) will probably be `rdf:Property` or some subproperty of it, such as the `owl:DatatypeProperty` or `owl:ObjectProperty` classes declared as part of the OWL standard. The type may also be some subproperty of one of these, declared as a specialized customization.

For now, though, we're more interested in the properties that don't have any type that we know of. Let's look at the output of this query when we run it against the VoID ontology:

```
--------------------------------------------------------
| p                                                    |
========================================================
| <http://purl.org/dc/elements/1.1/creator>            |
| <http://purl.org/dc/terms/FileFormat>                |
| <http://purl.org/dc/terms/created>                   |
| <http://purl.org/dc/terms/description>               |
| <http://purl.org/dc/terms/modified>                  |
| <http://purl.org/dc/terms/partOf>                    |
| <http://purl.org/dc/terms/publisher>                 |
| <http://purl.org/dc/terms/status>                    |
| <http://purl.org/dc/terms/title>                     |
| <http://purl.org/dc/terms/type>                      |
| <http://purl.org/vocab/vann/preferredNamespacePrefix> |
| <http://purl.org/vocab/vann/preferredNamespaceUri>   |
| <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>    |
| <http://www.w3.org/2000/01/rdf-schema#comment>       |
| <http://www.w3.org/2000/01/rdf-schema#domain>        |
| <http://www.w3.org/2000/01/rdf-schema#label>         |
| <http://www.w3.org/2000/01/rdf-schema#range>         |
| <http://www.w3.org/2000/01/rdf-schema#subClassOf>    |
| <http://www.w3.org/2000/01/rdf-schema#subPropertyOf> |
| <http://www.w3.org/ns/adms#accessURL>                |
| <http://www.w3.org/ns/adms#status>                   |
| <http://xmlns.com/foaf/0.1/homepage>                 |
| <http://xmlns.com/foaf/0.1/mbox>                     |
| <http://xmlns.com/foaf/0.1/member>                   |
| <http://xmlns.com/foaf/0.1/name>                     |
--------------------------------------------------------
```

When we see popular base URIs such as *http://purl.org/dc/elements/1.1/* and *http://xmlns.com/foaf/0.1/*, we know that we can't say that these predicates have never been declared anywhere, but only that the SPARQL engine didn't see any declarations for them. If you get the schemas or ontologies that declare the properties shown in this query output (in this case, Dublin Core, Dublin Core Metadata Terms, the VANN vocabulary for annotating vocabulary descriptions, the Asset Description Metadata Schema, RDF Schema, and FOAF) and include the triples from those schemas and ontologies in the data that you're querying, you'll find out much more about these properties.

To find a given schema or ontology, try sending your browser to the base URI of one of the declared names. For example, to find out more about *http://www.w3.org/ns/adms#accessURL*, send your browser to *http://www.w3.org/ns/adms*.

What if the query finds properties that really haven't been declared anywhere? This is not unusual with Linked Open Data, where a dataset's sponsors may have been more concerned with publishing an existing data source than with creating and distributing an explicit model for the data. Utilities that you use to create RDF from non-RDF data sources (for example, from spreadsheets) might generate properties for the output predicates based on input data (for example, a spreadsheet's column headers) without ever declaring those properties.

You don't need SPARQL to create declarations; these are just more triples to add to your data. For example, if you see an *http://learningsparql.com/ns/demo#foobar* predicate and can't find a declaration for it, you can simply add the triple `{<http://learningsparql.com/ns/demo#foobar> a rdf:Property}` to your data. (If you want to do this with SPARQL, you can do it with a CONSTRUCT query or an INSERT request.)

Declaring it as a more specific property type, such as `owl:DatatypeProperty` or `owl:ObjectProperty`, lets applications do more with that data. For example, if it looked like all of the `dm:foobar` values were URIs, then it's probably an object property. The following query would help you double-check by listing any `dm:foobar` values that aren't URIs before you declare this as an `owl:ObjectProperty`:

```
# filename: ex487.rq

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

SELECT ?o
WHERE
{
  ?s dm:foobar ?o .
  FILTER(!(isIRI(?o)))
}
```

If, on the other hand, none of its values looked like URIs and you were thinking of declaring it as an `owl:DatatypeProperty`, removing the exclamation point from the query above would give you a query that lists the objects from `dm:foobar` triples that *are* URIs.

When declaring new properties that are specialized versions of existing ones (for example, declaring a new `foo:glassBlower` property as a specialized version of `dc:creator`), declare it as an `rdfs:subPropertyOf` of the existing one.

Declarations of properties and especially of metadata about them (such as whether they are subproperties of existing ones) is much easier when you use a tool such as TopBraid Composer or Protégé, where you use a graphical user interface to do your data modeling before the tool saves your work using one of the standard RDF syntaxes.

See Also

- "How Do I Change the Datatype of a Certain Property's Values?" on page 345
- "How Do I Replace One Property with Another?" on page 343

How Do I Treat a URI as a String?

Problem

URI can be cryptic, but they may contain useful information in the domain name, the path name, or the local name. Sometimes you want to pull a piece out of a URI to create a new one, and sometimes you want to search URIs for specific substrings. But, they're not strings—they're URIs.

Solution

The Good Relations vocabulary follows the best practice of assigning an `rdfs:label` value to all of the classes it declares, but let's pretend that it doesn't and you're wondering which classes may have information about prices. When you enclose a URI (or a variable storing one) in the `str()` function, the function returns a string version of the URI, letting you do anything to it that you might do with a string, such as calling the `CONTAINS()` or `regex()` functions. The following lists all the URIs that have the word "price" in them, in any combination of uppercase and lowercase:

```
# filename: ex489.rq

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT *
WHERE
{
  ?s a owl:Class .
  FILTER(CONTAINS(LCASE(str(?s)),"price"))
}
```

When run with the Good Relations vocabulary, we find that two class URIs have this as a substring:

```
--------------------------------------------------------------
| s                                                          |
==============================================================
| <http://purl.org/goodrelations/v1#PriceSpecification>     |
| <http://purl.org/goodrelations/v1#UnitPriceSpecification> |
--------------------------------------------------------------
```

Discussion

For demonstration purposes, I pretended that Good Relations did not assign `rdfs:label` values to class names. If I really wondered about which Good Relations classes might be related to prices, I would have run a similar query that searched the `rdfs:label` values—although there would be no need to use the `str()` function, because `rdfs:label` values should always be strings.

You may well find RDFS schemas and OWL ontologies that do not follow this best practice, and this query demonstrates a handy technique for exploring such vocabularies. Once you use the `str()` function to treat the URI as a string, you can do more than search its contents with `CONTAINS()`; you can use any combination of string-related functions (and remember, SPARQL 1.1 added some great new ones) to scan, split and combine these strings.

See Also

- "Which Data or Property Name Includes a Certain Substring?" on page 334
- "How Do I Convert a String to a URI?" on page 336

Which Data or Property Name Includes a Certain Substring?

Problem

What if you're looking for something, and you're not sure whether to look in a dataset's URIs or strings? Maybe a given dataset mentions something you're interested in, but you're not sure where. You just want to search the whole thing for a particular string. For example, let's say we want to look for the string "publish" anywhere in a dataset.

Solution

The following query asks for all the triples in a dataset (think about the size of your dataset before making such a huge request!) and then uses a FILTER expression to say that we only want triples whose objects have the string "publish" in them somewhere:

```
# filename: ex473.rq

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

```
    FILTER (CONTAINS(LCASE(str(?o)), "publish"))
  }
```

This query gets results when you run it with both the VoID ontology file (void.ttl) and the Good Relations ontology file (goodrelations-v1.owl).

To search property names for a substring—in this case, if you were looking for property URIs with "publish" somewhere in their name—substitute `str(?p)` for `str(?o)` in the query.

Discussion

We saw in "How Do I Treat a URI as a String?" on page 333 that the `str()` function lets you treat a URI as a string. Passing a string as the argument to this function won't do anything—you're essentially saying "treat this string as a string"—but it won't cause an error, either. This is actually quite handy, because it means that the query above can pass all the potential values of `?o` to this function without worrying about whether `?o` is a URI, a string, or any other type.

You have a lot of flexibility with this query:

- The query's `str()` function call converts the passed value to a string before passing it to the `LCASE()` function. This is not completely necessary—especially if you're only searching values of a specific property such as `rdfs:label` or `rdfs:property` that you know will always be strings—but including the call to the `str()` function will let `CONTAINS()` act on URI values as well, checking whether they have the string "publish" in them somewhere.
- When searching for a substring in an object value, instead of the variable `?p` in the one triple pattern's predicate position, if you name a specific property, the search will be faster. For example, `rdfs:label`, `rdfs:comment`, and `skos:prefLabel` often have interesting text in them and are worth searching. (This will also be a faster, more efficient search.)
- The `LCASE()` function converts the `?o` value to lowercase before comparing it with the string "publish"; if you're only interested in finding a lowercase version of the string (or in some other specific combination of uppercase and lowercase letters in the string), omit this function call and your search will go a little quicker.
- Instead of using the `CONTAINS()` and `LCASE()` function calls, you could use `regex()`, which gives you more flexibility but could be a bit slower.

See Also

- "How Do I Treat a URI as a String?" on page 333
- "What Values Does a Given Property Have?" on page 326

How Do I Convert a String to a URI?

Problem

Plenty of tools can convert non-RDF data to RDF for you, but that's often just a start. Turning that data into the RDF that you want can mean various cleanup steps, including the creation of new URIs to use in your subjects and predicates—but how do you create those URIs?

For example, let's say you want to create new URIs to represent the people in the following dataset, basing those URIs on a combination of each person's first and last name:

```
# 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" .
```

Inserting actual data values into a URI can lead to several problems. In this case, if someone changes her name, having her old name in the URI can be misleading. And, if you have two people with the same first and last name, you need a way to give them two different URIs. When you create URIs from string values, you're more likely to use other kinds of data such as an ID value assigned to a resource on a legacy sytem—for example, to represent an employee, you might use an employee ID, if your SPARQL query has access to it.

Solution

There are two basic steps to creating a URI from a string. Each takes advantage of a function that was new with SPARQL 1.1:

1. Use `CONCAT()` (and other string manipulation and casting functions if necessary) to create a string that looks like the URI that you want.
2. Use the `IRI()` function—or its synonym, `URI()`—to convert that string to a URI.

The following query binds the `ab:firstName` and `ab:lastName` values to the variables `?fn` and `?ln` and then builds URIs from those values and the `?baseURI` string shown. It then creates new triples that use the generated URIs as subjects:

```
# filename: ex491.rq

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

CONSTRUCT
{
  ?personURI ab:firstName ?fn ;
             ab:lastName ?ln ;
             ab:email ?email ;
             ab:homeTel ?tel .
}
WHERE {
  ?person ab:firstName ?fn ;
          ab:lastName ?ln ;
          ab:email ?email .
  OPTIONAL {
    ?person ab:homeTel ?tel .
  }
  BIND(ENCODE_FOR_URI(CONCAT(?fn,?ln)) AS ?encodedName)
  BIND("http://learningsparql.com/new/namespace/" AS ?baseURI)
  BIND(URI(CONCAT(?baseURI,?encodedName)) AS ?personURI)
}
```

The first BIND statement, in addition to concatenating the first and last name values together, calls the `ENCODE_FOR_URI()` function to escape any potentially URI-unfriendly characters. For example, if someone has a first name of “Mary Beth” and a last name of “Smith”, this line will put “Mary%20BethSmith” in the `?encodedName` variable, because you don’t want the space between “Mary” and “Beth” to be part of the URI that you’re creating.

The third BIND statement, after concatenating the base URI and the encoded name into a string version of the URI that we want, uses the `URI()` function to turn that into a proper URI that can then be used as the subject of the CONSTRUCTed triples.

Discussion

In the example above, if the URIs that get created by the BIND statements are not legal RDF URIs, you’ll know it, because the CONSTRUCT clause won’t work.

All the function calls in the example’s BIND statements could have been nested together and used with a single BIND statement, but when they’re spread out over three BIND statements, the query is easier to read and to fit on the page. It’s also easier to debug, especially if the early drafts of the query use the SELECT keyword instead of the CONSTRUCT keyword so that you can make sure you’re creating the values that you planned to.

For example, if the CONSTRUCT clause isn't creating any triples, and you suspect that the URIs being created aren't quite right, a SELECT statement that checks on the value of `?encodedName` may show you the problem. See "Manual Debugging" on page 227 for more on this.

See Also

- "How Do I Treat a URI as a String?" on page 333

How Do I Query a Remote Endpoint?

Problem

Most SPARQL queries that we've looked at are designed to be handed directly to a SPARQL engine such as ARQ or one that's built into a triplestore. The Linked Data Cloud offers many SPARQL endpoints (web services that accept SPARQL queries) with data that can help your applications. SPARQL endpoints also play a growing role behind company firewalls, where they make data more easily available within an enterprise without sharing it with the rest of the world over the public web.

A SPARQL endpoint's documentation will tell you the URL where you should send queries. What's the simplest way to send a query to an endpoint at a given URL?

Solution

There are tools, such as the SNORQL tool included with DBpedia and other endpoints, that let you enter a query into a web form and then pass that query to the web service that executes the query against the data behind that endpoint. You can also embed the query in a URL built around the endpoint's URL, as described in "SPARQL and Web Application Development" on page 282.

A third option is to locally execute a query that uses the SPARQL SERVICE keyword to SELECT data from a remote endpoint. After the SERVICE keyword, put the endpoint URI followed by the graph pattern describing the triples that you want retrieved from that endpoint.

For example, the following query asks DBpedia for all of its triples about the British minister and novelist Joseph Hocking:

```
# filename: ex474.rq

SELECT ?p ?o
WHERE
{
  SERVICE <http://DBpedia.org/sparql>
    { <http://dbpedia.org/resource/Joseph_Hocking> ?p ?o . }
}
```

As described in "Querying a Remote SPARQL Service"on page 102, the ARQ command-line tool expects you to specify data to query on the command line even if the query will ignore that particular data. So, remember to include a `--data` parameter on the ARQ command line, with whatever RDF file you want, when trying this query.

Discussion

You can use the SERVICE keyword more than once in the same query, which lets you build federated queries. The following query, in addition to asking DBpedia for triples about Joseph Hocking, also asks Project Gutenberg for its triples about him:

```
# filename: ex475.rq

PREFIX gp:   <http://wifo5-04.informatik.uni-mannheim.de/gutendata/resource/people/>

SELECT ?dbpProperty ?dbpValue ?gutenProperty ?gutenValue
WHERE
{
  SERVICE <http://DBpedia.org/sparql>
    {
      <http://dbpedia.org/resource/Joseph_Hocking> ?dbpProperty ?dbpValue .
    }

  SERVICE <http://wifo5-04.informatik.uni-mannheim.de/gutendata/sparql>
    {
      gp:Hocking_Joseph ?gutenProperty ?gutenValue .
    }
}
```

The results of this query will actually be a cross-product of the two sets of retrieved results; see "Federated Queries: Searching Multiple Datasets with One Query" on page 105 for more on this.

See Also

- "How Do I Retrieve Triples from a Remote Endpoint?" on page 339

How Do I Retrieve Triples from a Remote Endpoint?

Problem

When you query a remote endpoint, maybe you don't want to just see the results listed in columns—maybe you want to pull down the actual triples from the remote endpoint so that you can store them locally, or perhaps create new triples locally from the information on the remote server.

Solution

As we saw in "Copying Data" on page 111, the creation of new triples often means taking a SELECT query that lists the information you want and replacing the SELECT clause with a CONSTRUCT clause that includes a graph pattern showing what you want your new triples to look like.

For example, let's say we want to retrieve all of DBpedia's triples about the British minister and novelist Joseph Hocking. If we take the first SELECT query in the Solution section of "How Do I Query a Remote Endpoint?" on page 338, we can remove the SELECT keyword and variable list and replace them with the bolded text in the following query:

```
# filename: ex476.rq

CONSTRUCT
{ <http://dbpedia.org/resource/Joseph_Hocking> ?p ?o . }
WHERE
{
  SERVICE <http://DBpedia.org/sparql>
   { <http://dbpedia.org/resource/Joseph_Hocking> ?p ?o . }
}
```

ARQ will output the result triples in Turtle, while other tools may have other ways of delivering these triples to you. For an application like ARQ that just displays the triples as command-line output, you can redirect the output to a file where later steps in your processing pipeline can get at it.

Discussion

Make sure to read all of "Copying Data" on page 111 to learn more about the role that the SERVICE keyword can play in your query. The section after that one, "Creating New Data" on page 115, brings up another important point: that in addition to letting you copy triples, the CONSTRUCT keyword lets you create new triples from retrieved data. This ability to build something new from existing data opens up some great possibilities when you retrieve data from multiple sources, letting you create a whole that is greater than the sum of its parts.

Even the straight copying of remote triples for local storage can offer some nice advantages for your applications:

- If you're not absolutely sure that the remote source will be available whenever you want it, storing local copies ensures that you'll always have that data when you need it.
- Locally stored triples will be faster to process than remote ones.

- If you've pulled data from multiple sources, having it together in one place makes it easier to identify potential connections within the data than it would be if the data were spread across multiple remote servers.

See Also

- "How Do I Query a Remote Endpoint?" on page 338

Creating and Updating Data

This section covers typical patterns of updating data, especially making common changes to multiple triples.

These queries all assume that you have write access to the data that you want to change.

How Do I Delete All the Data?

Problem

Let's say you're trying out a new triplestore that you've just installed, and you've added some sample data and want to start over—you want to completely clear out all the triples that are there.

Solution

"Deleting Data" on page 194 describes how to delete specific triples from the default graph, and "Dropping Graphs" on page 204 describes how to delete entire graphs. The latter section includes the following command, which deletes the default graph, all named graphs, and their contents:

```
# filename: ex337.ru

DROP ALL
```

Discussion

This is a very powerful command, and with great power comes great responsibility.

Take a look at any features that your system (for example, your triplestore) may offer to back up the data before executing a command like this. The backup may be as simple as running a query from "How Do I Look at All the Data at Once?" on page 306 and saving the output.

If you don't have any named graphs, a CONSTRUCT version of the first query in that section would give you all the data as RDF. If you have triples in named graphs and need to save those names with the triples, there is so far no standard way to save that as RDF; one approach would be to run the second query from that section and save the results in SPARQL Query Results XML format. An XSLT stylesheet or other XML-based tool could convert this into a series of SPARQL update requests that load this data into your storage tool if necessary. Saving the results as SPARQL Query Results JSON format and using a JSON-friendly scripting language is another option, as is saving the "quads" (triples plus graph names) in a TSV file, as described in "TSV Query Results" on page 251.

See Also

- "How Do I Look at All the Data at Once?" on page 306

How Do I Globally Replace a Property Value?

Problem

A group of resources in your dataset have a particular value for a certain property, and you want to change all of these values to a new one. For example, let's say we want to update all of the 2013 copyright dates below to 2014, while leaving the others alone:

```
# filename: ex477.ttl

@prefix dm:      <http://learningsparql.com/ns/demo#> .
@prefix dc:      <http://purl.org/dc/elements/1.1/> .
@prefix dcterms: <http://purl.org/dc/terms/> .

dm:i590 dc:title "Installation Guide" ; dcterms:dateCopyrighted "2012" .
dm:i591 dc:title "Users Guide" ; dcterms:dateCopyrighted "2013" .
dm:i592 dc:title "API Reference" ; dcterms:dateCopyrighted "2013" .
dm:i593 dc:title "Tutorial" ; dcterms:dateCopyrighted "2011" .
dm:i594 dc:title "Reference Guide" ; dcterms:dateCopyrighted "2013" .
```

Solution

Use a WHERE clause to name the triples you want to replace, a DELETE clause to delete them, and an INSERT clause to assign the new value to the same property for each resource that the WHERE clause found:

```
# filename: ex478.ru

PREFIX dcterms: <http://purl.org/dc/terms/>

DELETE {?document dcterms:dateCopyrighted "2013" . }
INSERT {?document dcterms:dateCopyrighted "2014" . }
WHERE  {?document dcterms:dateCopyrighted "2013" . }
```

Discussion

Note that the only difference between the triples being found (and deleted) and the ones being inserted is the values of their objects.

> As a safety measure, for this and all the queries in this chapter that will change your data, first try some SELECT or CONSTRUCT queries that show what data will be affected. The section "Themes and Variations" on page 303 at the beginning of this chapter shows the appropriate variations on the query above.

See Also

- "How Do I Replace One Property with Another?" on page 343
- "How Do I Change the Datatype of a Certain Property's Values?" on page 345

How Do I Replace One Property with Another?

Problem

Sometimes you might use a certain property in your data and then learn about another equivalent one that you'd rather use. Perhaps the other one is more popular and would give your data more interoperability with other datasets, or perhaps the other property fits better with recent additions to your data model.

For example, let's say that we're going to use the vCard vocabulary to add additional postal address information to the people listed in the VoID ontology, but before we do so, we want to replace its triples that use the `foaf:mbox` property with equivalent ones that use vCard's `v:email` property.

Solution

In "How Do I Globally Replace a Property Value?" on page 342, the solution was to select some triples, delete them, and replace them with what were essentially copies that had a new value in the object position. Here, we're going to do something similar, but we'll be replacing the predicates instead of the objects.

In the following code, the WHERE clause identifies the triples we're interested in and the DELETE clause deletes them. The INSERT clause adds new triples that have the same subjects and objects as the deleted ones, but they use the new property instead of the one from the deleted triples:

```
# filename: ex482.ru

PREFIX v:    <http://www.w3.org/2006/vcard/ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

DELETE { ?s foaf:mbox ?o . }
INSERT { ?s v:email ?o . }
WHERE  { ?s foaf:mbox ?o . }
```

Discussion

In "Converting Data" on page 120, we saw how a CONSTRUCT query can copy a set of triples but replace certain properties with those from another vocabulary—in that case, replacing the address book properties that I made up for this book with the equivalent properties in the vCard standard. You can use the technique described here to create a variation on that section's vCard CONSTRUCT query that replaces the existing data with the new data:

```
# filename: ex481.ru

PREFIX ab: <http://learningsparql.com/ns/addressbook#>
PREFIX v:  <http://www.w3.org/2006/vcard/ns#>

DELETE
{
 ?s ab:firstName  ?firstName ;
    ab:lastName   ?lastName ;
    ab:email      ?email .
    ?s ab:homeTel ?homeTel .
}
INSERT
{
 ?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 . }
}
```

This one INSERTs the triples that were originally CONSTRUCTed and DELETEs the triples that the WHERE clause found.

While the DELETE clause looks nearly the same as the WHERE clause, note that its `ab:homeTel` triple pattern is not enclosed in an OPTIONAL clause.

If you need to do this for a lot of different properties, it might be tempting to do it all with one SPARQL query. This can be more trouble to develop, test, and debug. The trouble can be worth it if you'll be running this query often—for example, if you have a weekly feed of data from a supplier that must be converted to fit with your own model every time. On the other hand, if you're only doing this once to a single batch of data, an incremental approach of changing a manageable subset of triples at a time with simple queries like the one above can be safer.

See Also

- "How Do I Globally Replace a Property Value?" on page 342
- "How Do I Find Undeclared Properties?" on page 330

How Do I Change the Datatype of a Certain Property's Values?

Problem

Sometimes a process or software tool expects the values of a certain property to be of a particular datatype, and you have triples (perhaps retrieved from somewhere else) where this property's values don't always have this type. How can you do a global replacement that changes the type of those values while leaving the values alone?

Solution

The following data shows the names and amounts of a few items, and we want all the amounts to be integers. The first two already are: the first one includes an explicit type, and the second one, being numeric digits with no quotation marks around it, is expressed using the Turtle (and SPARQL) shortcut for `xsd:integer`. The third item's `dm:amount` value is explicitly typed as a string, and the fourth is an untyped literal.

Current plans for RDF 1.1 will make the string type the default, so that the `"12"` with `d:item4` that follows would be the same as `"12"^^xsd:string`, but the important point for this example is that it's not represented as an integer.

```
# filename: ex483.ttl

@prefix dm:   <http://learningsparql.com/ns/demo#> .
@prefix d:    <http://learningsparql.com/ns/data#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

d:item1 rdfs:label "kerosene, 1 quart" ;
        dm:amount "14"^^xsd:integer .

d:item2 rdfs:label "double-knit polyester vest" ;
        dm:amount 10 .

d:item3 rdfs:label "gold-plated chain" ;
        dm:amount "30"^^xsd:string .

d:item4 rdfs:label "reverse flange" ;
        dm:amount "12" .
```

The following update request replaces all the `dm:amount` triples with new ones that have the same subject, the same predicate, and an object with the same value as the object from the original but cast to an integer type:

```
# filename: ex484.ru

PREFIX dm:  <http://learningsparql.com/ns/demo#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

DELETE
{ ?s dm:amount ?amount }
INSERT
{ ?s dm:amount ?integerAmount }
WHERE
{
  ?s dm:amount ?amount .
  BIND (xsd:integer(?amount) AS ?integerAmount )
}
```

To remove a type assignment like `xsd:integer`, the `str()` function converts a value to an untyped literal.

Discussion

This is a nice technique for beating imported data into shape, but it assumes that the value being converted will make sense as an integer—if an item in the example data above had a `dm:amount` value of “hello”, the `xsd:integer()` function wouldn’t know what to do with it.

After running your conversion, a SELECT query that checks on the resulting `dm:amount` values would be a good idea.

See Also

- "How Do I Globally Replace a Property Value?" on page 342
- "How Do I Find Undeclared Properties?" on page 330

How Do I Turn Resources into Instances of Declared Classes?

Problem

When an RDF triple tells you about a property name and value associated with a particular resource, the resource doesn't have to be an instance of any class. As long as you can refer to the resource with a URI, you can add data to it.

If the resource is a member of a declared class, though, you can do more with its data, because metadata about the class can tell an application more about the resource. For example, the metadata can describe other properties that might be associated with the instance. This helps applications build interfaces (both end user interfaces and Application Programming Interfaces) around that resource and related ones.

Identifying your resources as instances of classes can also make your queries run more quickly, because the query engine can narrow down the search space faster if it knows that a resource in the result set belongs to a specific class. If I ask DBpedia to list everything that has "Apple" in its name, that can take a long time, but if I ask it to list all instances of the *http://dbpedia.org/ontology/Company* class that have "Apple" in their name, that happens much more quickly.

So, if you have resources that aren't instances of classes, how can you turn them into instances of classes, and what classes should they be instances of?

Solution

The question of what classes to make your resources instances of is really a data modeling issue, but you can get some clues by looking at the data that you already have about those instances. For example, we can see here that some resources have `ab:firstName`, `ab:lastName`, `ab:email` and `ab:takingCourse` properties, while others have `ab:courseTitle` properties:

```
# 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 .
```

It looks like the first category of resources are people, and the second category are courses. Since those people are taking those courses, we'll assume for the purposes of this dataset that the people are students, and make them instances of a `d:Student` class.

What if these resources might be members of other classes as well? Do we have to figure out a common superclass, potential subclasses, and the relationships between these classes up front before we do anything else? No, this is RDF; if we later say that resource `d:i0432` is also a member of a new Instructor class or a Musician class, we won't break anything.

The following query adds declarations for `d:Student` and `d:Course` classes. It also makes any resources with both `ab:firstName` and `ab:lastName` properties instances of the `d:Student` class, and it makes any resource with an `ab:courseTitle` property a member of the `d:Course` class:

```
# filename: ex488.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#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

INSERT
{
   d:Student a rdfs:Class .
   d:Course a rdfs:Class .
   ?student a d:Student .
   ?course a d:Course .
}
WHERE {

  ?student ab:firstName ?fn ;
           ab:lastName ?ln .

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

Discussion

Note that, although the query binds the first name, last name, and course title values to variables, it doesn't actually use them. It just needs to confirm that these resources have values for these properties because that was my criteria for identifying members of these classes.

I could have added an additional condition before making a resource a member of the `d:Student` class, requiring it to have a `d:takingCourse` value as well as `ab:firstName` and `ab:lastName` values. I pictured a course registration system in which people are only there because they're students, and I wanted someone who hasn't signed up for any courses yet to still be assigned to the `d:Student` class. These are the kinds of application needs that your query would have to take into account.

Assigning resources to classes based on the existence of certain properties, or even specific property values, is very common with SPARQL. When exploring data, it's a good way to identify patterns that you've found. Some people use RDFS and OWL inferencing to describe which resources are members of which classes, but this can often be done more simply with a SPARQL query.

See Also

- "Which Classes Have Instances?" on page 313

Summary

In this chapter, we learned about:

- Several handy variations that you can apply to multiple queries in this chapter
- How to explore an unknown dataset to see what kind of data it has, what kind of structure it defines, and how much of that structure it uses—and, how to take more advantage of defined structures by turning resources into instances of declared classes
- Several techniques to search for different kinds of things within a dataset
- Some useful basic queries for working with SPARQL endpoints
- How to delete all of a dataset's triples, and how globally replace values and data types

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 (`[]`). Tools that serialize triples do not have to save the prefixed name, as long as any new ones maintain all the same connections.

  See also graph, prefixed name, serialization.

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.

dataset
: The collection of graphs that a given SPARQL query is querying. This collection consists of a default graph and optional named graphs.

  See also default graph, named graph.

default graph
: The triples in an RDF dataset that don't belong to a named graph.

Dublin Core
: A popular standard vocabulary providing a basic set of metadata terms such as `title`, `creator`, and `date`. Many specialized metadata vocabularies are 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 was 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 an RDF 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, RDF does an excellent job at this.

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.

namespace
: A set of names grouped together for a common purpose such as describing a particular domain. The set of Dublin Core terms is one example of a namespace. In the RDF and XML worlds, a namespace is identified by 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 their 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.

  See also OWL.

OWL
: A W3C standard that builds on RDFS to let you define ontologies.

  See also 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 that includes a prefix to indicate the name's namespace. For example, if the prefix `dc` has been declared to represent the Dublin Core namespace represented by the URI *http://purl.org/dc/elements/1.1/*, then 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.

quad
: A four-part data structure consisting of a triple and, if the triple is part of a named graph, a URI representing the graph name.

  See also triple, named graph.

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 set of rules about structure and datatypes used for validation to ensure data quality and more efficient systems. In the RDF 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
: The process of retrieving the HTML of a web page such as an airline flight schedule or a weather report and then extracting the data 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.

semantic web
: A set of standards and best practices for sharing data and the semantics of that data over the Web for use by applications. The key standards are RDF, SPARQL, RDFS, and OWL.

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 and RDF/XML; Turtle's ancestor N3 makes occasional appearances, although the majority of N3 files are effectively Turtle files.

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 returns 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 over 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 RDF technology, 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 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 co-chairing 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 Media's *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.