Chapter 6
SPARQL: Querying the Semantic Web

This chapter covers SPARQL, the last core component of the Semantic Web. With SPARQL, you will be able to locate specific information on the machine-readable Web, and the Web can therefore be viewed as a gigantic database, as many of us have been dreaming about.

This chapter will cover all the main aspects of SPARQL, including its concepts, its main language constructs and features, and certainly, real-world examples and related tools you can use when querying the Semantic Web. Once you are done with this chapter, you will have a complete tool collection that you can use to continue exploring the world of the Semantic Web.

6.1 SPARQL Overview

6.1.1 SPARQL in Official Language

SPARQL (pronounced "sparkle") is an RDF query language and data access protocol for the Semantic Web. Its name is a recursive acronym that stands for *SPARQL Protocol and RDF Query Language*. It was standardized by W3C's SPARQL Working Group (formerly known as the RDF Data Access Working Group) on 15 January 2008. You can follow its activities from the official Web site:

```
http://www.w3.org/2009/sparql/wiki/Main_Page
```

which also lists the specifications contained in the official W3C Recommendation.

The W3C Recommendation of SPARQL consists of three separate specifications. The first one *SPARQL Query Language specification*[1] makes up the core. Together with this language specification is the *SPARQL Query XML Results Format specification*[2] which describes an XML format for serializing the results of a SPARQL query (including both `SELECT` and `ASK` query). The third specification is

[1] http://www.w3.org/TR/rdf-sparql-query/

[2] http://www.w3.org/TR/rdf-sparql-XMLres/

DOI 10.1007/978-3-642-15970-1_6, © Springer-Verlag Berlin Heidelberg 2011

the *SPARQL Protocol for RDF specification*[3] that uses WSDL 2.0 to define simple HTTP and SOAP protocols for remotely querying RDF databases.

In total, therefore, SPARQL Recommendation consists of a query language, a XML format in which query results will be returned, and a protocol of submitting a query to a query processor service remotely.

The main focus of this chapter is on SPARQL query language itself; all of the key language constructs will be covered with ample examples. It will also cover the new features proposed by SPARQL 1.1, which have not been standardized at the time of this writing. Nevertheless, these new features will likely to remain stable and should be very useful for you to become even more productive with SPARQL.

6.1.2 SPARQL in Plain English

At this point, we have learned RDF, a model and data format that we can use to create structured content for machine to read. We have also learned RDF schema and OWL language, and we can use these languages to create ontologies.

With ontologies, anything we say in our RDF documents, we have a reason to say them. And more importantly, since the RDF documents we create all share these common ontologies, it becomes much easier for machines to make inferences based on these RDF contents, therefore generating even more RDF statements, as we have seen in the last chapter.

As a result, there will be more and more content being expressed in RDF format. And indeed, for last several years, a large amount of RDF documents have been published on the Internet and a machine-readable Web has started to take shape. Following all these is the need to locate specific information on this data Web.

A possible solution is to build a new kind of search engine that will work on this emerging Semantic Web. Since the underlying Web is machine readable, this new search engine will have a much better performance than that delivered by the search engines working under traditional Web environment.

However, this solution will not be able to take full advantage of the Semantic Web. More specifically, a search engine does not directly give us answers; instead, it returns to us a collection of pages that might contain the answer. Since we are working a machine-readable Web, why not directly ask for the answer?

Therefore, to push the solution one step further, we need a query language that we can use on this data Web. By simply submitting a query, we should be able to directly get the answer.

SPARQL query language is what we are looking for. In plain English,

> *SPARQL is a query language that we can use to query the RDF data content and SPARQL also provides a protocol that we need to follow if we want to query a remote RDF data set.*

The benefit of having a query language such as SPARQL is also obvious. To name a few,

[3]http://www.w3.org/TR/rdf-sparql-protocol/

- to query RDF graphs to get specific information;
- similarly, to query a remote RDF server and to get streaming results back;
- to run automated regular queries again RDF dataset to generate reports;
- to enable application development at a higher level, i.e., application can work with SPARQL query results, not directly with RDF statements.

6.1.3 Other Related Concepts: RDF Data Store, RDF Database, and Triple Store

RDF data store, RDF database, and triple store are three concepts you will hear a lot when you start to work with SPARQL. In fact, all these three phrases mean exactly the same thing and are interchangeable. To understand them, we only need to understand any one of them. Let us look at RDF data store in more detail.

To put it simple, an *RDF data store* is a special database system built for the storage and retrieval of RDF statements.

As we know, a relational database management system (DBMS) is built for general purpose. Since no one can predict what data model (tables, schemas, etc.) will be needed for a specific project, all the functionalities, such as adding, deleting, updating, and locating a record, have to be built as general as possible. To gain this generality, performance cannot be the top priority at all time.

If we devote a database system to store RDF statements *only*, we know what kind of data model will be stored already. More specifically, every record is a short statement in the form of subject–predicate–object. As a result, we can modify a DBMS so that it is optimized for the storage and retrieval of RDF statements only.

Therefore, an RDF data store is like a relational database in that we can store RDF statements there and retrieve them later by using a query language. However, an RDF data store is specially made and optimized for storing and retrieving RDF statement only.

An RDF data store can be built as a specialized database engine from scratch, or it can be built on top of existing commercial relational database engines. Table 6.1 shows some RDF data stores created by different parties using different languages.

Any given RDF data store, be it built from scratch or on top of an existing commercial data base system, should have the following features:

Table 6.1 Examples of RDF data store implementations

RDF data store name	implementation language	home page
4store	C	`http://www.4store.org`
ARC	C	`http://arc.semsol.org`
Joseki	Java	`http://www.joseki.org`
Redland	C	`http://librdf.org`
Sesame	Java	`http://www.openrdf.org`
Virtuoso	C	`http://virtuoso.openlinksw.com`

- a common storage medium for any application that manipulates RDF content;
- a set of APIs that allow applications to add triples to the store, query the store, and delete triples from the store.

There can be different add-on features provided by different implementations. For example, some RDF data stores will support loading an RDF document from a given URL, and some will support RDF schema, therefore offering some basic forms of inferencing capability. It is up to you to understand the features of the RDF data store that you have on hand.

In this chapter, we will be using an RDF data store called Joseki, and we will help you to set it up in the next section.

6.2 Set up Joseki SPARQL Endpoint

A *SPARQL endpoint* can be understood as an interface that users (human or application) can access to query an RDF data store by using SPARQL query language. Its function is to accept queries and return result accordingly. For human users, this endpoint could be a stand-alone or a Web-based application. For applications, this endpoint takes the form of a set of APIs that can be used by the calling agent.

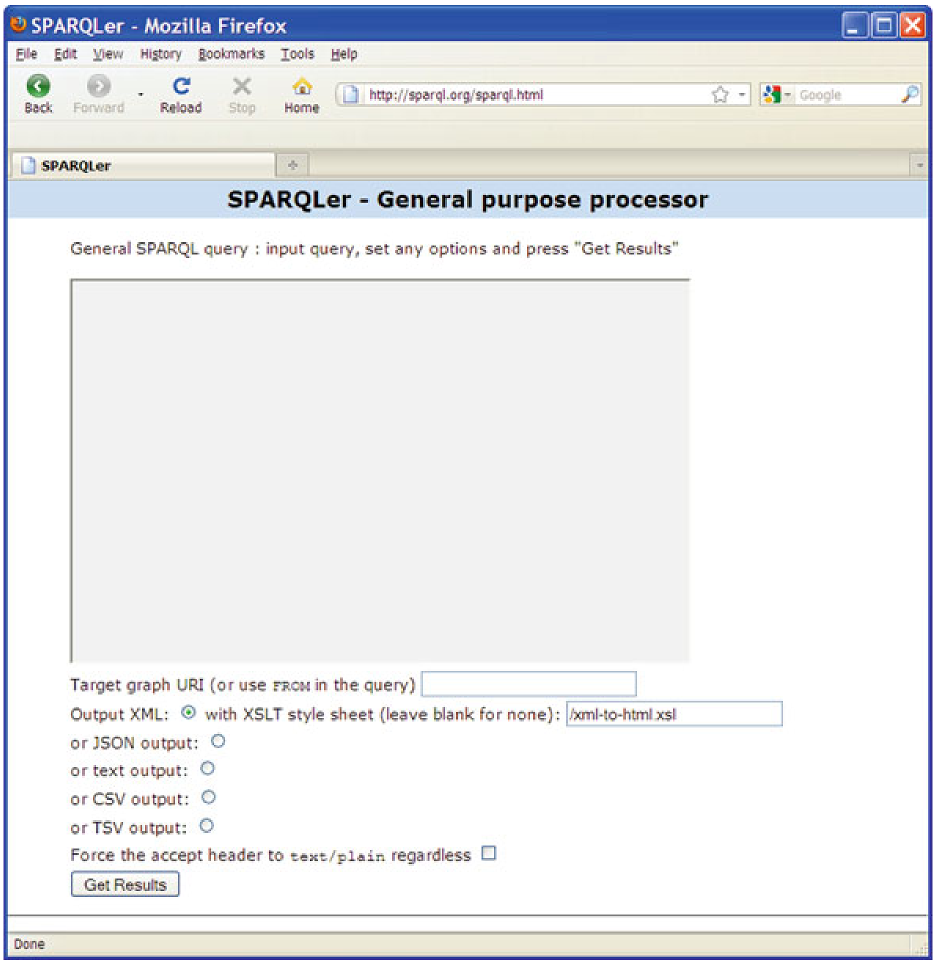
A SPARQL endpoint can be configured to return results in a number of different formats. For instance, when used by human users in an interactive way, it presents the result in the form of a HTML table, which is often constructed by applying XSL transforms to the XML result that is returned by the endpoint. When accessed by applications, the results are serialized into machine-processable formats, such as RDF/XML or Turtle format, just to name a few.

SPARQL endpoints can be categorized as *generic* endpoints and *specific* endpoints. A generic endpoint works against any RDF dataset, which could be stored locally or accessible from the Web. A specific endpoint is tied to one particular dataset, and this dataset cannot be switched to another endpoint.

In this chapter, our main goal is to learn SPARQL's language feature; we are going to use a SPARQL endpoint that works in a command line fashion so that we can test our queries right away. In later chapters, we will see how to query RDF statements by using programmable SPARQL endpoint in a soft agent.

There are quite a few SPARQL endpoints available and we have selected *Joseki* as our test bed throughout this chapter. Joseki comes with the Jena Semantic Web framework developed by HP, which is arguably the most popular tool suite for developing Semantic Web applications (more details about Jena in later chapters).

More specifically, Joseki is a Web-based SPARQL endpoint. It contains its own Web server, which hosts a Java servlet engine to support its SPARQL endpoint. It renders a Web query form so that we can enter our query manually, and by submitting the query form, we will be presented with the query result right away. Joseki endpoint also provides a set of SPARQL APIs that we can use to submit queries programmatically, as we will see in later chapters.


SPARQLer - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Back Forward Reload Stop Home
http://sparql.org/sparql.html
Google
SPARQLer
Get Results
Done

SPARQLer - General purpose processor

General SPARQL query : input query, set any options and press "Get Results"

Target graph URI (or use FROM in the query)

Output XML: [✓] with XSLT style sheet (leave blank for none): /xml-to-html.xsl

or JSON output:

or text output:

or CSV output:

or TSV output:

Force the accept header to text/plain regardless

Fig. 6.1 Joseki SPARQL endpoint

There are two ways to access Joseki. The easiest way is to access Joseki endpoint directly online:

```
http://sparql.org/sparql.html
```

and this endpoint is shown in Fig. 6.1.

You can use the simple text form shown in Fig. 6.1 to enter your query and click `Get Results` button to see the query result – we will see more details in the coming sections.

Another choice is to download Joseki package and install it on your machine so that you can test SPARQL query anytime you want, even when you are offline. You will be seeing exactly the same interface as shown in Fig. 6.1, except that it is hosted at this location:

```
http://localhost:2020/
```

We will now discuss how to setup Joseki on your local machine. If you decide to go with online endpoint, you can simply skip the following discussion.

To start, download Joseki from this location

```
http://www.joseki.org/download.html
```

and remember to get the latest version. Once you have downloaded it, you can install it on your machine, at any location you want. To start using it, you need to finish the following two simple setup steps.

The first step is to set the `JOSEKIROOT` environment variable so that it points to the location of your installation. For example, I have installed Joseki at this location:

```
C:\liyang\DevApp\Joseki-3.2
```

Therefore, I have added the following environment variable:

```
JOSEKIROOT=C:\liyang\DevApp\Joseki-3.2
```

The second step is to update your `CLASSPATH` environment variable and note that every jar file under lib directory (`C:\liyang\DevApp\Joseki-3.2\lib`) has to be added into the `CLASSPATH` variable. Again, use my installation as an example; part of my `CLASSPATH` variable should look like the following:

```
CLASSPATH=.;C:\liyang\DevApp\Joseki-3.2\lib\antlr-2.7.5.jar;
                  C:\liyang\DevApp\Joseki-3.2\lib\arq.jar; ......
```

where . . . . . . represents the other jar files.

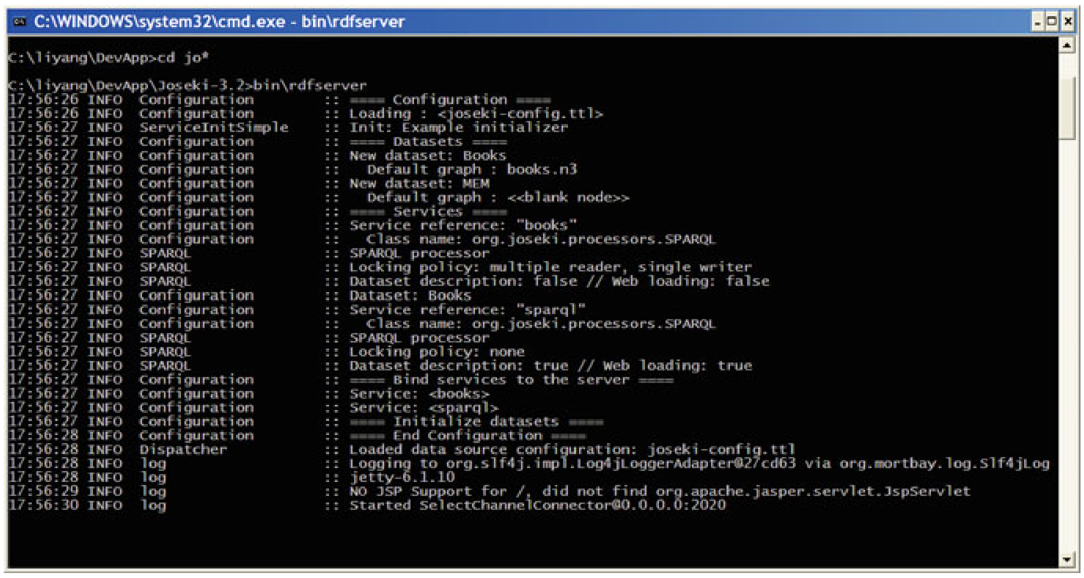
Now, you are ready to start Joseki SPARQL endpoint. Go to the following location (using my installation as example):

```
C:\liyang\DevApp\Joseki-3.2
```

type in the following command to start it:

```
bin\rdfserver
```

This should start Joseki server successfully. If everything works fine, you should see a window output that is similar to the one shown in Fig. 6.2.



```
C:\liyang\DevApp>cd jo*

C:\liyang\DevApp\Joseki-3.2>bin\rdfserver
17:56:26 INFO  Configuration        :: ==== Configuration ====
17:56:26 INFO  Configuration        :: Loading : <joseki-config.ttl>
17:56:27 INFO  ServiceInitSimple    :: Init: Example initializer
17:56:27 INFO  Configuration        :: ==== Datasets ====
17:56:27 INFO  Configuration        :: New dataset: Books
17:56:27 INFO  Configuration        ::   Default graph : books.n3
17:56:27 INFO  Configuration        :: New dataset: MEM
17:56:27 INFO  Configuration        ::   Default graph : <<blank node>>
17:56:27 INFO  Configuration        :: ==== Services ====
17:56:27 INFO  Configuration        :: Service reference: "books"
17:56:27 INFO  Configuration        ::   Class name: org.joseki.processors.SPARQL
17:56:27 INFO  SPARQL               :: SPARQL processor
17:56:27 INFO  SPARQL               :: Locking policy: multiple reader, single writer
17:56:27 INFO  SPARQL               :: Dataset description: false // Web loading: false
17:56:27 INFO  Configuration        :: Dataset: Books
17:56:27 INFO  Configuration        :: Service reference: "sparql"
17:56:27 INFO  Configuration        ::   Class name: org.joseki.processors.SPARQL
17:56:27 INFO  SPARQL               :: SPARQL processor
17:56:27 INFO  SPARQL               :: Locking policy: none
17:56:27 INFO  SPARQL               :: Dataset description: true // Web loading: true
17:56:27 INFO  Configuration        :: ==== Bind services to the server ====
17:56:27 INFO  Configuration        :: Service: <books>
17:56:27 INFO  Configuration        :: Service: <sparql>
17:56:27 INFO  Configuration        :: ==== Initialize datasets ====
17:56:28 INFO  Configuration        :: ==== End Configuration ====
17:56:28 INFO  Dispatcher           :: Loaded data source configuration: joseki-config.ttl
17:56:28 INFO  log                  :: Logging to org.slf4j.impl.Log4jLoggerAdapter@27cd63 via org.mortbay.log.Slf4jLog
17:56:28 INFO  log                  :: jetty-6.1.10
17:56:29 INFO  log                  :: NO JSP Support for /, did not find org.apache.jasper.servlet.JspServlet
17:56:30 INFO  log                  :: Started SelectChannelConnector@0.0.0.0:2020
```

Fig. 6.2 Check if you have installed Joseki correctly

Now open a browser and put the following into the address bar:

```
http://localhost:2020/sparql.html
```

You will see a SPARQL query form that looks exactly like the one shown in Fig. 6.1. For the rest of this chapter, this is what we are going to use for all our SPARQL queries.

6.3 SPARQL Query Language

Now, we are ready to study the query language itself. We need to select an RDF data file first and use it as our example throughout this chapter. This example data file has to be clear enough for us to understand, yet it cannot be too trivial to reflect the power of SPARQL query language.

Let us take Dan Brickley's FOAF document as our example. FOAF represents a project called *Friend Of A Friend*, and Dan Brickley is one of the two founders of this project. We have a whole chapter coming up to discuss FOAF in detail; for now, understanding the following about FOAF is good enough for you to continue:

- The goal of FOAF project is to build a social network using the Semantic Web technology so that we can do experiments with it and build applications that are not easily built under traditional Web.
- The core element of FOAF project is the FOAF ontology, a collection of terms that can be used to describe a person: name, home page, e-mail address, interest, and people he/she knows, etc.
- Anyone can create an RDF document to describe himself/herself by using this FOAF ontology, and he/she can join the friends network as well.

Dan Brickley's FOAF document is therefore a collection of RDF statements that he created to describe himself, and the terms he used to do so come from FOAF ontology. You can find his file at this URL:

```
http://danbri.org/foaf.rdf
```

And since the file is quite long, List 6.1 shows only part of it, so that you can get a feeling about how a FOAF document looks like. Note that Dan Brickley can change his FOAF document at any time, therefore at the time you are reading this book, the exact content of this RDF document could be different. However, the main idea is the same, and all the queries against this file will still be valid.

List 6.1 Part of Dan Brickley's FOAF document

```
<?xml version="1.0"?>

<rdf:RDF
       xml:lang="en"
       xmlns:wot="http://xmlns.com/wot/0.1/"
```

```
      xmlns:rdf=
            "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:dct="http://purl.org/dc/terms/"
      xmlns:lang=
         "http://purl.org/net/inkel/rdf/schemas/lang/1.1#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:rss="http://purl.org/rss/1.0/"
      xmlns="http://xmlns.com/foaf/0.1/"
      xmlns:foaf="http://xmlns.com/foaf/0.1/"
      xmlns:wn="http://xmlns.com/wordnet/1.6/"
      xmlns:air=
            "http://www.megginson.com/exp/ns/airports#"
      xmlns:contact=
            "http://www.w3.org/2000/10/swap/pim/contact#"
      xmlns:dc="http://purl.org/dc/elements/1.1/">

<Person rdf:ID="danbri">

  <foaf:name>Dan Brickley</foaf:name>
  <foaf:nick>danbri</foaf:nick>

  <mbox rdf:resource="mailto:danbri@danbri.org"/>
  <mbox rdf:resource="mailto:danbri@porklips.org"/>

  <plan>Save the world and home in time for tea.</plan>

  <knows>
    <Person>
     <mbox rdf:resource=
               "mailto:libby.miller@bristol.ac.uk"/>
     <mbox rdf:resource="mailto:libby@asemantics.com"/>
    </Person>
  </knows>

  <knows>
    <Person rdf:about=
       "http://www.w3.org/People/Berners-Lee/card#i">
      <name>Tim Berners-Lee</name>
      <isPrimaryTopicOf rdf:resource=
         "http://en.wikipedia.org/wiki/Tim_Berners-Lee"/>
      <homepage rdf:resource=
                "http://www.w3.org/People/Berners-Lee/"/>
      <mbox rdf:resource="mailto:timbl@w3.org"/>
      <rdfs:seeAlso rdf:resource=
            "http://www.w3.org/People/Berners-Lee/card"/>
```

```
42:      </Person>
43:    </knows>
44:
45: </Person>
46:
47:</rdf:RDF>
```

As you can see, he has included his name and nick name (line 20, 21), his e-mail addresses (line 23, 24), and his plan (line 26). He has also used `foaf:knows` to include some of his friends, as shown in line 28–33, line 35–43. Note that a default namespace is declared in line 11, and that default namespace is the FOAF ontology namespace (see next chapter for details). As a result, he can use terms from FOAF ontology without adding any prefix, such as `Person`, `knows`, `mbox`, `plan`.

6.3.1 The Big Picture

SPARQL provides four different forms of query:

- `SELECT` query
- `ASK` query
- `DESCRIBE` query
- `CONSTRUCT` query

Among these forms, `SELECT` query is the most frequently used query form. In addition, all these query forms are based on two basic SPARQL concepts: triple pattern and graph pattern. Let us understand these two concepts first before we start to look at SPARQL queries.

6.3.1.1 Triple Pattern

As we have learned, RDF model is built on the concept of triple, a three-tuple structure consisting of subject, predicate, and object. Likewise, SPARQL is built upon the concept of *triple pattern*, which is also written as subject, predicate, and object, and has to be terminated with a full stop. The difference between RDF triple and SPARQL triple pattern is that a SPARQL triple pattern can include variables: any or all of the subject, predicate, and object values in a triple pattern can be a variable. Clearly, an RDF triple is also a SPARQL triple pattern.

The second line in the following example is a SPARQL triple pattern (note that the Turtle syntax is used here):

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
<http://danbri.org/foaf.rdf#danbri> foaf:name ?name.
```

As you can tell, the subject of this triple pattern is Dan Brickley's URI, the predicate is `foaf:name`, and the object component of this triple pattern is a variable, identified by the `?` character in front of a string `name`.

Note that a SPARQL variable can be prefixed with either a `?` character or a `$` character, and these two are interchangeable. In this book, we will use the `?` character. In other words, a SPARQL variable is represented by the following format:

```
?variableName
```

where the `?` character is necessary, and `variableName` is given by the user.

The best way to understand a variable in a triple pattern is to view it as a placeholder that can match any value. More specifically, here is what happens when the above triple pattern is used against an RDF graph:

0. Create an empty RDF document, call it `resultSet`.
1. Get the next triple from the given RDF graph; if there is no more triple, return the `resultSet`.
2. Match the current triple with the triple pattern: if both the subject and the predicate of the current triple match the given subject and predicate in the triple pattern, the actual value of the object from the current triple will *bind* to the variable called `name`, therefore creates a new concrete triple that will be collected into the `resultSet`.
3. Go back to step 1.

Obviously, the above triple pattern can be read as follows:

find the value of `foaf:name` property defined for RDF resource identified by `http://danbri.org/foaf.rdf#danbri`.

And based on the above steps, it is clear that all possible bindings are included. Therefore, if we have multiple instances of `foaf:name` property defined for `http://danbri.org/foaf.rdf#danbri`, all these multiple bindings will be returned.

It is certainly fine to have more than one variable in a triple pattern. For example, the following triple pattern has two variables:

```
<http://danbri.org/foaf.rdf#danbri> ?property ?name.
```

And it means to find all the properties and their values that have been defined to the resource identified by `http://danbri.org/foaf.rdf#danbri`.

It is also fine to have all components as variables:

```
?subject ?property ?name.
```

This triple pattern will then match all triples in a given RDF graph.

6.3.1.2 Graph Pattern

Another important concept in SPARQL is called *graph pattern*. Similar to triple pattern, graph pattern is also used to select triples from a given RDF graph, but it can specify a much more complex "selection rule" compared to simple triple pattern.

First off, a collection of triple patterns is called a graph pattern. In SPARQL, `{` and `}` are used to specify a collection of triple patterns. For example, the following three triple patterns present one graph pattern:

```
{
   ?who foaf:name ?name.
   ?who foaf:interest ?interest.
   ?who foaf:knows ?others.
}
```

To understand how graph pattern is used to select resources from a given RDF graph, we need to remember one key point about the graph pattern: if a given variable shows up in multiple triple patterns within the graph pattern, its value in all these patterns has to be the same. In other words, each resource returned must be able to substitute into all occurrences of the variable. More specifically,

0. Create an empty set called `resultSet`.
1. Get the next resource from the given RDF graph. If there is no more resource left, return `resultSet` and stop.
2. Process the first triple pattern:
   - If the current resource does not have a property instance called `foaf:name`, go to 6.
   - Otherwise, bind the current resource to variable `?who` and bind the value of property `foaf:name` to variable `?name`.
3. Process the second triple pattern:
   - If the current resource (represented by variable `?who`) does not have a property instance called `foaf:interest`, go to 6.
   - Otherwise, bind the value of property `foaf:interest` to variable `?interest`.
4. Process the third triple pattern:
   - If the current resource (represented by variable `?who`) does not have a property instance called `foaf:knows`, go to 6.
   - Otherwise, bind the value of property `foaf:knows` to variable `?others`.
5. Collect the current resource into `resultSet`.
6. Go to 1.

Based on these steps, it is clear that this graph pattern in fact tries to find any resource that has all three of the desired properties defined. The above process will stop its inspection at any point and move on to a new resource if the current resource does not have any of the required property defined.

You should be able to understand other graph patterns in a similar way just by remembering this basic rule: within a graph pattern, a variable must always be bound to the same value no matter where it shows up.

And now, we are ready to dive into the world of SPARQL query language.

6.3.2 SELECT Query

The SELECT query form is used to construct standard queries, and it is probably the most popular form among the four. In addition, most of its features are shared by other query forms.

6.3.2.1 Structure of a SELECT Query

List 6.2 shows the structure of a SPARQL SELECT query:

List 6.2 The structure of a SPARQL SELECT query

```
# base directive
BASE <URI>

# list of prefixes
PREFIX pref: <URI>
...

# result description
SELECT...

# graph to search
FROM ...

# query pattern
WHERE {
   ...
}

# query modifiers
ORDER BY...
```

As shown in List 6.2, a SELECT query starts with a BASE directive and a list of PREFIX definitions which may contain an arbitrary number of PREFIX statements. These two parts are optional and they are used for URI abbreviations. For example, if you assign a label pref to a given URI, then the label can be used anywhere in a query in place of the URI itself. Also note that pref is simply a label, we can name it anyway we want. This is all quite similar to Turtle language abbreviation we have discussed in Chap. 2, and we will see more details about these two parts in the upcoming query examples.

The SELECT clause comes next. It specifies which variable bindings, or data items, should be returned from the query. As a result, it “picks up” what information to return from the query result.

The FROM clause tells the SPARQL endpoint against which graph the search should be conducted. As you will see later, this is also an optional item – in some cases, there is no need to specify the dataset that is being queried against.

The WHERE clause contains the graph patterns that specify the desired results; it tells the SPARQL endpoint what to query for in the underlying data graph. Note that the WHERE clause is not optional, although the WHERE keyword itself is optional. However, for clarity and readability, it is a good idea not to omit WHERE.

The last part is generally called query modifiers. The main purpose is to tell the SPARQL endpoint how to organize the query results. For instance, ORDER BY clause and LIMIT clause are examples of query modifiers. Obviously, query modifiers are also optional.

6.3.2.2 Writing Basic SELECT Query

As we have discussed, our queries will be issued against Dan Brickley's FOAF document. And our first query will accomplish the following: since FOAF ontology has defined a group of properties that one can use to describe a person, it would be interesting to see which of these properties are actually used by Brickley. List 6.3 shows the query:

List 6.3 What FOAF properties did Dan Brickley use to describe himself?

```
1: base <http://danbri.org/foaf.rdf>
2: prefix foaf: <http://xmlns.com/foaf/0.1/>

3: select *
4: from <http://danbri.org/foaf.rdf>
5: where
6: {
7:   <#danbri> ?property ?value.
8: }
```

Since this is our first query, let us study it in greater detail. First off, note that SPARQL is not case sensitive, so all the keywords can be either in small letters or in capital letters.

Now, lines 1 and 2 are there for abbreviation purpose. Line 1 uses BASE keyword to define a base URI against which all relative URIs in the query will be resolved, including the URIs defined with PREFIX keyword. In this query, PREFIX keyword specifies that foaf will be the shortcut for an absolute URI (line 2), so the URI foaf stands for does not have to be resolved against the BASE URI.

Line 3 specifies which data items should be returned by the query. Note that only variables in the graph pattern (line 6–8) can be chosen as returned data items. In this example, we would like to return both the property names and their values, so we should have written the SELECT clause like this:

```
select ?property ?value
```

Since ?property and ?value are the only two variables in the graph pattern, we do not have to specify them one by one as shown above, we can simply use a $*$ as a wildcard for all the variables, as shown in line 3.

Line 4 specifies the data graph against which we are doing our search. Note that Joseki can be used as either a generic or a specific SPARQL endpoint, and in this chapter, we will always explicitly specify the location of Brickley's FOAF document.

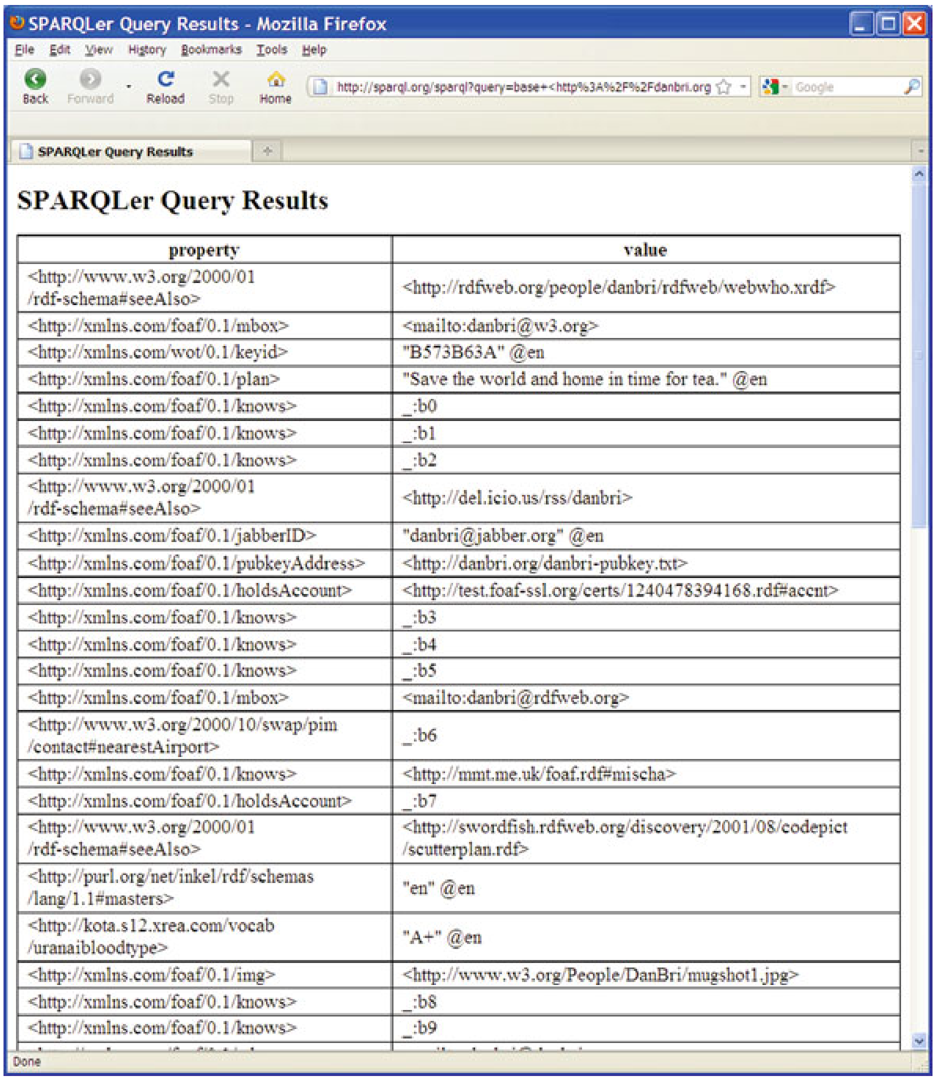
Line 5 is the `where` keyword, indicating that the search criteria will be the next, and lines 6–8 give the criteria represented by a graph pattern. Since we have discussed the concepts of triple pattern and graph pattern already, we understand how they are used to select the qualified triples. For this particular query, the graph pattern should be quite easy to follow. More specifically, this graph pattern has only one triple pattern, and it tries to match all the property instances and their values that are ever defined for the resource representing Brickley in real life.

Note that the resource representing Brickley has a relative URI as shown in line 7, and it is resolved by concatenating the `BASE` URI with this relative URI. The resolved URI is given as

```
http://danbri.org/foaf.rdf#danbri
```

which is the one that Brickley has used in his FOAF file.

Now you can put List 6.3 into the query box as shown in Fig. 6.1 and click `Get Result` button; you should be able to get the results back. Figure 6.3 shows part of the result.


SPARQLer Query Results - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Back Forward Reload Stop Home
http://sparql.org/sparql?query=base+<http%3A%2F%2Fdanbri.org
Google
SPARQLer Query Results
Done

SPARQLer Query Results

property	value
<http://www.w3.org/2000/01/rdf-schema#seeAlso>	<http://rdfweb.org/people/danbri/rdfweb/webwho.xrdf>
<http://xmlns.com/foaf/0.1/mbox>	<mailto:danbri@w3.org>
<http://xmlns.com/wot/0.1/keyid>	"B573B63A" @en
<http://xmlns.com/foaf/0.1/plan>	"Save the world and home in time for tea." @en
<http://xmlns.com/foaf/0.1/knows>	_:b0
<http://xmlns.com/foaf/0.1/knows>	_:b1
<http://xmlns.com/foaf/0.1/knows>	_:b2
<http://www.w3.org/2000/01/rdf-schema#seeAlso>	<http://del.icio.us/rss/danbri>
<http://xmlns.com/foaf/0.1/jabberID>	"danbri@jabber.org" @en
<http://xmlns.com/foaf/0.1/pubkeyAddress>	<http://danbri.org/danbri-pubkey.txt>
<http://xmlns.com/foaf/0.1/holdsAccount>	<http://test.foaf-ssl.org/certs/1240478394168.rdf#accnt>
<http://xmlns.com/foaf/0.1/knows>	_:b3
<http://xmlns.com/foaf/0.1/knows>	_:b4
<http://xmlns.com/foaf/0.1/knows>	_:b5
<http://xmlns.com/foaf/0.1/mbox>	<mailto:danbri@rdfweb.org>
<http://www.w3.org/2000/10/swap/pim/contact#nearestAirport>	_:b6
<http://xmlns.com/foaf/0.1/knows>	<http://mmt.me.uk/foaf.rdf#mischa>
<http://xmlns.com/foaf/0.1/holdsAccount>	_:b7
<http://www.w3.org/2000/01/rdf-schema#seeAlso>	<http://swordfish.rdfweb.org/discovery/2001/08/codepict/scutterplan.rdf>
<http://purl.org/net/inkel/rdf/schemas/lang/1.1#masters>	"en" @en
<http://kota.s12.xrea.com/vocab/uranaibloodtype>	"A+" @en
<http://xmlns.com/foaf/0.1/img>	<http://www.w3.org/People/DanBri/mugshot1.jpg>
<http://xmlns.com/foaf/0.1/knows>	_:b8
<http://xmlns.com/foaf/0.1/knows>	_:b9

Fig. 6.3 Part of the query result when running the query shown in List 6.3

From the result, we can see which properties have been used. For instance, `foaf:knows` and `foaf:mbox` are the most commonly used ones. Other properties such as `foaf:name, foaf:nick, foaf:homepage, foaf:holdsAccount` are also used.

Note that Brickley's FOAF file could be under constant updation, so at the time you are trying this query, you might not see the same result as shown in Fig. 6.3. Also, we will not continue to show the query results from now on unless it is necessary to do so, so this chapter will not be too long.

Let us try another simple query: find all the people known by Brickley. List 6.4 shows the query:

List 6.4 Find all the people known by Dan Brickley

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select *
from <http://danbri.org/foaf.rdf>
where
{
   <#danbri> foaf:knows ?friend.
}
```

Try to run the query, and right away you will see the problem: the query itself is right, but the result does not really tell us anything: there are way too many blank nodes.

In fact, it is quite often that when we include a friend we know in our FOAF documents, instead of using his/her URI, we simply use a blank node to represent this friend. For example, in List 6.1, one friend, Libby Miller, is represented by a blank node. As we will see in Chap. 7, even a blank node is used to represent a friend; as long as `foaf:mbox` property value is also included for this resource, any application will be able to recognize the resource.

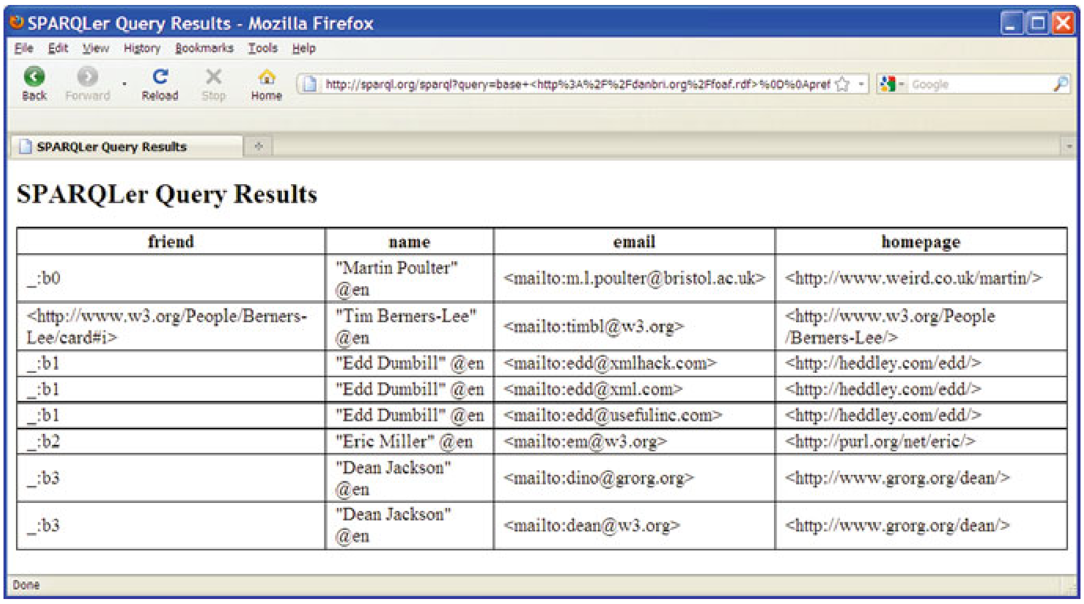
Let us change List 6.4 to accomplish the following: find all the people known by Brickley and show their name, e-mail address, and home page information. List 6.5 is the query that can replace the one in List 6.4:

List 6.5 Find all the people known by Brickley, show their names, e-mail addresses, and home page information

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select *
from <http://danbri.org/foaf.rdf>
where
{
   <#danbri> foaf:knows ?friend.
   ?friend foaf:name ?name.
```

```
    ?friend foaf:mbox ?email.
    ?friend foaf:homepage ?homepage.
}
```


SPARQLer Query Results - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Back Forward Reload Stop Home
http://sparql.org/sparql?query=base+<http%3A%2F%2Fdanbri.org%2Ffoaf.rdf>%0D%0Apref
Google
SPARQLer Query Results
Done

SPARQLer Query Results

friend	name	email	homepage
_:b0	"Martin Poulter" @en	<mailto:m.l.poulter@bristol.ac.uk>	<http://www.weird.co.uk/martin/>
<http://www.w3.org/People/Berners-Lee/card#i>	"Tim Berners-Lee" @en	<mailto:timbl@w3.org>	<http://www.w3.org/People /Berners-Lee/>
_:b1	"Edd Dumbill" @en	<mailto:edd@xmlhack.com>	<http://heddley.com/edd/>
_:b1	"Edd Dumbill" @en	<mailto:edd@xml.com>	<http://heddley.com/edd/>
_:b1	"Edd Dumbill" @en	<mailto:edd@usefulinc.com>	<http://heddley.com/edd/>
_:b2	"Eric Miller" @en	<mailto:em@w3.org>	<http://purl.org/net/eric/>
_:b3	"Dean Jackson" @en	<mailto:dino@grorg.org>	<http://www.grorg.org/dean/>
_:b3	"Dean Jackson" @en	<mailto:dean@w3.org>	<http://www.grorg.org/dean/>

Fig. 6.4 Results from running the query shown in List 6.5

The graph pattern in List 6.5 contains four triple patterns. Also, variable `?friend` is used as the object in the first triple pattern, but used as subject of the other three triple patterns. This is the so-called object-to-subject transfer in SPARQL queries. By doing this transfer, we can traverse multiple links in the RDF graph.

If we run this query against Brickley's FOAF graph, we do see the names, e-mails, and home pages of Brickley's friends, which make the result much more readable, as shown in Fig. 6.4.

However, the number of friends showing up in this result is much less than the number indicated by the result from running the query in List 6.4 – looks like some friends are missing. What is wrong? We will leave the answer to the next section, and before that, let us try some more SPARQL queries.

Some of Brickley's friends do have their pictures posted on the Web, and let us say for some reason we are interested in the formats of these pictures. The query shown in List 6.6 tries to find all the picture formats that have been used by Brickley's friends:

List 6.6 Find all the picture formats used by Brickley's friends

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix dc: <http://purl.org/dc/elements/1.1/>

select *
from <http://danbri.org/foaf.rdf>
```

```
where
{
   <#danbri> foaf:knows ?friend.
   ?friend foaf:depiction ?picture.
   ?picture dc:format ?imageFormat.
}
```

We have seen and discussed the object-to-subject transfer in List 6.5. In List 6.6, this transfer happens at a deeper level. The graph pattern in List 6.6 tries to match Brickley's friend, who has a `foaf:depiction` property instance defined, and this instance further has a `dc:format` property instance created, and we will like to return the value of this property instance. This chain of reference is the key to write queries using SPARQL, and is also frequently used.

In order to construct the necessary chain of references when writing SPARQL queries, we need to understand the structure of the ontologies that the given RDF graph file has used. Sometimes, in order to confirm our understanding, we need to read the graph which we are querying against. This should not surprise you at all; if you have to write SQL queries against some database tables, the first thing is to understand the structures of these tables, including the relations between them. The table structures and the relations between these tables can in fact be viewed as the underlying ontologies for these tables.

6.3.2.3 Using `OPTIONAL` Keyword for Matches

`Optional` keyword is another frequently used SPARQL feature, and the reason why `optional` keyword is needed is largely due to the fact that RDF data graph is only a semi-structured data model. For example, two instances of the same class type in a given RDF graph may have different set of property instances created for each one of them.

Let us take a look at Brickley's FOAF document, which has defined a number of `foaf:Person` instances. For example, one instance is created to represent Brickley himself, and quite a few others are defined to represent people he knows. Some of these `foaf:Person` instances do not have `foaf:name` property defined, and similarly, not every instance has `foaf:homepage` property instance created either.

This is perfectly legal, since there is no `owl:minCardinality` constraint defined on class `foaf:Person` regarding any of the above properties. For instance, not everyone has a home page; it is therefore not reasonable to require each `foaf:Person` instance to have a `foaf:homepage` property value. Also, recall that `foaf:mbox` is a inverse functional property, which is in fact used to uniquely identify a given person. As a result, having a `foaf:name` value or not for a given `foaf:Person` instance is not vital either.

This has answered the question we had in the previous section from List 6.5: not every Brickley's friend has a name, e-mail, and home page defined. And since the query in List 6.5 works like a logical AND, it only matches a friend whom Brickley knows and has *all* these three properties defined. Obviously, this will return less number of people compared to the result returned by the query in List 6.4.

Now, we can change this query in List 6.5 a little bit: find all the people known by Brickley and show their name, e-mail, and home page if *any* of that information is available.

To accomplish this, we need `optional` keyword, as shown in List 6.7:

List 6.7 Change List 6.5 to use `optional` keyword

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select *
from <http://danbri.org/foaf.rdf>
where
{
   <#danbri> foaf:knows ?friend.
   optional { ?friend foaf:name ?name. }
   optional { ?friend foaf:mbox ?email. }
   optional { ?friend foaf:homepage ?homepage. }
}
```

This query says, find all the people known by Brickley and show their name, e-mail, and home page information if *any* of this information is available. Run this query, you will see the difference between this query and the one shown in List 6.5. And here is the rule about `optional` keyword: the search will try to match all the graph patterns but does not fail the whole query if the graph pattern modified by `optional` keyword fails.

Note that in List 6.7, there are three different graph patterns modified by `optional` keyword, and any number of these graph patterns can fail, yet without causing the solution to be dropped. Clearly, if a query has multiple `optional` blocks, these `optional` blocks act independently of one another, any one of them can be omitted from or present in a solution.

Also note that the graph pattern modified by an `optional` keyword can have any number of triple patterns inside it. In List 6.7, each graph pattern modified by `optional` keyword happens to contain only one triple pattern. If a graph pattern modified by `optional` keyword contains multiple triple patterns, every single triple pattern in this graph pattern has to be matched in order to include a solution in the result set. For example, consider the query in List 6.8:

List 6.8 Use `optional` keyword on the whole graph pattern (compared with List 6.7)

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select *
from <http://danbri.org/foaf.rdf>
```

```
where
{
   <#danbri> foaf:knows ?friend.
   optional {
              ?friend foaf:name ?name.
              ?friend foaf:mbox ?email.
              ?friend foaf:homepage ?homepage.
            }
}
```

and compare the result from the one returned by List 6.7, you will see the difference. List 6.8 says, find all the people known by Brickley, show their name, e-mail, and home page information if *all* these information are available.

Let us look at one more example before we move on, which will be used in our later chapters: find all the people known by Brickley, for anyone of them, if she/he has e-mail address, show it, and if she/he has `rdfs:seeAlso` value, also get this value. This query is shown in List 6.9, and I will leave it for you to understand:

List 6.9 Find Dan Brickley's friends, who could also have e-mail addresses and `rdfs:seeAlso` defined

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

select *
from <http://danbri.org/foaf.rdf>
where
{
   <#danbri> foaf:knows ?friend.
   optional { ?friend foaf:mbox ?email. }
   optional { ?friend rdfs:seeAlso ?ref. }
}
```

6.3.2.4 Using Solution Modifier

At this point, we know SPARQL query is about matching patterns. More specifically, SPARQL engine tries to match the triples contained in the graph patterns against the RDF graph, which is a collection of triples. Once a match is successful, it will bind the graph pattern's variables to the graph's nodes, and one such variable binding is called a *query solution*. Since the `select` clause has specified a list of variables (or all the variables, as shown in our examples so far), the values of these listed variables will be selected from the current query solution to form a row that will be included in the final query result, and this row is called a *solution*. Obviously, another successful match will add another new solution, so on and so forth, therefore creating a table as the final result, with each solution presented as a row in this table.

Sometimes, it is better or even necessary for the solutions in the result table to be reorganized according to our need. For this reason, SPARQL has provided several *solution modifiers*, which will be the topic of this section.

The first one to look at is the `distinct` modifier, which eliminates duplicate solutions from the result table. Recall the query presented in List 6.3, which tries to find all the properties and their values that Brickley has used to describe himself. Now, let us change the query so that only the property names are returned, as shown in List 6.10:

List 6.10 Change List 6.3 to return only one variable back

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?property
from <http://danbri.org/foaf.rdf>
where
{
  <#danbri> ?property ?value.
}
```

run the query, and you can see a lot of repeated properties. Modify List 6.10 once more time to make it look as the query in List 6.11:

List 6.11 Use `distinct` keyword to eliminate repeated solutions from List 6.10

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select distinct ?property
from <http://danbri.org/foaf.rdf>
where
{
  <#danbri> ?property ?value.
}
```

and you will see the difference: all the duplicated solutions are now gone.

Another frequently used solution modifier is `order by`, which is also quite often used together with `asc()` or `desc()`. It orders the result set based on one of the variables listed in the `where` clause. For example, the query in List 6.12 tries to find all the people that have ever been mentioned in Brickley's FOAF file, and they are listed in a more readable way:

List 6.12 Use `order by` and `asc()` to modify results

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>
```

```
select ?name ?email
from <http://danbri.org/foaf.rdf>
where
{
  ?x a foaf:Person.
  ?x foaf:name ?name.
  ?x foaf:mbox ?email.
}
order by asc(?name)
```

Note that a pair of solution modifiers, namely `offset/limit`, is often used together with `order by` to take a defined slice from the solution set. More specifically, `limit` sets the maximum number of solutions to be returned, and `offset` sets the number of solutions to be skipped. These modifiers can certainly be used separately, for example, using `limit` alone will help us to ensure that not too many solutions are collected. Let us modify the query in List 6.12 to make it look like the one shown in List 6.13:

List 6.13 Use `limit/offset` to modify results

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?name ?email
from <http://danbri.org/foaf.rdf>
where
{
  ?x a foaf:Person.
  ?x foaf:name ?name.
  ?x foaf:mbox ?email.
}
order by asc(?name)
limit 10 offset 1
```

Run this query and compare with the result from List 6.12, you will see the result from `offset/limit` clearly.

6.3.2.5 Using `FILTER` Keyword to Add Value Constraints

If you have used SQL to query a database, you know it is quite straightforward to add value constraints in SQL. For example, if you are querying against a student database system, you may want to find all the students whose GPA is within a given range. In SPARQL, you can also add value constraints to filter the solutions in the result set, and the keyword to use is called `filter`.

More specifically, value constraints specified by `filter` keyword are logical expressions that evaluate to `boolean` values when applied on values of bound variables. Since these constraints are logical expressions, we can therefore combine them together by using logical `&&` and || operators. Only those solutions that are evaluated to be `true` by the given value constraints will be included in the final result set. Let us take a look at some examples.

List 6.14 will help us to accomplish the following: if Tim Berners-Lee is mentioned in Brickley's FOAF file, then we want to know what has been said about him:

List 6.14 What has been said about Berners-Lee?

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select distinct ?property ?propertyValue
from <http://danbri.org/foaf.rdf>
where
{
  ?person foaf:name "Tim Berners-Lee"@en.
  ?person ?property ?propertyValue.
}
```

When you run this query against Brickley's FOAF document, you will indeed see some information about Tim Berners-Lee, such as his home page and his e-mail address. However, note that this query does not use any `filter` keyword at all; instead, it directly adds the constraints to the triple pattern.

This is certainly fine, but with the major drawback that you have to specify the information with exact accuracy. For example, in List 6.14, if you replace the line

```
?person foaf:name "Tim Berners-Lee"@en.
```

with this line

```
?person foaf:name "tim Berners-Lee"@en.
```

the whole query will not work at all.

A better way to put constraints on the solution is to use `filter` keyword. List 6.15 shows how to do this:

List 6.15 Use `filter` keyword to add constraints to the solution

```
1:  base <http://danbri.org/foaf.rdf>
2:  prefix foaf: <http://xmlns.com/foaf/0.1/>

3:  select distinct ?property ?propertyValue
4:  from <http://danbri.org/foaf.rdf>
5:  where
6:  {
7:    ?timB foaf:name ?y.
8:    ?timB ?property ?propertyValue.
9:    filter regex(str(?y), "tim berners-Lee", "i").
10: }
```

Table 6.2 Functions and operators provided by SPARQL

Category	Functions and operators
Logical	`!, &&, ||`
Math	`+, -,*, /`
Comparison	`=, !=, >, <`
SPARQL testers	`isURL(), isBlank(), isLiteral(), bound()`
SPARQL accessors	`str(), lang(), datatype()`
Other	`sameTerm(), langMatches(), regex()`

Line 9 uses `filter` keyword to put constraints on the solution set. It uses a regular expression function called `regex()` to do the trick: for a given triple, its object component is taken and converted into a string by using the `str()` function, and if this string matches the given string, "`tim berners-Lee`," this `filter` will be evaluated to be `true`, in which case, the subject of the current triple will be bound to a variable called `?timB`. Note that "`i`" means ignore the case, so the string is matched even if it starts with a small `t`.

The rest of List 6.15 is easy: line 7 together with line 9 will bind variable `?timB` to the right resource, and line 8 will pick up all the properties and their related values that are ever used on this resource, which accomplishes our goal.

Run this query, and you will see it gives exactly the same result as given by List 6.14. However, it does not require us to know exactly how the desired resource is named in a given RDF file.

Note that `str()` is a function provided by SPARQL for us to use together with `filter` keyword. Table 6.2 summarizes the frequently used functions and operators; we will not go into the detailed description of each one of them, you can easily check them out.

Let us take a look at another example of using `filter` keyword: we want to find those who are known by Brickley and are also related to W3C (note that if someone has an e-mail address such as `someone@w3.org`, we will then assume he/she is related to W3C). List 6.16 is the query we can use:

List 6.16 Find Dan Brickley's friends who are related to W3C

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

select ?name ?email
from <http://danbri.org/foaf.rdf>
where
{
   <#danbri> foaf:knows ?friend.
   ?friend foaf:mbox ?email.
   filter regex(str(?email), "w3.org", "i" ).
   optional { ?friend foaf:name ?name. }
}
```

Note that this query does not put e-mail address as an optional item; in other words, if someone known by Brickley is indeed related to W3C, however without his/her `foaf:mbox` information presented in the FOAF document, this person will not be selected.

List 6.17 gives the last example of using filter. It tries to find all those defined in Brickley's FOAF file and whose birthday is after the start of 1970 and before the start of 1980. It shows another flavor of `filter` keyword and also how to do necessary data conversations for the correct comparison we need.

List 6.17 Find all the person whose birthday is within a given time frame

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>

select ?name ?dob
from <http://danbri.org/foaf.rdf>
where
{
   ?person a foaf:Person.
   ?person foaf:name ?name.
   ?person foaf:dateOfBirth ?dob.
   filter ( xsd:date(str(?dob)) >= "1970-01-01"^^xsd:date &&
            xsd:date(str(?dob)) < "1980-01-01"^^xsd:date )
}
```

6.3.2.6 Using `Union` Keyword for Alternative Match

Sometimes, a query needs to be expressed by multiple graph patterns that are mutually exclusive, and any solution will have to match exactly one of these patterns. This situation is defined as an *alternative match* situation, and SPARQL has provided `union` keyword for us to accomplish this.

A good example is from FOAF ontology, which provides two properties for e-mail address: `foaf:mbox` and `foaf:mbox_sha1sum`. The first one takes a readable plain text as its value, and the second one uses hash codes of an e-mail address as its value to further protect the owner's privacy. If we want to collect all the e-mail addresses that are included in Brickley's FOAF file, we have to accept either one of these two alternative forms, as shown in List 6.18:

List 6.18 Using `union` keyword to collect e-mail information

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?name ?mbox
from <http://danbri.org/foaf.rdf>
where
{
   ?person a foaf:Person.
   ?person foaf:name ?name.
   {
      { ?person foaf:mbox ?mbox. }
      union
      { ?person foaf:mbox_sha1sum ?mbox. }
   }
}
```

Now, any solution has to match one and exactly one of the two graph patterns that are connected by the `union` keyword. If someone has both a plain text e-mail address and a hash-coded address, then both of these addresses will be included in the solution set. This is also the difference between `union` keyword and `optional` keyword; as seen in List 6.19, since `optional` keyword is used, a given solution can be included in the result set without matching any of the two graph patterns at all:

List 6.19 Using `optional` keyword is different from using `union`, as shown in List 6.18

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?name ?mbox
from <http://danbri.org/foaf.rdf>
where
{
   ?person a foaf:Person.
   ?person foaf:name ?name.
   optional { ?person foaf:mbox ?mbox. }
   optional { ?person foaf:mbox_sha1sum ?mbox. }
}
```

After you run the query in List 6.19, you can find those solutions in the result set which do not have any e-mail address, and these solutions will for sure not be returned when the query in List 6.18 is used.

Another interesting change of List 6.18 is shown in List 6.20:

List 6.20 Another example using `union` keyword, different from List 6.18

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?name ?mbox ?mbox1

from <http://danbri.org/foaf.rdf>
where
{
   ?person a foaf:Person.
   ?person foaf:name ?name.
   {
      { ?person foaf:mbox ?mbox. }
      union
      { ?person foaf:mbox_sha1sum ?mbox1. }
   }
}
```

I will leave it to you to run the query in List 6.20 and to understand the query result.

Before we move on to the next section, let us take a look at one more example of `union` keyword, and this example will be useful in a later chapter.

As we know, almost any given Web page has links to other pages, and these links are arguably what makes the Web interesting. A FOAF file is also a Web page, with the only difference of being a page that machine can understand. Therefore, it should have links pointing to the outside world as well. The question is, what are these links in a given FOAF page (we will see how to use these links in a later chapter)?

At this point, at least the following three links can be identified:

- `rdfs:seeAlso`
- `owl:sameAs`
- `foaf:isPrimaryTopicOf`

Since all these properties can take us to either another document or another resource on the Web, they can be understood as links to the outside world.

The query shown in List 6.21 can help us to collect all these links from Brickley's FOAF document. If you can think of more properties that can be used as links, you can easily add them into List 6.21:

List 6.21 Using `union` keyword to collection links to the outside world

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>

select ?name ?seeAlso ?sameAs ?topicOf
from <http://danbri.org/foaf.rdf>
where
{
   ?person a foaf:Person.
   ?person foaf:name ?name.
   {
      { ?person rdfs:seeAlso ?seeAlso. }
      union
      { ?person owl:sameAs ?sameAs. }
      union
      { ?person foaf:isPrimaryTopicOf ?topicOf. }
   }
}
```

6.3.2.7 Working with Multiple Graphs

So far, all the queries we have seen have involved only one RDF graph, and we have specified it by using the `from` clause. This graph, in the world of SPARQL, is called a *background graph*. In fact, in addition to this background graph, SPARQL allows us to query any number of *named graphs*, and this is the topic of this section.

The first thing to know is how to make a named graph available to a query. As with the background graph, named graphs can be specified by using the following format:

```
from named <uri>
```

where <**uri**> specifies the location of the graph. Within the query itself, named graphs are used with the `graph` keyword, together with either a variable name that will bind to a named graph or the same <**uri**> for a named graph, as we will see in the examples. In addition, each named graph will have its own graph patterns to match against.

To show the examples, we need to have another RDF graph besides the FOAF file created by Brickley. For our testing purpose, we will use my own FOAF document as the second graph. You can see my FOAF file here:

```
http://www.liyangyu.com/foaf.rdf
```

Now, let us say that we would like to find those people who are mentioned by both Brickley and myself in our respective FOAF files. List 6.22 is our initial solution, which works with multiple graphs and uses `graph` keyword in conjunction with a variable called `graph_uri`:

List 6.22 Find those who are mentioned by both Brickley's and my own FOAF documents

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
select distinct ?graph_uri ?name ?email
from named <http://www.liyangyu.com/foaf.rdf>
from named <http://danbri.org/foaf.rdf>
where
{
   graph ?graph_uri
   {
      ?person a foaf:Person.
      ?person foaf:mbox ?email.
      optional { ?person foaf:name ?name. }
   }
}
```

Table 6.3 Partial result from query List 6.22

Graph_uri	Name	E-mail
. . .	. . .	. . .
<http://danbri.org/foaf.rdf>	"Libby Miller"@en	<mailto:libby.miller@bristol.ac.uk>
<http://danbri.org/foaf.rdf>	"Tim Berners-Lee"@en	<mailto:timbl@w3.org>
. . .	. . .	. . .
<http://www.liyangyu.com/foaf.rdf>		<mailto:libby.miller@bristol.ac.uk>

First of all, note that lines 3 and 4 specify the two named graphs by using `from named` keyword, and each graph is given by its own <**`uri`**>. As you can tell, one of the graphs is the FOAF file created by Brickley and the other one is my own FOAF document. Furthermore, lines 8–12 define a graph pattern, which will be applied to each of the named graphs available to this query.

When this query is executed by SPARQL engine, variable `graph_uri` will be bound to the URI of one of the named graphs, and the graph pattern shown from lines 8–12 will be matched against this named graph. Once this is done, variable `graph_uri` will be bound to the URI of the next name graph, and the graph pattern is again matched against this current named graph, so on and so forth, until all the named graphs are finished. If a match is found during this process, the matched person's `foaf:mbox` property value will be bound to `email` variable, and the `foaf:name` property value (if exists) will be bound to `name` variable. Finally, line 2 shows the result: it not only shows all the names and e-mails of the selected people but also shows from which file the information is collected.

To make our discussion easier to follow, Table 6.3 shows part of the result. At the time you are running this query, it is possible that you will see different results, but the discussion here will still apply.

Table 6.3 shows that both FOAF files have mentioned a person whose e-mail address is given by the following:

```
<mailto:libby.miller@bristol.ac.uk>
```

As we will see in Chap. 7, a person can be uniquely identified by her/his `foaf:mbox` property value, no matter whether we have assigned a `foaf:name` property value to this person or not. And to show this point, in my FOAF file, the name of the person identified by the above e-mail address is intentionally not provided.

Note that in order to find those people who are mentioned by both FOAF documents, we have to manually read the query result shown in Table 6.3, i.e., to find common e-mail addresses from both files. This is certainly not the best solution for us, and we need to change our query to *directly* find those who are mentioned in both graphs.

And this new query is shown in List 6.23:

List 6.23 Change List 6.22 to direct get the required result

```
1:  prefix foaf: <http://xmlns.com/foaf/0.1/>
2:  select distinct ?name ?email
3:  from named <http://www.liyangyu.com/foaf.rdf>
4:  from named <http://danbri.org/foaf.rdf>
5:  where
6:  {
7:     graph <http://www.liyangyu.com/foaf.rdf>
8:     {
9:        ?person a foaf:Person.
10:       ?person foaf:mbox ?email.
11:       optional { ?person foaf:name ?name. }
12:    }.
13:    graph <http://danbri.org/foaf.rdf>
14:    {
15:       ?person1 a foaf:Person.
16:       ?person1 foaf:mbox ?email.
17:       optional { ?person1 foaf:name ?name. }
18:    }.
19: }
```

In this query, the `graph` keyword is used with the URI of a named graph (line 7, 13). The graph pattern defined in lines 8–12 will be applied on my FOAF file, and if matches are found in this graph, they become part of a query solution; the value of `foaf:mbox` property is bound to a variable named `email`, and the value of `foaf:name` property (if exists) is bound to a variable called `name`. The second graph pattern (lines 14–18) will be matched against Brickley's FOAF graph, and the bound variables from the previous query solution will be tested here; the same variable `email` will have to be matched here, and if possible, the same `name` variable should match as well. Recall the key of graph patterns: any given variable, once bound to a value, has to bind to that value during the whole matching process.

Note that the variable representing a person is different in two graph patterns: in the first graph pattern, it is called `person` (line 9) and in the second graph pattern, it is called `person1` (line 15). The reason should be clear now; it does not matter whether this variable holds the same value or not in both patterns, since `email` value is used to uniquely identify a person resource. Also, it is possible that a given person resource is represented by blank nodes in both graphs, and blank nodes only have a scope that is within the graph that contains them. Therefore, even if they represent the same resource in the real world, it is simply impossible to match them at all.

Now run the query in List 6.23, we will be able to find all those people who are mentioned simultaneously in both graphs. Table 6.4 shows the query result.

As we have discussed earlier, part of the power of RDF graphs comes from data aggregation. Since both RDF files have provided some information about Libby

Table 6.4 result from query List 6.23

Name	E-mail
`"Libby Miller"@en`	`<mailto:libby.miller@bristol.ac.uk>` `<mailto:libby.miller@bristol.ac.uk>`

Miller, it will be interesting to aggregate these two pieces of information together and see what have been said about Miller as a `foaf:Person` instance. List 6.24 accomplishes this:

List 6.24 Data aggregation for Libby Miller

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
select distinct ?graph_uri ?property ?hasValue
from named <http://www.liyangyu.com/foaf.rdf>
from named <http://danbri.org/foaf.rdf>

where
{
   graph <http://www.liyangyu.com/foaf.rdf>
   {
      ?person1 a foaf:Person.
      ?person1 foaf:mbox ?email.
      optional { ?person1 foaf:name ?name. }
   }.

   graph <http://danbri.org/foaf.rdf>
   {
      ?person a foaf:Person.
      ?person foaf:mbox ?email.
      optional { ?person foaf:name ?name. }
   }.

   graph ?graph_uri
   {
      ?x a foaf:Person.
      ?x foaf:mbox ?email.
      ?x ?property ?hasValue.
   }

}
```

So far, we have seen examples where `graph` keyword is used together with a variable that will bind to a named graph (List 6.22), or it is used with an `<uri>` that represents a named graph (List 6.23). In fact, these two usage patterns can be mixed

Table 6.5 Result from query List 6.24

Graph_uri	Property	hasValue
<danbri:foaf.rdf>	<rdfs:seeAlso>	<http://www.libbymiller.com/webwho.xrdf>
<danbri:foaf.rdf>	<foaf:workplaceHomepage>	<http://ilrt.org/>
<danbri:foaf.rdf>	<foaf:mbox>	<mailto:libby.miller@bristol.ac.uk>
<danbri:foaf.rdf>	<foaf:name>	"Libby Miller"@en
<danbri:foaf.rdf>	<rdf:type>	<http://xmlns.com/foaf/0.1/Person>
<danbri:foaf.rdf>	<foaf:depiction>	<http://rdfweb.org/people/danbri/rdfweb/libby.gif>
<danbri:foaf.rdf>	<foaf:img>	<http://swordfish.rdfweb.org/~libby/libby.jpg>
<danbri:foaf.rdf>	<foaf:mbox>	<mailto:libby@asemantics.com>
<liyang:foaf.rdf>	<faof:homepage>	<http://www.ilrt.bris.ac.uk/~ecemm/>
<liyang:foaf.rdf>	<foaf:mbox>	<mailto:libby@asemantics.com>
<liyang:foaf.rdf>	<rdf:type>	<http://xmlns.com/foaf/0.1/Person>

liyang: http://www.liyangyu.com/
danbri: http://danbri.org/

together, as shown in List 6.24. After the discussion of Lists 6.22 and 6.23, List 6.24 is quite straightforward; clearly, lines 7–18 find those who have been included in both graphs, and lines 19–24 provide a graph pattern that will collect everything that has been said about those instances from all the named graphs.

Table 6.5 shows the query result generated by List 6.24. Again, at the time you are running the query, the result could be different. Nevertheless, as shown in Table 6.5, statements about Miller from both FOAF files have been aggregated together. This simple example in fact shows the basic flow of how an aggregation agent may work by using the search capabilities provided by SPARQL endpoints.

The last example of this section is given by List 6.25, where a background graph and a named graph are used together. Read this query and try to find what it does before you read on:

List 6.25 What this query does? Think about it before reading on

```
1:  prefix foaf: <http://xmlns.com/foaf/0.1/>

2:  select distinct ?property ?hasValue
3:  from <http://danbri.org/foaf.rdf>
4:  from named <http://www.liyangyu.com/foaf.rdf>
5:  from named <http://danbri.org/foaf.rdf>

6:  where
7:  {
8:     graph <http://www.liyangyu.com/foaf.rdf>
9:     {
10:       ?person1 a foaf:Person.
11:        ?person1 foaf:mbox ?email.
12:        optional { ?person1 foaf:name ?name. }
13:     }.

14:     graph <http://danbri.org/foaf.rdf>
15:     {
16:        ?person a foaf:Person.
17:        ?person foaf:mbox ?email.
18:        optional { ?person foaf:name ?name. }
19:     }.
20:     ?x a foaf:Person.
21:     ?x foaf:mbox ?email.
22:     ?x ?property ?hasValue.
23: }
```

The interesting part of this query is at lines 3–5, where a background graph is specified in line 3, and two named graphs are introduced in lines 4 and 5. Note that lines 3 and 5 are in fact the same graph.

Now, lines 8–19 are the same as the query in List 6.24 (trying to find those who are mentioned in both graphs), and lines 20–22 is a graph pattern that does not specify any graph, so this pattern is matched against the background graph. Therefore, this query, after finding those who are the mentioned in both Brickley's file and my file, tries to collect everything that has been said about them from Brickley's file *only*. This is not a data aggregation case as shown by List 6.24, but it shows the combination usage of a background graph and a named graph.

At this point, we have covered quite some features about SAPRQL's `select` query, and the examples presented here should have given you enough to explore their other features on your own. Let us move on to SPARQL's other query styles, and we will briefly discuss them in the next several sections.

6.3.3 CONSTRUCT Query

`construct` query is another query form provided by SPARQL which, instead of returning a collection of query solutions, returns a new RDF graph. Let us take a look at some examples.

List 6.26 creates a new FOAF graph, which has a collection of all the names and e-mails of those who are mentioned in Brickley's FOAF document. List 6.27 shows part of this new graph:

List 6.26 Example of a `construct` query

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

construct {
   ?person a foaf:Person.
   ?person foaf:name ?name.
   ?person foaf:mbox ?email.
}
from <http://danbri.org/foaf.rdf>
where
{
   ?person a foaf:Person.
   ?person foaf:name ?name.
   ?person foaf:mbox ?email.
}
```

List 6.27 Part of the generated RDF graph

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <foaf:Person>
    <foaf:mbox rdf:resource="mailto:craig@coolstuffhere.co.uk"/>
    <foaf:name xml:lang="en">Craig Dibble</foaf:name>
  </foaf:Person>
  <foaf:Person>
    <foaf:name xml:lang="en">Joe Brickley</foaf:name>
    <foaf:mbox rdf:resource=
               "mailto:joe.brickley@btopenworld.com"/>
  </foaf:Person>
  <foaf:Person>
    <foaf:mbox rdf:resource="mailto:libby.miller@bristol.ac.uk"/>
    <foaf:name xml:lang="en">Libby Miller</foaf:name>
  </foaf:Person>
... more ...
```

This generated new graph is indeed clean and nice, but it is not that much interesting. In fact, a common use of `construct` query form is to transform a given graph to a new graph that uses a different ontology.

For example, List 6.28 will transfer FOAF data to `vCard` data, and List 6.29 shows part of the resulting graph:

List 6.28 Another `construct` query which changes FOAF document into `vCard` document

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

construct {
   ?person vCard:FN ?name.
   ?person vCard:URL ?homepage.
}
```

```
from <http://danbri.org/foaf.rdf>

where
{
   optional {
     ?person foaf:name ?name.
     filter isLiteral(?name).
   }
   optional {
     ?person foaf:homepage ?homepage.
     filter isURI(?homepage).
   }
}
```

List 6.29 Part of the new graph expressed as vCard data

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:vCard="http://www.w3.org/2001/vcard-rdf/3.0#">
<rdf:Description>
    <vCard:FN xml:lang="en">Dan Connolly</vCard:FN>
</rdf:Description>
<rdf:Description>
    <vCard:FN xml:lang="en">Dan Brickley</vCard:FN>
</rdf:Description>
<rdf:Description>
    <vCard:FN xml:lang="en">Jim Ley</vCard:FN>
</rdf:Description>
<rdf:Description>
    <vCard:FN xml:lang="en">Eric Miller</vCard:FN>
    <vCard:URL rdf:resource="http://purl.org/net/eric/"/>
</rdf:Description>

... more ...
```

6.3.4 `DESCRIBE` *Query*

At this point, every query we have constructed requires us to know something about the data graph. For example, when we are querying against a FOAF document, at least we know some frequently used FOAF terms, such as `foaf:mbox` and `foaf:name`, so we can provide a search criteria to SPARQL query processor. This is similar to writing SQL queries against a database system; we will have to be familiar with the structures of the tables in order to come up with queries.

However, sometimes, we just don't know much about the data graph, and we don't even know what to ask. If this is the case, we can ask a SPARQL query processor to describe the resource we want to know, and it is up to the processor to provide some useful information about the resource we have asked.

And this is the reason behind the `describe` query. After receiving the query, a SPARQL processor will create and return an RDF graph; the content of the graph is decided by the query processor, not the query itself.

For example, List 6.30 is one such query:

List 6.30 Example of `describe` query

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
describe ?x
from <http://danbri.org/foaf.rdf>
where
{
   ?x foaf:mbox <mailto:timbl@w3.org>.
}
```

In this case, the only thing we know is the e-mail address, so we provide this information and ask SPARQL processor to tell us more about the resource whose e-mail address is given by `<mailto:timbl@w3.org>`. The query result is another RDF graph whose statements are determined by the query processor.

At the time of this writing, SPARQL Working Group has adopted `describe` keyword without reaching consensus. A description will be determined by the particular SPARQL implementation, and the statements included in the description are left to the nature of the information in the data source. For example, if you are looking for a description about a book resource, the author information could be included in the result RDF graph.

For this reason, we are not going to cover any more details. However, understanding the reason why this keyword is proposed will certainly help you in a later time when hopefully some agreement can be reached regarding the semantics of this keyword.

6.3.5 `ASK` Query

SPARQL's **`ask`** query is identified by `ask` keyword, and the query processor simply returns a `true` or `false` value, depending on whether the given graph pattern has any matches in the dataset or not.

List 6.31 is a simple example of `ask` query:

List 6.31 Example of using `ask` query

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
ask
from <http://danbri.org/foaf.rdf>
```

```
where
{
   ?x foaf:mbox <mailto:danbri@danbri.org>.
}
```

This query has a graph pattern that is equivalent to the following question: is there a resource whose `foaf:mbox` property uses `<mailto:danbri@danbri.org>` as its value? To answer this query, the processor tries to match the graph pattern against the FOAF data graph, and apparently, a successful match is found, therefore, `true` is returned as the answer.

It is fun to work with `ask` query. For example, List 6.32 tries to decide whether it is true that Brickley was either born before 1 January 1970 or after 1 January 1980:

List 6.32 Ask the birthday of Dan Brickley

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>

ask
from <http://danbri.org/foaf.rdf>
where
{
   <#danbri> foaf:dateOfBirth ?dob.
   filter ( xsd:date(str(?dob)) <= "1970-01-01"^^xsd:date ||
            xsd:date(str(?dob)) >= "1980-01-01"^^xsd:date )
}
```

And certainly, this will give `false` as the answer. Note that we should understand this answer in the following way: the processor cannot find any binding to compute a solution to the graph pattern specified in this query.

Obviously, a `true` or `false` answer depends on the given graph pattern. In fact, you can use any graph pattern together with `ask` query. As the last example, you can ask whether both Brickley's FOAF file and my own FOAF file have described anyone in common, and this query should look very familiar, as shown in List 6.33:

List 6.33 Ask if the two FOAF documents have described anyone in common

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

ask
from named <http://www.liyangyu.com/foaf.rdf>
from named <http://danbri.org/foaf.rdf>
where
{
    graph <http://www.liyangyu.com/foaf.rdf>
    {
       ?person a foaf:Person.
```

```
        ?person foaf:mbox ?email.
        optional { ?person foaf:name ?name. }
     }.
     graph <http://danbri.org/foaf.rdf>
     {
        ?person1 a foaf:Person.
        ?person1 foaf:mbox ?email.
        optional { ?person1 foaf:name ?name. }
     }.
}
```

And if you run the query, you can get `true` as the answer, as you have expected.

6.4 What Is Missing from SPARQL?

At this point, we have covered the core components of the current SPARQL standard. If you are experienced with SQL queries, you have probably realized the fact that there is something missing in the current SPARQL language constructs. Let us briefly discuss these issues in this section, and in the next section, we will take a closer look at SPARQL 1.1, SPARQL Working Group's latest progress.

The most obvious fact about SPARQL is that it is read-only. In its current stage, SPARQL is only a retrieval query language; there are no equivalents of the SQL `insert`, `update` and `delete` statements.

The second noticeable missing piece is that SPARQL does not support any grouping capabilities or aggregate functions, such as `min`, `max`, `avg`, `sum`, just to name a few. There are some implementations of SPARQL that provide these functions; yet, standardization is needed, so a uniform interface can be reached.

Another important missing component is the service description. More specifically, there is no standard way for a SPARQL endpoint to advertise its capabilities and its dataset.

There are other functionalities that are missing, and we are not going to list them all there. The good news is, some of these missing features have long been on the task list of W3C SPARQL Working Group, and a potentially updated standard called SPARQL 1.1 will be ready soon.

6.5 SPARQL 1.1

6.5.1 Introduction: What Is New?

SPARQL 1.1 is the collective name of the work produced by the current SPARQL Working Group. The actual components being worked on include the following major pieces:

- SPARQL 1.1 Query
- SPARQL 1.1 Update
- SPARQL 1.1 Protocol
- SPARQL 1.1 Service Description
- SPARQL 1.1 Uniform HTTP Protocol for Managing RDF Graphs
- SPARQL 1.1 Entailment Regimes
- SPARQL 1.1 Property Paths

and you can also find more details here:

`http://www.w3.org/2009/sparql/wiki/Main_Page`

At the time of this writing, these standards are still under active discussion and revision. To give you some basic idea of these new standards, we will concentrate on SPARQL 1.1 Query and Update. Not only because these two pieces are relatively stable but also because they are the ones that are most relevant to our day-to-day development work on the Semantic Web.

6.5.2 SPARQL 1.1 Query

In this section, the following new features added by SPARQL 1.1 Query will be briefly discussed:

- aggregate functions
- subqueries
- negation
- expressions with `SELECT`
- property paths

Again understand that at the time of this writing, SPARQL 1.1 Query is not finalized yet. The material presented here is based on the latest working drafts from SPARQL Working Group; it is therefore possible that the final standard will be more or less different. However, the basic ideas that will be discussed here should remain the same.

6.5.2.1 Aggregate Functions

If you are experienced with SQL queries, chance is that you are familiar with aggregate functions. These functions operate over the columns of a result table and can conduct operations such as counting, numerical averaging, or selecting the maximal/minimal data element from the given column. The current SPARQL query standard does not provide these operations, and if an application needs these functions, the application has to take a SPARQL query result set and calculate the aggregate values by itself.

Obviously, enabling a SPARQL engine to calculate aggregates for the users will make the application more light weighted, since the work will be done on the

SPARQL engine side. In addition, this will normally result in significantly smaller result set being returned to the application. If the SPARQL endpoint is accessed over HTTP, this will also help to lessen the traffic on the network.

With these considerations, SPARQL 1.1 will support aggregate functions. As usual, a query pattern yields a solution set, and from this solution set, a collection of columns will be returned to the user as the query result. An aggregation function will then operate on this set to create a new solution set which normally contains a single value representing the result from the aggregate function.

The following aggregate functions will be supported by SPARQL 1.1:

- `COUNT`
- `SUM`
- `MIN/MAX`
- `AVG`
- `GROUP_CONCAT`
- `SAMPLE`

Let us study some examples to understand more.

The first example queries about how many people have their e-mail addresses provided in a given FOAF document. List 6.34 shows the query itself:

List 6.34 Example of using `count()` aggregate function

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
SELECT count(*)
from <http://danbri.org/foaf.rdf>
WHERE {
   ?x a foaf:Person;
        foaf:mbox ?mbox.
}
```

This does not require much of an explanation at all. Note that at the time of this writing, this query is supported by the online Joseki endpoint which can be accessed at this URL:

`http://sparql.org/sparql.html`

If you are using other SPARQL endpoints, this query might not work well. In addition, even the SPARQL endpoint you are using does support aggregate functions, its implementation might require slightly different syntax.

The following example scans a given FOAF document and tries to sum up all the ages of the people included in this document, as shown in List 6.35:

List 6.35 Example of using `sum()` aggregate function

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
SELECT ( sum(?age) AS ?ages )
from <http://danbri.org/foaf.rdf>
```

```
WHERE {
    ?x a foaf:Person;
         foaf:age ?age.
}
```

the final result will be stored in the `ages` variable.

Again, this query has been tested by using online Joseki endpoint, and it works well. For the rest of this chapter, without explicitly mentioning, all the new SPARQL 1.1 features will be tested using the same Joseki endpoint. At the time when you are reading this book, you can either use Joseki endpoint or use your favorite one, which should be working as well.

The last example we would like to discuss here is the `SAMPLE` function. To put it simple, the newly added `SAMPLE` aggregate tries to solve the issue where it is not possible to project a particular variable or apply functions over that variable out of a `GROUP` since we are not grouping on that particular variable. Now, with `SAMPLE` aggregate function, this is very easy, as shown in List 6.36:

List 6.36 Example of using `SAMPLE()` aggregation function

```
SELECT ?subj SAMPLE(?obj)
from <http://danbri.org/foaf.rdf>
WHERE {
  ?subj ?property ?obj.
} GROUP BY ?subj
```

6.5.2.2 Subqueries

Subquery is not something new either. More specifically, it is sometimes necessary to use the result from one query to continue the next query.

For example, let us consider a simple request. Assume we would like to find all the friends I have, and for each one of them, I would like to know their e-mail addresses. As you know, a single query can be constructed by using the current SPARQL constructs to finish the task, there is no need to write two queries at all.

However, let us slightly change the request: find all the friends I have, and for each one of them, I would like to know their e-mail address and I only want one e-mail address for each friend I have.

Now to construct this query using the current SPARQL constructs, you will have to write two queries. The pseudo-code in List 6.37 shows the solution you will have to use:

List 6.37 Pseudo-code that finds friends and only one e-mail address of each friend

```
queryString = "
 SELECT ?person WHERE {
<http://www.liyangyu.com/foaf.rdf#liyang> foaf:knows ?friend.
 }";
```

```
resultSet = do_query(queryString);

foreach (result in resultSet) {
  person = result.get("friend");
  queryString = "SELECT ?mbox WHERE {
                    ?person foaf:mbox ?mbox.
                  } LIMIT 1";
  // do the query and get the e-mail address
}
```

Now, using the subquery feature provided by SPARQL 1.1, only one query is needed to finish the above task, as shown in List 6.38:

List 6.38 Using subquery feature to accomplish the same as by List 6.37

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

SELECT ?friend ?mbox
from <http://www.liyangyu.com/foaf.rdf>
WHERE {
<http://www.liyangyu.com/foaf.rdf#liyang> foaf:knows ?friend.
  {
    SELECT ?mbox WHERE {
      ?friend foaf:mbox ?mbox
    } LIMIT 1
  }
}
```

6.5.2.3 Negation

Negation is something that can be implemented by simply using the current version of SPARQL. Let us consider the request of finding all the people in a given FOAF document who do not have any e-mail address specified. The query in List 6.39 will accomplish this (and note that it only uses the language features from the current standard of SPARQL):

List 6.39 Find all the people from a given FOAF document who do not have any e-mail address specified

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

SELECT ?name
from <http://danbri.org/foaf.rdf>
WHERE {
   ?x foaf:givenName ?name.
   OPTIONAL { ?x foaf:mbox ?mbox }.
   FILTER (!BOUND(?mbox))
}
```

Let us understand `BOUND()` operator first. `BOUND()` operator is used as follows:

```
xsd:boolean BOUND(variable var)
```

It returns `true` if `var` is bounded to a value, it returns `false` otherwise. `BOUND()` operator is quite often used to test that a graph patter is *not* expressed by specifying an `OPTIONAL` graph pattern which uses a variable and then to test to see that the variable is not bound. This testing method is called *negation as failure* in logic programming.

With this said, the query in List 6.39 is easy to understand; it matches the people with a name but no expressed e-mail address. Therefore, it accomplishes what we have requested.

However, this negation as failure method is not quite intuitive and has a somewhat convoluted syntax. To fix this, SPARQL 1.1 has adopted at least one new operator called `NOT EXISTS`, and this will make the same query much intuitive and easier, as shown in List 6.40:

List 6.40 Example of negation using `NOT EXISTS` operator

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

SELECT ?name
from <http://danbri.org/foaf.rdf>
WHERE {
   ?x foaf:givenName  ?name.
   NOT EXISTS { ?x foaf:mbox ?mbox }.
}
```

The SPARQL Working Group has also considered another different design for negation, namely the `MINUS` operator. List 6.41 shows a possible query which uses `MINUS` operator. Again, this accomplishes the same goal with a cleaner syntax:

List 6.41 Example of negation using `MINUS` operator

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

SELECT ?name
from <http://danbri.org/foaf.rdf>
WHERE {
   ?x foaf:givenName ?name.
   MINUS {
     ?x foaf:mbox ?mbox.
   }
}
```

6.5.2.4 Expressions with SELECT

The expressions with `SELECT` feature added by SPARQL 1.1 is closely related to another feature called *projected expressions*. We will present these two closely related features together in this section.

In the current standard SPARQL Query language, a `SELECT` query (also called a *projection query*) may only project out variables bound in the query. More specifically, since variables can be bound only via triple pattern matching, it is impossible to project out values that are not matched in the underlying RDF dataset.

Expressions with `SELECT` query or projected expressions introduced by SPARQL 1.1 offers the ability for `SELECT` queries to project *any* SPARQL expression, rather than just bounded variables. A projected expression can be a variable, a constant URI/literal, or an arbitrary expression which may include functions on variables and constants. Functions could include both SPARQL built-in functions and extension functions supported by an implementation. Also, the variable used in the expression can be one of the following cases:

- a new variable introduced by `SELECT` clause (using the keyword `AS`);
- a variable binding already in the query solution; or
- a variable defined earlier in the `SELECT` clause.

List 6.42 shows one simple example:

List 6.42 Example of using expressions with SELECT query

```
prefix foaf:<http://xmlns.com/foaf/0.1/>
prefix fn:<http://www.w3.org/2005/xpath-functions>

SELECT fn:concat(?first, " ", ?last) AS ?name
from <http://danbri.org/foaf.rdf>
WHERE  {
   ?person foaf:firstName ?first;
           foaf:lastName ?last.
}
```

This query tries to find the first and last names of all the people included in a given FOAF document. Instead of simply showing the binding variables (`?first` and `?last`), the `SELECT` clause uses projected expression to first concatenate the first name and last name by using the binding variables `?first` and `?last`; the result is saved in a new variable called `?name` by using keyword `AS`. The query result includes only the `?name` variable and could be something like “Dan Brickley.”

The next example (List 6.43) is taken from W3C’s SPARQL Query Language 1.1 Working Draft,[4] since it is very helpful to show the usage of expressions in SELECT clause:

[4]http://www.w3.org/TR/2010/WD-sparql11-query-20100126/

List 4.43 Another example of using expressions in `SELECT` query

```
Data:
@prefix dc:   <http://purl.org/dc/elements/1.1/>.
@prefix :     <http://example.org/book/>.
@prefix ns:   <http://example.org/ns#>.

:book1  dc:title  "SPARQL Tutorial".
:book1  ns:price  42.
:book1  ns:discount 0.1.

:book2  dc:title  "The Semantic Web".
:book2  ns:price  23.
:book2  ns:discount 0.

Query:
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
PREFIX  ns:  <http://example.org/ns#>
SELECT  ?title (?p*(1-?discount) AS ?price)
   { ?x ns:price ?p.
     ?x dc:title ?title.
     ?x ns:discount ?discount
   }
```

In this case, the expression makes use of binding variables such as `?title` and `?discount`, and the final price (after the discount) is stored in the new variable `?price`. Table 6.6 shows the result of this query.

Table 6.6 Result of query in List 6.43

Title	Price
“The Semantic Web”	23
“SPARQL Tutorial”	37.8

6.5.2.5 Property Paths

We have covered several SPARQL 1.1 Query features so far at this point, including aggregate functions, subqueries, negation, and projected expressions. These features are currently marked by W3C’s SPARQL Query Language 1.1 Working Group as *required*. In addition to these required features, there are several other features being considered by the working group as *time permitting*. For example, property paths, basic federated query, and some commonly used SPARQL functions are all considered as time-permitting features.

For obvious reason, we will not cover these time-permitting features in details. Before we move on to SPARQL 1.1 Update, however, we will take a brief look at property paths, just to give you an idea of these time-permitting features.

If you have been writing quite a lot queries by using the current SPARQL standard, you have probably seen the cases where you need to follow paths to finish your query. More specifically, in order to find what you want, you need to construct a query that covers fixed-length paths to traverse along the hierarchical structure expressed in the given data store. List 6.44 shows one example. This query tries to find the name of my friend's friend:

List 6.44 Find the name of my friend's friend

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

SELECT ?name
from <http://www.liyangyu.com/foaf.rdf>
where {
  ?myself foaf:mbox <mailto:liyang910@yahoo.com>.
  ?myself foaf:knows ?friend.
  ?friend foaf:knows ?friendOfFriend.
  ?friendOfFriend foaf:name ?name.
}
```

As shown in List 6.44, the paths we have traversed include the following: `myself`, `friend`, `friendOfFriend`, `name of friendOfFriend`. This is quite long and cumbersome. The property paths featured by SPARQL Query 1.1 will make this a lot simpler for us. List 6.45 shows the tentative syntax of this feature:

List 6.45 Example of property path

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

SELECT ?name
from <http://www.liyangyu.com/foaf.rdf>
where {
  ?myself foaf:mbox <mailto:liyang910@yahoo.com>.
  ?myself foaf:knows/foaf:knows/foaf:name ?name.
}
```

This accomplishes exactly the same goal as the query in List 6.44, but with a much cleaned syntax.

6.5.3 SPARQL 1.1 Update

If you are experienced with SQL queries, you know how easy it is to change the data in the database. There are SQL statements provided for you to do that, you don't have to know the mechanisms behind these statements.

We all know that SPARQL to RDF data stores is as SQL to databases. However, changing an RDF graph is not as easy as updating a table in a database. To add, update, or remove statements from a given RDF graph, there is no SPARQL language constructs we can use; instead, we have to use a programming language and a set of third-party APIs to accomplish this.

To allow an RDF graph or an RDF store to be manipulated the same way as SQL to database, a language extension to the current standard of SPARQL is proposed. Currently, this language extension is called SPARQL Update 1.1, and it includes the following features:

- Insert new triples into an RDF graph.
- Delete triples from an RDF graph.
- Perform a group of update operations as a single action.
- Create a new RDF graph in a graph store.
- Delete an RDF graph from a graph store.

The first three operations are called *graph update*, since they are responsible for addition and removal of triples from one specific graph. The next two operations are called *graph management*, since they are responsible for creating and deleting graphs within a given graph store.

In this section, we will discuss these operations briefly. Again, understand this standard is not finalized yet, and by the time you are reading this book, it could be changed or updated. However, the basic idea should remain the same.

6.5.3.1 Graph Update: Adding RDF Statements

One way to add one or more RDF statements into a given graph is to use the `INSERT DATA` operation. This operation creates the graph if it does not exist. List 6.46 shows one example:

List 6.46 Example of using `INSERT DATA` to update a given graph

```
prefix foaf:<http://xmlns.com/foaf/0.1/>
prefix liyang: <http://www.liyangyu.com/foaf.rdf#>
INSERT DATA
{
  GRAPH <http://www.liyangyu.com/foaf.rdf>
  {
 liyang:liyang foaf:workplaceHomepage <http://www.delta.com> ;
                  foaf:schoolHomepage  <http://www.osu.edu>.
   }
}
```

This will insert two new statements into my personal FOAF document, namely http://www.liyangyu.com/foaf.rdf. And these two statements show the home page of the company I work for and the home page of the school I graduated from.

Note that you can insert any number of RDF statements within one `INSERT DATA` request. In addition, the `GRAPH` clause is optional; an `INSERT DATA` request without the `GRAPH` clause will simply operate on the default graph in the RDF store.

Another way to add one or more RDF statements into a given graph is to use the `INSERT` operation. List 6.47 shows one example to use the `INSERT` operation. This example copies RDF statement(s) from my old FOAF document into my current FOAF document. More specifically, all the information about my interests are moved into the new FOAF document so I don't have to add each one of them manually.

List 6.47 Example of using `INSERT` operation to update a given graph

```
prefix foaf:<http://xmlns.com/foaf/0.1/>
prefix liyang: <http://www.liyangyu.com/foaf.rdf#>

INSERT
{
   GRAPH <http://www.liyangyu.com/foaf.rdf>
   { liyang:liyang foaf:topic_interest ?interest. }
}
WHERE
{
   GRAPH <http://www.liyangyu.com/foafOld.rdf>
   { liyang:liyang foaf:topic_interest ?interest. }
}
```

This is a very useful operation, and with this operation, we can move statements from one graph to another based on any graph pattern we have specified by using the `WHERE` clause.

6.5.3.2 Graph Update: Deleting RDF Statements

Similar to adding statements, deleting statements from a given RDF graph can be done in two ways. One way is to use the `DELETE DATA` operation, which removes triples from a graph. List 6.48 shows one example:

List 6.48 Example of using `DELETE DATA` operation to update a given graph

```
prefix foaf:<http://xmlns.com/foaf/0.1/>
prefix liyang: <http://www.liyangyu.com/foaf.rdf#>
DELETE DATA
{
  GRAPH <http://www.liyangyu.com/foaf.rdf>
  {
    liyang:liyang foaf:workplaceHomepage
           <http://www.delta.com> ;
                    foaf:schoolHomepage  <http://www.osu.edu>.
   }
}
```

This will delete the two statements we have just added into my FOAF document. Similarly, you can delete any number of statements in one `DELETE DATA` request, and the `GRAPH` clause is optional; an `DELETE DATA` statement without the `GRAPH` clause will simply operate on the default graph in the RDF store.

Another way to delete one or more RDF statements from a given graph is to use the `DELETE` operation. List 6.49 shows one example. And as you can tell, we will be able to specify a graph pattern using `WHERE` clause so as to delete statements more effectively. In this particular example, we would like to delete all the e-mail information that have been included in my FOAF document:

List 6.49 Example of using `DELETE` operation to update a given graph

```
prefix foaf:<http://xmlns.com/foaf/0.1/>
DELETE
{
   GRAPH <http://www.liyangyu.com/foaf.rdf>
   { ?person foaf:mbox ?mbox. }
}
```

Similar to `INSERT` operation, `DELETE` operation can have a `WHERE` clause to specify more interesting graph pattern one can match, as shown in List 6.50:

List 6.50 Example of using `DELETE` operation together with `WHERE` clause

```
prefix foaf:<http://xmlns.com/foaf/0.1/>
prefix dc: <http://purl.org/dc/elements/1.1/>
prefix liyang: <http://www.liyangyu.com/foaf.rdf#>

DELETE
{
   GRAPH <http://www.liyangyu.com/foaf.rdf>
   { ?logItem ?pred ?obj. }
}
WHERE
{
   GRAPH <http://www.liyangyu.com/foaf.rdf>
   {
      liyang:liyang foaf:weblog ?logItem;
      ?logItem dc:date ?date.
      FILTER ( ?date < "2005-01-01T00:00:00-2:00"^^xsd:
        dateTime )
      ?logItem ?pred ?obj
   }
}
```

As you can tell, this will delete all the blog items I have written before January 1st of 2005.

6.5.3.3 Graph Update: **LOAD** and **CLEAR**

Two more operations that can be useful are the LOAD and CLEAR operations. With what we have learned so far, these two operations are not necessarily needed since their functionalities can be implemented by simply using INSERT and DELETE operations. However, they do provide a more convenient and effective choice when needed.

The LOAD operation copies all the triples of a remote graph into the specified target graph. If no target graph is specified, the defect graph will be used. For example, List 6.51 will load all the statements from my old FOAF document to my current FOAF document:

List 6.51 Example of using LOAD operation to update a given graph

```
LOAD <http://www.liyangyu.com/foafOld.rdf>
INTO <http://www.liyangyu.com/foaf.rdf>
```

The CLEAR operation deletes all the statements from the specified graph. If no graph is specified, it will operate on the default graph. Note that this operation does not remove the graph from the RDF graph store. For example, List 6.52 will delete all the statements from my old FOAF document:

List 6.52 Example of using CLEAR operator to update a given graph

```
CLEAR GRAPH <http://www.liyangyu.com/foafOld.rdf>
```

6.5.3.4 Graph Management: Graph Creation

As we have mentioned earlier, graph management operations create and destroy named graphs in the graph store. Note that, however, these operations are optional since based on the current standard, graph stores are not required to support named graphs.

The following is used to create a new named graph:

```
CREATE [SILENT] GRAPH <uri>
```

This creates a new empty graph whose name is specified by *uri*. After the graph is created, we can manipulate its content by adding new statements into it, as we have discussed in previous sections.

Note that the optional SILENT keyword is for error handling. If the graph named *uri* already exists in the store, unless this keyword is present, the SPARQL 1.1 Update service will flag an error message back to the user.

6.5.3.5 Graph Management: Graph Removal

The following operation will remove the specified named graph from the graph store:

```
DROP [SILENT] GRAPH <uri>
```

Once this operation is successfully completed, the named graph cannot be accessed with further operations. Similarly, SPARQL 1.1 Update service will report an error message if the named graph does not exist. If the optional `SILNET` keyword is present, no error message will be generated.

6.6 Summary

We have covered SPRAQL in this chapter, the last core technical component of the Semantic Web.

First off, understand how SPARQL fits into the technical structure of the Semantic Web and how to set up and use a SPARQL endpoint to submit queries against RDF models.

Second, understand the following main points about SPARQL query language:

- basic SPARQL query language concepts such as triple pattern and graph pattern;
- basic SPARQL query forms such as `SELECT` query, `ASK` query, `DESCRIBE` query and `CONSTRUCT` query;
- key SPARQL language features and constructs, and use them effectively to build queries, including working with multiple graphs.

Finally, this chapter has also covered SPARQL 1.1, a collection of new features added to the current SPARQL standard. Make sure you understand the following about SPARQL 1.1:

- the SPARQL 1.1 technical components;
- the language features and constructs of SPARQL 1.1 Query, including aggregate functions, subqueries, negation, expressions with `SELECT` query, and property paths;
- the language features and constructs of SPARQL 1.1 Update, including inserting and deleting operations on a single graph, creating and deleting graphs from a graph store.

At this point in the book, we have covered all the core technical components of the Semantic Web. The next five chapters will give you some concrete examples of the Semantic Web at work, which will further enhance your understanding about the materials presented so far in the book.