*Querying and Updating with SPARQL 1.1*

2nd Edition

# *Learning* SPARQL

O'REILLY®

*Bob DuCharme*

# Learning SPARQL

Gain hands-on experience with SPARQL, the RDF query language that's bringing new possibilities to semantic web, linked data, and big data projects. This updated and expanded edition shows you how to use SPARQL 1.1 with a variety of tools to retrieve, manipulate, and federate data from the public web as well as from private sources.

With this book, you'll start writing queries right away. Author Bob DuCharme then shows you the bigger picture of how SPARQL fits into RDF technologies. Using short examples that you can run yourself with open source software, you'll learn how to update, add to, and delete data in RDF datasets.

- Get the big picture on RDF, linked data, and the semantic web
- Use SPARQL to find bad data and create new data from existing data
- Use datatype metadata and functions in your queries
- Learn techniques and tools to help your queries run more efficiently
- Use RDF Schemas and OWL ontologies to extend the power of your queries
- Discover the roles that SPARQL can play in your applications

Purchase the ebook edition of this O'Reilly title at oreilly.com and get free updates for the life of the edition. Our ebooks are optimized for several electronic formats, including PDF, EPUB, Mobi, and DAISY—all DRM-free.

*"I buy* Learning SPARQL *and give it away at every course I teach on the semantic web. It is a well-written and entirely practical way into this world. You can explore a significant portion of the concepts without having to set anything up. I have seen no better way to get started."*

**—Brian Sletten**
*Bosatsu Consulting*

**Strata**
Making Data Work

Strata is the emerging ecosystem of people, tools, and technologies that turn big data into smart decisions. Find information and resources at oreilly.com/data.

Twitter: @oreillymedia
facebook.com/oreilly

**US $39.99** **CAN $41.99**
ISBN: 978-1-449-37143-2


O'REILLY®
oreilly.com

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.

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

**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

# 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:

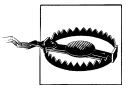
```
# filename: ex001.rq

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


Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us 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.

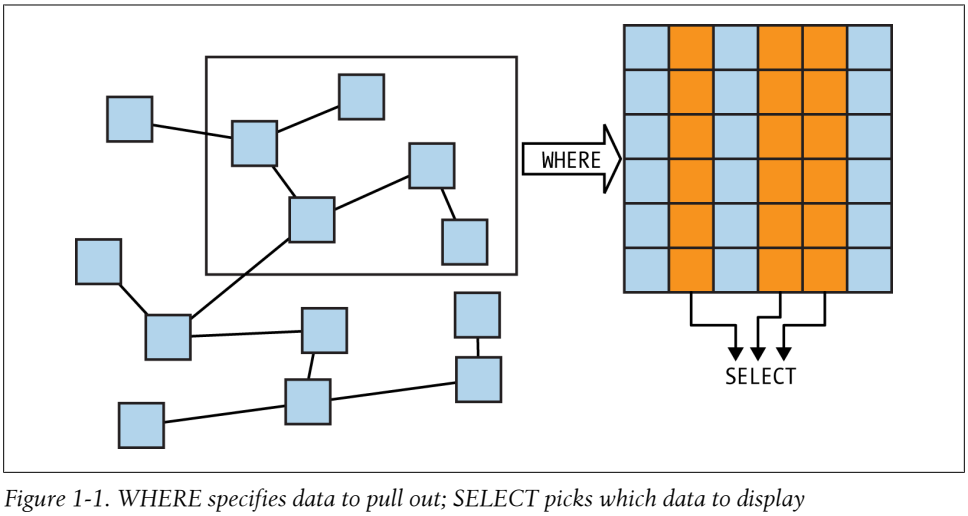



*Figure 1-1. WHERE specifies data to pull out; SELECT picks which data to display*

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:

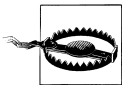
- You don't need to use prefixes in your query, but they can make the query more compact and easier to read than one that uses full URIs. When you do use a full URI, enclose it in angle brackets to show the processor that it's a URI.
- Whitespace doesn't affect SPARQL syntax. The new query has carriage 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:

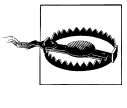
```
--------------
| person     |
==============
| ab:richard |
--------------
```

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.)

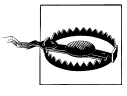
```
# filename: ex023.rq

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.


*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.

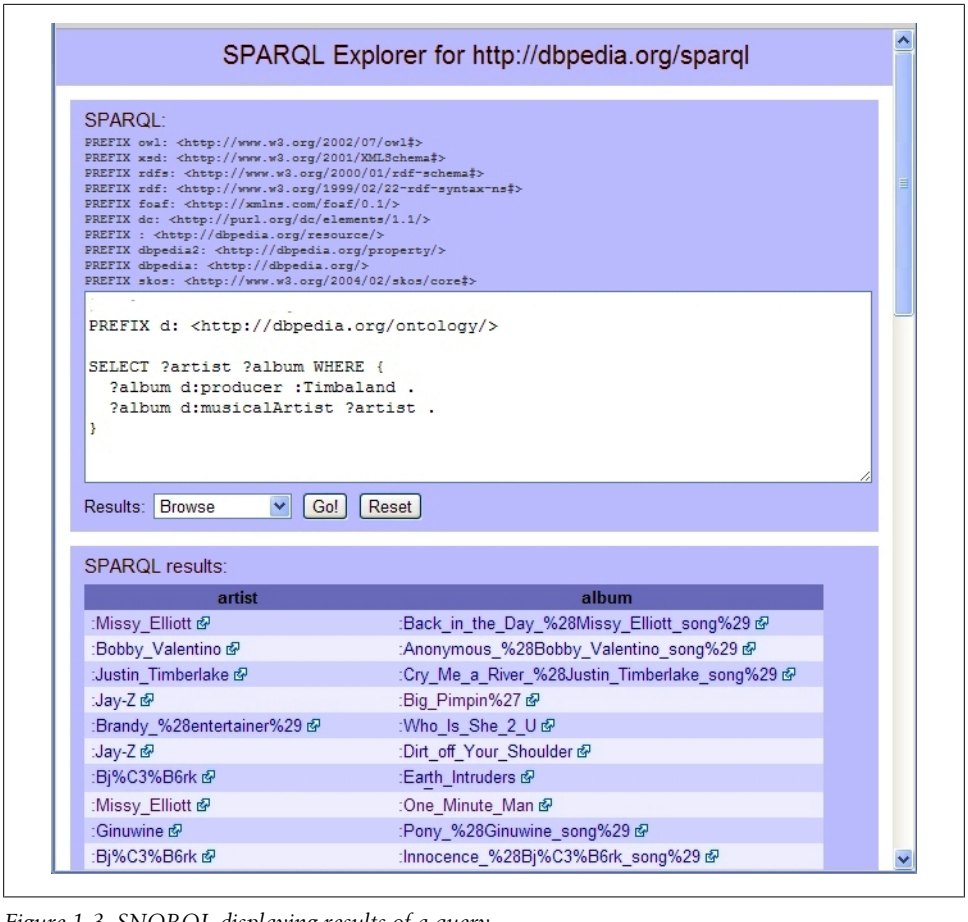



*Figure 1-3. SNORQL displaying results of a query*

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.