CHAPTER 7

# Querying

While machine-readable datasets are published primarily for software agents, automatic data extraction is not always an option. Semantic Information Retrieval often involves users searching for the answer to a complex question, based on the formally represented knowledge in a dataset or database. While Structured Query Language (SQL) is used to query relational databases, querying graph databases and flat Resource Description Framework (RDF) files can be done using the SPARQL Protocol and RDF Query Language (SPARQL), the primary query language of RDF, which is much more powerful than SQL. SPARQL is a standardized language capable of querying local and online RDF files, Linked Open Data (LOD) datasets, and graph databases; constructing new RDF graphs, based on the information in the queried graphs; adding new RDF statements to or deleting triples from a graph; inferring logical consequences; and federating queries across different repositories. SPARQL can query multiple data sources at once, to dynamically merge the smaller graphs into a large supergraph. While graph databases often have a proprietary query language (often based on or extending SPARQL), most publicly available datasets have a SPARQL endpoint, from which you can run SPARQL queries. As for the developers, many Semantic Web software tools provide a SPARQL application programming interface (API) for programmatic access.

## SPARQL: The Query Language for RDF

As mentioned before, the primary *query language* of RDF is *SPARQL* (pronounced "sparkle," a recursive acronym for *SPARQL Protocol and RDF Query Language*), which can be used to retrieve and manipulate information stored in RDF or in any format that can be retrieved as RDF [1]. The output can be a result set or an RDF graph.

### Structure and Syntax

SPARQL uses a Notation3-like syntax. The URIs can be written in full between the less than (<) and greater than (>) characters (see Listing 7-1) or abbreviated using the namespace mechanism with the `PREFIX` keyword (see Listing 7-2).

***Listing 7-1.*** Full URI Syntax in SPARQL

```
<http://example.com>
```

***Listing 7-2.*** Using the Namespace Mechanism in SPARQL

```
PREFIX schema: <http://schema.org/>
```

After declaring the Schema.org namespace (`http://schema.org/`), for example, `http://schema.org/Person` can be abbreviated as `schema:Person`. The *default namespace* of a SPARQL query can be set by using the `PREFIX` directive with no prefix (e.g., `PREFIX : <http://yourdefaultnamespace.com/>`), so that you can use empty prefixes in your queries, such as `as ?a :knows ?b`. Similar to N3, the URI `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` or `rdf:type` can be abbreviated as `a`. Literals can be written with or without a language tag and typing. Plain string literals are delimited by quotation marks, such as `"a plain literal"`, while plain literals including a language tag end in the `@` sign and the standard language code, such as `"Wagen"@de` (the word *car* in German). Typed literals are written analogously to the typed literals in RDF, as, for example, `"55"^^xsd:integer` (55 is an integer number rather than two meaningless characters in a string literal). Frequently used typed literals can be abbreviated such that `"true"^^xsd:boolean` corresponds to `true`, while integer and decimal numbers are automatically assumed to be of type `xsd:integer` or `xsd:decimal`, respectively. As a consequence, `"5"^^xsd:integer` can be abbreviated as `5`, while `"13.1"^^xsd:decimal` can be written as `13.1`.

Each SPARQL query has a head and a body. The head of a SPARQL query is an expression for constructing the answer for the query. The evaluation of a query against an RDF graph is performed by checking whether the body is matched against the graph, which results in a set of bindings for the variables in the body. These bindings are processed using relational operators such as projection and distinction to generate the output for the query. The body can be a simple triple pattern expression or a complex RDF graph pattern expression containing triple patterns, such as subject-predicate-object RDF triples, where each subject, predicate, or object can be a variable. The body can also contain conjunctions, disjunctions, optional parts, and variable value constraints (see Figure 7-1).

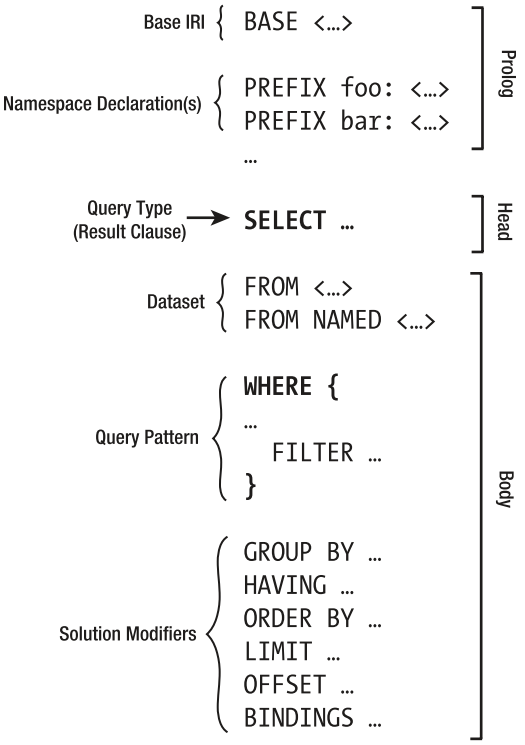


***Figure 7-1.*** *The structure of SPARQL queries*

The BASE directive, the namespace declarations (PREFIX), the dataset declaration (FROM, FROM NAMED), and the query modifiers (GROUP BY, HAVING, ORDER BY, LIMIT, OFFSET, BINDINGS) are optional. The BASE directive and the list of prefixes are used to abbreviate URIs. The BASE keyword defines the base URI against which all relative URIs in the query are resolved. The list of prefixes can contain an arbitrary number of PREFIX statements. The prefix abbreviation pref preceding the semicolon represents the prefix URI, which can be used throughout the SPARQL query, making it unnecessary to repeat long URIs (standard namespace mechanism). The FROM clause specifies the default graph to search. The FROM NAMED clause can be used to specify a named graph to query. In some cases, as, for example, when the SPARQL endpoint used for the query is dedicated to the LOD dataset from which you want to retrieve data, the FROM clause is optional and can be safely omitted. The WHERE clause specifies the patterns used to extract the desired results. The query modifiers, such as ORDER BY or LIMIT, if present, are in the last part of the query.

## SPARQL 1.0 and SPARQL 1.1

The first version of SPARQL, SPARQL 1.0, was released in 2008 [2]. SPARQL 1.0 introduced the SPARQL grammar, the SPARQL query syntax, the RDF term constraints, the graph patterns, the solution sequences and solution modifiers, and the four core query types (SELECT, CONSTRUCT, ASK, and DESCRIBE). SPARQL 1.0 has been significantly extended with new features in SPARQL 1.1 [3].

For example, SPARQL 1.1 supports *aggregation*. To perform aggregation, first you have to segregate the results into groups, based on the expression(s) in the GROUP BY clause. Then, you evaluate the projections and aggregate functions in the SELECT clause, to get one result per group. Finally, the aggregated results have to be filtered in a HAVING clause.

The *SPARQL 1.1 Update language* supports graph update operations (INSERT DATA, DELETE DATA, DELETE/INSERT, LOAD, CLEAR) and graph management operations (CREATE, DROP, COPY, MOVE, and ADD) [4]. The INSERT DATA operation adds some triples written inline in the request into the graphstore. The DELETE DATA operation is used to remove RDF triples, if the respective graphs in the graphstore contain them. The DELETE/INSERT operation can be used to remove triples from, or add triples to, the graphstore, based on bindings for a query pattern specified in a WHERE clause. The LOAD operation reads an RDF document from an internationalized resource identifier (IRI) and inserts its triples into the specified graph in the graphstore. The CLEAR operation removes all the triples in the specified graph(s) in the graphstore. The CREATE operation creates a new graph in the graphstore. The DROP operation removes a graph and all of its contents. The COPY operation modifies a graph to contain a copy of another graph. In other words, it inserts all data from an input graph into a destination graph. The MOVE operation moves all of the data from one graph into another graph. The ADD operation reproduces all data from one graph into another graph. It is also possible to update RDF graphs through a protocol known as the *SPARQL 1.1 Uniform HTTP Protocol* [5].

The *SPARQL 1.1 Service Description* specification [6] provides a method for discovering information about SPARQL services, such as the supported extension functions, and details of the default dataset. It also has a vocabulary for describing SPARQL services, which has the namespace IRI http://www.w3.org/ns/sparql-service-description# and the prefix sd. In Semantic Web applications, it is not always possible to explicitly write graph structures for graph pattern matching, and that is why SPARQL 1.1 defines semantic entailment relations called *entailment regimes* [7]. These standard semantic entailment relations can be used in applications that rely on RDF statements inferred from explicitly given assertions, so that the graph pattern matching is performed using semantic entailment relations instead of explicitly given graph structures. SPARQL 1.1 supports additional serialization formats for the query output, including JSON [8], CSV, and TSV [9], beyond the formats supported by SPARQL 1.0, such as XML [10]. Beyond the four core SPARQL query types introduced in SPARQL 1.0, SPARQL 1.1 also supports reasoning queries and federated queries, as you will see in the next section.

## Query Types

The optional namespace declarations are followed by the *query*. The four core query types in SPARQL are the `SELECT`, the `ASK`, the `CONSTRUCT`, and the `DESCRIBE` queries. `SELECT` queries provide a value selection for the variables matching the query patterns. The yes/no queries (`ASK` queries) provide a Boolean value. `CONSTRUCT` queries create new RDF data from the above values, as well as resource descriptions. `DESCRIBE` queries return a new RDF graph containing matched resources. The most frequently used SPARQL queries are the `SELECT` queries.

Beyond the basic query types, SPARQL 1.1 also supports reasoning through `REASON` queries and executes queries distributed over different SPARQL endpoints (*federated queries*), using the `SERVICE` keyword [11].

## Pattern Matching

The query output result clause is followed by the *pattern matching*. Two different pattern types can be used in SPARQL queries: the *triple patterns* and the *graph patterns*. The SPARQL triple patterns are similar to the subject-predicate-object triples of RDF, but they can also include *variables*. This makes it possible to select RDF triples from an RDF graph that match your criteria described in the pattern. Any or all subject, predicate, or object values can be variables, all of which are identified by a question mark[1] preceding the string, such as `?name`. To match an exact RDF triple, you have to write the subject-predicate-object names followed by a `.`, such as shown in Listing 7-3.

***Listing 7-3.*** Exact RDF Triple Matching in SPARQL

```
ex:Person schema:familyName "Sikos" .
```

To match one variable, you have to substitute the appropriate triple component (the subject, the predicate, or the object) with a variable (see Listing 7-4).

***Listing 7-4.*** Matching One Variable in SPARQL

```
?person schema:familyName "Sikos" .
```

A variable is not limited to any part of the triple pattern. You can substitute any triple components (the subject, the predicate, or the object) with variables (see Listing 7-5).

***Listing 7-5.*** Matching Multiple Variables in SPARQL

```
?person schema:familyName ?name .
```

Even all components can be variables. For instance, the triple pattern `?subject ?object ?name` will match all triples in your RDF graph. Sometimes, much more complex selection rules are required than what you can express in a single triple pattern. A collection of triple patterns is called a *graph pattern* and is delimited by curly braces (see Listing 7-6).

***Listing 7-6.*** A Graph Pattern in SPARQL

```
{
  ?who schema:name ?name.
  ?who iswc:research_topic ?research_topic.
  ?who foaf:knows ?others.
}
```

[1]Alternatively, the dollar sign (`$`) can be used.

Graph patterns can be used for matching optional parts, creating a union of patterns, nesting, filtering values of possible matchings, and choosing the data source to be matched by the pattern. As a result, a graph pattern will find all resources with all the desired properties written in the pattern. Graph patterns make it possible to write complex queries, because each returned resource can be substituted into all occurrences of the variable, in case the same variable is used in multiple triple patterns. This leads to a truly sophisticated selection unknown to the conventional Web, whereby you cannot use multiple filtering in searches beyond some basic operators, such as AND, OR, or XOR. However, SPARQL supports filter functions too, including logical (`!`, `&&`, `||`, `=`, `!=`, `<`, `<=`, `>`, and `>=`) and mathematical operations (`+`, `-`, `*`, `/`), as well as comparisons (`=`, `!=`, `>`, `<`). SPARQL has built-in tests for checking web addresses, blank graph nodes, literals, and bounds (`isURI`, `isBlank`, `isLiteral`, `bound`), accessors such as `str`, `datatype`, and `lang` (see Listing 7-7), and other functions, such as `sameTerm`, `langMatches`, and `regex`, for checking same terms, language matching, and writing regular expressions.

***Listing 7-7.*** Language Checking in SPARQL

```
lang(?title)="en"
```

Beyond the SPARQL 1.0 operators and functions, SPARQL 1.1 also supports existence-checking functions (`EXISTS`, `NOT EXISTS`), both of which can be used as part of a graph pattern (such as in Listing 7-8, to find persons who do not have an e-mail address) as well as in `FILTER` expressions.

***Listing 7-8.*** Existence Checking in a Graph Pattern

```
SELECT ?person
WHERE
{
  ?person rdf:type foaf:Person .
  NOT EXISTS { ?person foaf:mbox ?email }
}
```

SPARQL 1.1 also has additional functions, such as `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`, `SAMPLE`, and `GROUP_CONCAT`. Furthermore, SPARQL 1.1 supports *property paths* that allow triple patterns to match arbitrary-length paths through a graph. Predicates are combined with operators similar to regular expressions (see Table 7-1).

***Table 7-1.*** *Property Path Constructs in SPARQL 1.1*

| Construct | Meaning |
|---|---|
| `path1/path2` | Forwards path (`path1` followed by `path2`) |
| `^path1` | Backwards path (object to subject) |
| `path1\|path2` | Either `path1` or `path2` |
| `path1*` | `path1`, repeated zero or more times |
| `path1+` | `path1`, repeated one or more times |
| `path1?` | `path1`, optionally |
| `path1{m,n}` | At least `m` and no more than `n` occurrences of `path1` |
| `path1{n}` | Exactly `n` occurrences of `path1` |
| `path1{m,}` | At least `m` occurrences of `path1` |
| `path1{,n}` | At most `n` occurrences of `path1` |

## Solution Modifiers

The last optional part of the SPARQL queries are the *solution modifiers*. Once the output of the pattern has been computed (in the form of a table of values of variables), solution modifiers allow you to modify these values, applying standard classical operators such as projection,[2] `DISTINCT` (removes duplicates), `ORDER` (sorting mechanism), and `LIMIT` (sets the maximum number of results returned).

## SELECT Queries

The most common SPARQL queries are the `SELECT` queries. The `SELECT` clause specifies data items (variable bindings) to be returned by the SPARQL query. Even through LOD datasets can contain thousands or even millions of RDF triples, you can select those items that meet your criteria. For example, from a writer's dataset, you can list those writers who lived in the 20th century or those who are American. SPARQL supports joker characters, so you can select all variables mentioned in the query, using `SELECT *`. If you want to eliminate potential duplicates, use the `DISTINCT` keyword after `SELECT`, such as `SELECT DISTINCT ?var.` `SELECT` queries are often used to extract triples through specific variables and expressions. For example, assume we need a query to extract all names mentioned in someone's FOAF file declared using `foaf:name`. The abbreviation of the namespace requires a `PREFIX` declaration. The query is a `SELECT` query, which uses a variable for the names (`?name`), and a `WHERE` clause with a triple pattern to find all subjects (`?person`) and objects (`?name`) linked with the `foaf:name` predicate (see Listing 7-9).

***Listing 7-9.*** A `SELECT` Query to Find Subjects and Objects Linked with a Declared Predicate

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
  ?person foaf:name ?name .
}
```

If we need all those people from an FOAF file who have an e-mail address specified, we have to declare two predicates, one for the name (`foaf:name`) and another for the e-mail address (`foaf:mbox`), while all the subjects (`?person`) and objects (`?name`, `?email`) are variables (see Listing 7-10).

***Listing 7-10.*** A `SELECT` Query to Find Subjects and Objects Linked with Two Different Predicates

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
SELECT ?name ?email
WHERE {
  ?person foaf:name ?name .
  ?person foaf:mbox ?email .
}
```

The output will contain all the names and e-mail addresses.

[2]Only those expressions that consist of aggregates and constants can be projected in SPARQL query levels using aggregates. The only exception is using `GROUP BY` with one or more simple expressions consisting of one variable only, which variable can be projected from that level.

■ **Note** If the persons are described using `Schema.org/Person`, the names can be expressed using the `givenName` and `familyName` properties, and the e-mail address using the `email` property, while the namespace has to be modified to `http://schema.org/`.

The results of the `SELECT` queries are typically displayed as a table of values (in HTML, XML, or JSON).

## Filtering

If we have to extract all those landlocked countries from DBpedia that have a population larger than 5 million, we need the `FILTER` keyword in the `WHERE` clause (see Listing 7-11).

***Listing 7-11.*** A `SELECT` Query with a Filter to Extract All Landlocked Countries from DBpedia with More Than 5 Million Inhabitants

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX type: <http://dbpedia.org/class/yago/>
PREFIX prop: <http://dbpedia.org/property/>
SELECT ?country_name ?population
WHERE {
    ?country a type:LandlockedCountries ;
             rdfs:label ?country_name ;
             prop:populationEstimate ?population .
    FILTER (?population > 5000000) .
}
```

The Boolean condition(s) provided will filter out the unwanted query results, in this case, all those landlocked countries that have fewer than 5 million inhabitants.

■ **Note** The preceding example uses the `;` shortcut to separate triple patterns sharing the same subject `?country`.

## ASK Queries

If you need the answer to a yes/no question, you can use the ASK query in SPARQL. For instance, you can query DBpedia to find out whether the Amazon is longer than the Nile (see Listing 7-12).

***Listing 7-12.*** An `ASK` Query in SPARQL

```
PREFIX prop: <http://dbpedia.org/property/>
ASK
{
  <http://dbpedia.org/resource/Amazon_River> prop:length ?amazonLength .
  <http://dbpedia.org/resource/Nile> prop:length ?nileLength .
  FILTER(?amazon > ?nile) .
}
```

The result of `ASK` queries is either `true` or `false`. In our example, the output is `true`.

## CONSTRUCT Queries

SPARQL can be used not only to retrieve information from datasets but also to create new graphs, or to reshape existing RDF graphs by adding new triples. Such queries are called CONSTRUCT queries. Assume you want to extend your family tree description by adding a grandmother. To do so, you have to identify the gender and parent-child relationships of the other family members (see Listing 7-13).

***Listing 7-13.*** Preparing a CONSTRUCT Query

```
:Ben   :hasParent  :Christina ;
       :gender     :male .
:Luke  :hasParent  :Linda ;
       :gender     :male .
:Christina :hasParent :Anna ;
       :gender    :female .
:Linda :hasParent :Anna ;
       :gender    :female .
:Anna  :gender    :female .
```

The next step is to run a CONSTRUCT query to create new triples based on the preceding ones, to specify who is whose grandmother (see Listing 7-14).

***Listing 7-14.*** A CONSTRUCT Query

```
PREFIX : <http://samplefamilytreeonto.com/>

CONSTRUCT { ?p :hasGrandmother ?g . }

WHERE {?p      :hasParent ?parent .
       ?parent :hasParent ?g .
       ?g      :gender    :female .}
```

The newly constructed triples describe the relationship between the two grandchildren and their grandmother (see Listing 7-15).

***Listing 7-15.*** A CONSTRUCT Query Generates New Triples

```
:Ben
      :hasGrandmother  :Anna .

:Luke
      :hasGrandmother  :Anna .
```

## DESCRIBE Queries

The DESCRIBE queries describe the resources matched by the given variables. For example, if you run a DESCRIBE query on a dataset of countries (see Listing 7-16), the output will be all the triples related to the queried country (see Listing 7-17).

***Listing 7-16.*** A DESCRIBE Query

```
DESCRIBE ?country
```

***Listing 7-17.*** The Output of a DESCRIBE Query

```
ex:Hungary a geo:Country;
ex:continent geo:Europe;
ex:flag <http://yourwebsite.com/img/flag-hun.png> ;
…
```

## Federated Queries

In SPARQL 1.1, queries can issue a query on another SPARQL endpoint during query execution. These queries are called *federated queries*, in which the remote SPARQL endpoint is declared by the SERVICE keyword, which sends the corresponding part of the query to the remote SPARQL endpoint. For example, if the remote SPARQL endpoint is DBpedia's endpoint, a federated query can be written as shown in Listing 7-18.

***Listing 7-18.*** A Federated Query in SPARQL 1.1

```
SELECT DISTINCT ?person
WHERE {
  SERVICE <http://dbpedia.org/sparql> { ?person a <http://schema.org/Person> . }
} LIMIT 10
```

A sample output of this query is shown in Listing 7-19, identifying ten persons from DBpedia.

***Listing 7-19.*** Federated Query Result Example

```
-------------------------------------------------------
| person                                              |
=======================================================
| <http://dbpedia.org/resource/%C3%81ngel_Gim%C3%A9nez> |
| <http://dbpedia.org/resource/Aaron_Lines>           |
| <http://dbpedia.org/resource/Abel_Lafleur>          |
| <http://dbpedia.org/resource/Ada_Maimon>            |
| <http://dbpedia.org/resource/Adam_Krikorian>        |
| <http://dbpedia.org/resource/Albert_Constable>      |
| <http://dbpedia.org/resource/Alex_Reid_(actress)>   |
| <http://dbpedia.org/resource/Alex_Reid_(art_dealer)> |
| <http://dbpedia.org/resource/Alex_Reid_(fighter)>   |
| <http://dbpedia.org/resource/Alex_Reid_(footballer)> |
-------------------------------------------------------
```

## REASON Queries

In SPARQL 1.1, reasoning can be performed by executing a SPARQL query with the REASON keyword, followed by a rule set in an ontology or declarative language (declared as a URL to a rule delimited by <> or inline N3 rules between curly braces), and a combination of an OVER and a WHERE clause to define the triples for reasoning.

For example, to list all the acquaintances of Leslie Sikos from two different datasets, you can write a federated query with reasoning, as shown in Listing 7-20, regardless of whether Leslie Sikos listed them as acquaintances or other people stated that they know him.

***Listing 7-20.*** Find Acquaintances Regardless of the Relationship Direction

```
REASON {
  { ?x foaf:knows ?y } => { ?y foaf:knows ?x }
}
OVER {
  :LeslieSikos foaf:knows ?person .
}
WHERE {
  {
    SERVICE <http://examplegraph1.com/sparql> { :LeslieSikos foaf:knows ?person . }
  } UNION {
    SERVICE <http://examplegraph2.com/sparql> { :LeslieSikos foaf:knows ?person . }
  }
}
```

## URL Encoding of SPARQL Queries

In order to provide the option for automated processes to make SPARQL queries, SPARQL can be used over HTTP, using the SPARQL Protocol (abbreviated by the *P* in SPARQL). SPARQL endpoints can handle a SPARQL query with parameters of an HTTP `GET` or `POST` request. The query is URL-encoded to escape special characters and create the query string as the value of the `query` variable. The parameters are defined in the standardized SPARQL Protocol [12]. As an example, take a look at the default DBpedia SPARQL endpoint (`http://dbpedia.org/sparql/`) query shown in Listing 7-21.

***Listing 7-21.*** A URL-Encoded SPARQL Query

```
http://dbpedia.org/sparql?default-graph-uri=http%3A%2F%2Fdbpedia.org&query=select+distinct+
%3FConcept+where+{[]+a+%3FConcept}+LIMIT+100&format=text%2Fhtml&timeout=30000&debug=on
```

---

■ **Note** The second parameter (`default-graph-uri`) is DBpedia's proprietary implementation, which extends the standard URL-encoded SPARQL query.

---

## Graph Update Operations

In SPARQL 1.1, new RDF triples can be added to a graph, using the `INSERT DATA` operation. If the destination graph does not exist, it is created. As an example, assume there are two RDF statements about a book in a graph, title and format (see Listing 7-22).

***Listing 7-22.*** Data Before the INSERT DATA Operation

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

<http://www.lesliesikos.com/mastering-structured-data-on-the-semantic-web/> dc:title ↩
 "Mastering Structured Data on the Semantic Web" .
<http://www.lesliesikos.com/mastering-structured-data-on-the-semantic-web/> ↩
 schema:bookFormat schema:Paperback .
```

To add two new triples to this graph about the author and the language the book was written in, you can use the `INSERT DATA` operation, as shown in Listing 7-23. Because the subject is the same for both triples, it has to be declared just once in a semicolon-separated list.

***Listing 7-23.*** Adding New Triples to a Graph, Using the INSERT DATA Operation

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX schema: <http://schema.org/>

INSERT DATA
{
  <http://www.lesliesikos.com/mastering-structured-data-on-the-semantic-web/> dc:creator ↩
   "Leslie Sikos" ;
   schema:inLanguage "English" .
}
```

As a result, the graph will contain four triples about the book, as demonstrated in Listing 7-24.

***Listing 7-24.*** Data After the INSERT DATA Operation

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

<http://www.lesliesikos.com/mastering-structured-data-on-the-semantic-web/> dc:title ↩
 "Mastering Structured Data on the Semantic Web" .
<http://www.lesliesikos.com/mastering-structured-data-on-the-semantic-web/> ↩
 schema:bookFormat schema:Paperback .
<http://www.lesliesikos.com/mastering-structured-data-on-the-semantic-web/> dc:creator ↩
 "Leslie Sikos" .
<http://www.lesliesikos.com/mastering-structured-data-on-the-semantic-web/> dc:inLanguage ↩
 "English" .
```

SPARQL 1.1 also supports the removal of RDF triples, using the `DELETE DATA` operation. For example, to remove the book format and the language of the book from Listing 7-24, you declare the prefixes, use the `DELETE DATA` operation, and list the statements to remove (see Listing 7-25).

***Listing 7-25.*** Removing Triples from a Graph, Using the DELETE DATA Operation

```
PREFIX schema: <http://schema.org/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

DELETE DATA
{
  <http://www.lesliesikos.com/mastering-structured-data-on-the-semantic-web/> ↩
   schema:bookFormat schema:Paperback ; dc:inLanguage "English" .
}
```

## Graph Management Operations

In SPARQL 1.1, RDF statements can be copies from the default graph to a named graph using the `COPY` operation. As an example, assume we have the triples shown in Listing 7-26.

***Listing 7-26.*** Data Before Copying

```
# Default Graph

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

<http://examplegraph.com/Leslie> a foaf:Person .
<http://examplegraph.com/Leslie> foaf:givenName "Leslie" .
<http://examplegraph.com/Leslie> foaf:mbox  <mailto:leslie@examplegraph.com> .

# Graph http://examplenamedgraph.com

<http://examplenamedgraph.com/Christina> a foaf:Person .
<http://examplenamedgraph.com/Christina> foaf:givenName "Christina" .
```

All triples of the default graph can be copied to the named graph with the `COPY` operation, as shown in Listing 7-27.

***Listing 7-27.*** A COPY DEFAULT TO Operation SPARQL 1.1

```
COPY DEFAULT TO <http:/examplenamedgraph.com>
```

The result of the `COPY DEFAULT TO` operation is shown in Listing 7-28.

***Listing 7-28.*** Data After the `COPY DEFAULT TO` Operation

```
# Default Graph

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

<http://examplegraph.com/Leslie> a foaf:Person .
<http://examplegraph.com/Leslie> foaf:givenName "Leslie" .
<http://examplegraph.com/Leslie> foaf:mbox  <mailto:leslie@examplegraph.com> .

# Graph http://examplenamedgraph.com

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

<http://examplegraph.com/Leslie> a foaf:Person .
<http://examplegraph.com/Leslie> foaf:givenName "Leslie" .
<http://examplegraph.com/Leslie> foaf:mbox  <mailto:leslie@examplegraph.com> .
```

---

■ **Note** The original content of the named graph is lost by the `COPY` operation.

---

Similarly, RDF statements can be moved from the default graph to a named graph, using the `MOVE` operation. For example, assume you have the data shown in Listing 7-29.

***Listing 7-29.*** Data Before the `MOVE DEFAULT TO` Operation

```
# Default Graph

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

<http://examplegraph.com/Nathan> a foaf:Person .
<http://examplegraph.com/Nathan> foaf:givenName "Nathan" .
<http://examplegraph.com/Nathan> foaf:mbox  <mailto:nathan@examplegraph.com> .

# Graph http://examplenamedgraph.com

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

<http://examplenamedgraph.com/Peter> a foaf:Person .
<http://examplenamedgraph.com/Peter> foaf:givenName "Peter" .
```

To move all RDF statements from the default graph into a named graph, you can use the `MOVE` operation, as shown in Listing 7-30.

***Listing 7-30.*** A `MOVE DEFAULT TO` Operation

```
MOVE DEFAULT TO http://examplenamedgraph.com
```

---

■ **Note** The original content of the named graph is lost by the `MOVE` operation (see Listing 7-31).

---

***Listing 7-31.*** Data After the `MOVE DEFAULT TO` Operation

```
# Default Graph

# Graph http://examplenamedgraph.com

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

<http://examplegraph.com/Nathan> a foaf:Person .
<http://examplegraph.com/Nathan> foaf:givenName "Nathan" .
<http://examplegraph.com/Nathan> foaf:mbox  <mailto:nathan@examplegraph.com> .
```

RDF statements can be inserted from an input graph to a destination graph, using the `ADD` operation. Listing 7-32 shows sample RDF triples to be added from the default graph to a named graph.

***Listing 7-32.*** Data Before the ADD Operation

```
# Default graph
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

<http://examplegraph.com/Michael> a foaf:Person .
<http://examplegraph.com/Michael> foaf:givenName "Michael" .
<http://examplegraph.com/Michael> foaf:mbox  <mailto:mike@examplegraph.com> .
```

```
# Graph http://examplenamedgraph.com
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

<http://examplenamedgraph.com/Jemma> a foaf:Person .
```

The ADD operation in Listing 7-33 performs the task.

***Listing 7-33.*** An ADD Operation in SPARQL 1.1

```
ADD DEFAULT TO <http://examplenamedgraph.com>
```

As a result, the default graph is merged to the named graph (see Listing 7-34).

***Listing 7-34.*** The Result of an ADD Operation

```
# Default graph
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

<http://examplegraph.com/Michael> a foaf:Person .
<http://examplegraph.com/Michael> foaf:givenName "Michael" .
<http://examplegraph.com/Michael> foaf:mbox  <mailto:mike@examplegraph.com> .

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

<http://examplenamedgraph.com/Jemma> a foaf:Person .

<http://examplegraph.com/Michael> a foaf:Person .
<http://examplegraph.com/Michael> foaf:givenName "Michael" .
<http://examplegraph.com/Michael> foaf:mbox <mailto:mike@example> .
```

# Proprietary Query Engines and Query Languages

While most Semantic Web platforms and graph databases have SPARQL support through a SPARQL query engine such as Apache Jena's ARQ or AllegroGraph's sparql-1.1, some vendors provide their own query language. Many of these query languages are partially compatible with, or similar to, SPARQL but are often incompatible with other Semantic Web software products beyond the ones they are released with.

## SeRQL: The Sesame RDF Query Language

Sesame supports not only SPARQL but also *SeRQL* (pronounced "circle"), the *Sesame RDF Query Language* [13]. Regardless of whether you have a connection to a local or remote Sesame repository, you can perform a SeRQL query on a `SesameRepository` object, retrieve the result as a table, and display the values (see Listing 7-35).

***Listing 7-35.*** A SeRQL Query, Using the Sesame Repository API

```
String query = "SELECT * FROM {x} p {y}";
QueryResultsTable resultsTable = myRepository.performTableQuery(QueryLanguage.SERQL, query);

int rowCount = resultsTable.getRowCount();
int columnCount = resultsTable.getColumnCount();
```

```
for (int row = 0; row < rowCount; row++) {
    for (int column = 0; column < columnCount; column++) {
        Value value = resultsTable.getValue(row, column);

        if (value != null) {
            System.out.print(value.toString());
        }
        else {
            System.out.print("null");
        }

        System.out.print("\t");
    }

    System.out.println();
}
```

---

■ **Note** Some repository operations require elevated privileges, so you might have to log on to the `SesameService` before obtaining the repository object. For instance, if you do not have read access to a repository, you will get an `AccessDeniedException`.

---

If you have to change a lot of RDF triples in your repository, you can use the Sesame Graph API in combination with SeRQL `CONSTRUCT` queries. For instance, if we have a repository that describes the manuscript of a book with many triples, and the book has been published since the last update of the repository, there are many obsolete property values (`draft`) for the `PublicationStatus` property from the Publishing Status Ontology (PSO) that should be changed to `published`. Rather than changing the property values manually for each affected triple, we derive the new triples from the existing `PublicationStatus` statements, changing the object of these statements from `draft` to `published` (see Listing 7-36).

***Listing 7-36.*** Changing Multiple Property Values, Using the Sesame Graph API

```
myRepository.addGraph(QueryLanguage.SERQL, ➥
"CONSTRUCT {X} <http://purl.org/spar/pso/PublicationStatus> {\"published\"} " + ➥
"FROM {X} <http://purl.org/spar/pso/PublicationStatus> {\"draft\"}");
```

Now all the triples are updated with the new property value; however, these triples are duplicated. The original triples with the obsolete property value have to be removed from the graph. To do so, we select all `PublicationStatus` triples with the object `draft` (the old value) and remove these triples from the repository (see Listing 7-37).

***Listing 7-37.*** Removing All Triples with the Obsolete Property Value

```
myRepository.removeGraph(QueryLanguage.SERQL, ➥
"CONSTRUCT * " + "FROM {X} <http://purl.org/spar/pso/PublicationStatus> {\"draft\"}";
```

---

■ **Note** This workaround is needed only because SeRQL does not support update operations. Another option to update a Sesame repository is using the SAIL API.

---

# CQL: Neo4j's Query Language

Neo4j has a proprietary query language called *Cypher Query Language* (*CQL*), a declarative pattern-matching language with an SQL-like, very simple, and human-readable syntax [14]. The most frequently used Neo4j CQL commands and clauses are `CREATE` (to create nodes, relationships, and properties), `MATCH` (to retrieve data about nodes, relationships, and properties), `RETURN` (to return query results), `WHERE` (to provide conditions to filter retrieval data), `DELETE` (to delete nodes and relationships), `REMOVE` (to delete properties of nodes and relationships), `ORDER BY` (to sort retrieval data), and `SET` (to add or update labels). The most frequently used Neo4j CQL functions are `String` (to work with String literals), `Aggregation` (to perform some aggregation operations on CQL Query results), and `Relationship` (to get details of Relationships such as `startnode` and `endnode`). The data types of Neo4j CQL are similar to the Java programming language. The datatypes are used to define properties of nodes and relationships, such as `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `string`.

The `MATCH` command identifies a node through the node name and the node label. The `MATCH` command expects the node name and the label name as the arguments in curly brackets separated by a colon (see Listing 7-38).

***Listing 7-38.*** `MATCH` Command Syntax

```
MATCH
(
  node-name:label-name
)
```

The `RETURN` clause is used in CQL, with the `MATCH` command, to retrieve data about nodes, relationships, and properties from a Neo4j graph database. The `RETURN` clause can retrieve some or all properties of a node and retrieve some or all properties of nodes and associated relationships. The arguments of the `RETURN` clause are the node name(s) and property name(s) (see Listing 7-39).

***Listing 7-39.*** `RETURN` Clause Syntax

```
RETURN
  node-name.property1-name, … node-name. propertyn-name
```

For example, to retrieve all property data of the `Fac` node representing a university's faculties, you can combine the `MATCH` command with the `RETURN` clause, as shown in Listing 7-40.

***Listing 7-40.*** Retrieving All Faculty Data

```
MATCH (fac: Fac)
RETURN fac.facno,fac.fname,fac.location
```

The number of rows returned by the command will be identical to the number of faculties of the university stored in the database.

Similar to SPARQL, Neo4j's CQL uses the `WHERE` clause to get the desired data (see Listing 7-41). Rather than using the `WHERE` clause with the `SELECT` command, however, in CQL, you use it with `SELECT`'s CQL equivalent, `MATCH`.

***Listing 7-41.*** The Syntax of the `WHERE` Clause in CQL

```
WHERE condition boolean_operator additional_condition
```

The first argument provides the condition, consisting of a property name, a comparison operator, and a value. The property name is the name of a graph node or the name of a relationship. The comparison operator is one of = (equal to), <> (not equal to), < (less than), > (greater than), <= (less than or equal to), or >= (greater than or equal to). The value is a literal value, such as a number, a string literal, etc. The second and third arguments (the Boolean operator and multiple conditions) are optional. The Boolean operator can be `AND`, `OR`, `NOT`, or `XOR`.

To sort rows in ascending order or descending order, use the `ORDER BY` clause with the `MATCH` command (similar to SPARQL's `ORDER BY` on the `SELECT` queries). The `ORDER BY` clause contains the list of properties used in sorting, optionally followed by the `DESC` keyword (when sorting in descending order).

# Identify Datasets to Query

To access data from LOD datasets, you can perform a semantic search, browse dataset catalogs, or run queries directly from a dedicated query interface. For searching machine-readable data, you can use *semantic search engines* such as Sindice (`http://sindice.com`) or FactForge (`http://factforge.net`). Third-party *data marketplaces* such as `http://datamarket.com` can find open data from secondary data sources and consume or acquire data for data seekers.

To retrieve information from datasets, you can run SPARQL queries on purpose-built access points called *SPARQL endpoints* (as mentioned earlier) that usually provide a web interface and, optionally, an API. The automatic discovery of the SPARQL endpoint for a given resource is not trivial; however, *dataset catalogs* such as `http://datahub.io` and `http://dataportals.org` can be queried for a SPARQL endpoint with a given URI. Because a prerequisite for all LOD datasets to be added to the LOD Cloud Diagram is to provide a dedicated SPARQL endpoint, Datahub registries often include a SPARQL endpoint URL.

Another approach for identifying SPARQL endpoints is using the VoID standardized vocabulary, which is specifically designed for describing datasets. In VoID files, the descriptions are provided as URLs, which can be canonically derived from a URI.

# Public SPARQL Endpoints

Many SPARQL endpoints are publicly available and typically have a default LOD dataset set for querying. Your SPARQL queries will run on the default graph of the endpoint, unless you refer to named graphs in your queries. For example, DBpedia's SPARQL endpoint is `http://dbpedia.org/sparql/`, which runs queries on DBpedia by default.

---

■ **Note** DBpedia offers two other interfaces to its SPARQL endpoint. The first is called *SPARQL Explorer* and is available at `http://dbpedia.org/snorql/`. The second is the *DBpedia Query Builder* available at `http://querybuilder.dbpedia.org`, which can be used to build your own queries. Because the dataset is the same in each case, the SPARQL query results are identical on all three interfaces.

---

SPARQL endpoints dedicated to a particular dataset can be domain-specific. Since the Web of Data is highly distributed, there is no SPARQL endpoint to query the entire Semantic Web (like Google searches on the conventional Web). However, the most frequently used public SPARQL endpoints can query extremely large datasets containing millions or even billions of triples, which are suitable to answer complex questions (see Table 7-2).

***Table 7-2.*** *Popular Public SPARQL Endpoints*

| Service/Dataset | SPARQL Endpoint |
|---|---|
| Datahub/CKAN | http://semantic.ckan.net/sparql |
| DBpedia | http://dbpedia.org/sparql/ |
| GeoNames | http://geosparql.org/ |
| Linked Open Commerce | http://linkedopencommerce.com/sparql/ |
| Linked Open Data Cloud | http://lod.openlinksw.com/sparql |
| LinkedGeoData | http://linkedgeodata.org/sparql |
| Sindice | http://sparql.sindice.com/ |
| URIBurner | http://uriburner.com/sparql |

# Setting Up Your Own SPARQL Endpoint

If you publish your LOD dataset on your server, you might want to set up a dedicated SPARQL endpoint to provide easy access to it. There are a couple of free, open source, and commercial products available, not all of which have a full SPARQL 1.1 support, but most have a complete SPARQL 1.0 support. Some products are standalone SPARQL endpoints that you can install on your web server, while others are more comprehensive products that provide a SPARQL endpoint as a feature. The most widely deployed SPARQL endpoints are OpenLink Virtuoso, Fuseki, D2R, 4store SPARQL Server, and PublishMyData.

## OpenLink Virtuoso

*OpenLink Virtuoso* is by far the most widely deployed SPARQL endpoint. Among others, Virtuoso is implemented as the SPARQL endpoint for DBpedia and DBpedia Live, LinkedGeoData, Sindice, the BBC, BioGateway, data.gov, CKAN, and the LOD Cloud Cache.

The Virtuoso SPARQL Query Editor provides the default LOD dataset associated with a particular installation, which can be overridden when querying named graphs. For example, the default dataset of the DBpedia SPARQL endpoint is `http://dbpedia.org`, as shown in Figure 7-2. This is obviously different in each installation of Virtuoso, but the interface is usually very similar, if not identical.

The `Query Text` is a multiline text area in which you can write your SPARQL queries. This textarea usually contains a skeleton query, which you can easily modify by overwriting, removing, or adding SPARQL code. Under the textarea, you can select the output format, optionally the maximum time after which the query execution will stop, and the rigorous check of the query. Some installations offer a set of sample queries as a drop-down list. You typically have two buttons as well: one to run the query you wrote (`Run Query`) and another to clear the textarea (`Reset`).

The output format drop-down might vary somewhat from installation to installation, but generally, you have options such as HTML, Spreadsheet, XML, JSON, JavaScript, Turtle, RDF/XML, N-Triples, CSV, TSV, and CXML. Some of the output formats might not be available, due to configuration or missing components. For example, the CXML data exchange format suitable for faceted view, which can be displayed by programs such as Microsoft Pivot, require the Virtuoso Universal Server (Virtuoso Open Source does not contain some required functions), the ImageMagick plug-in, the QRcode plug-in (before version 0.6; after version 0.6 it is optional), and the sparql_cxml VAD package to be installed, in order to get this option.

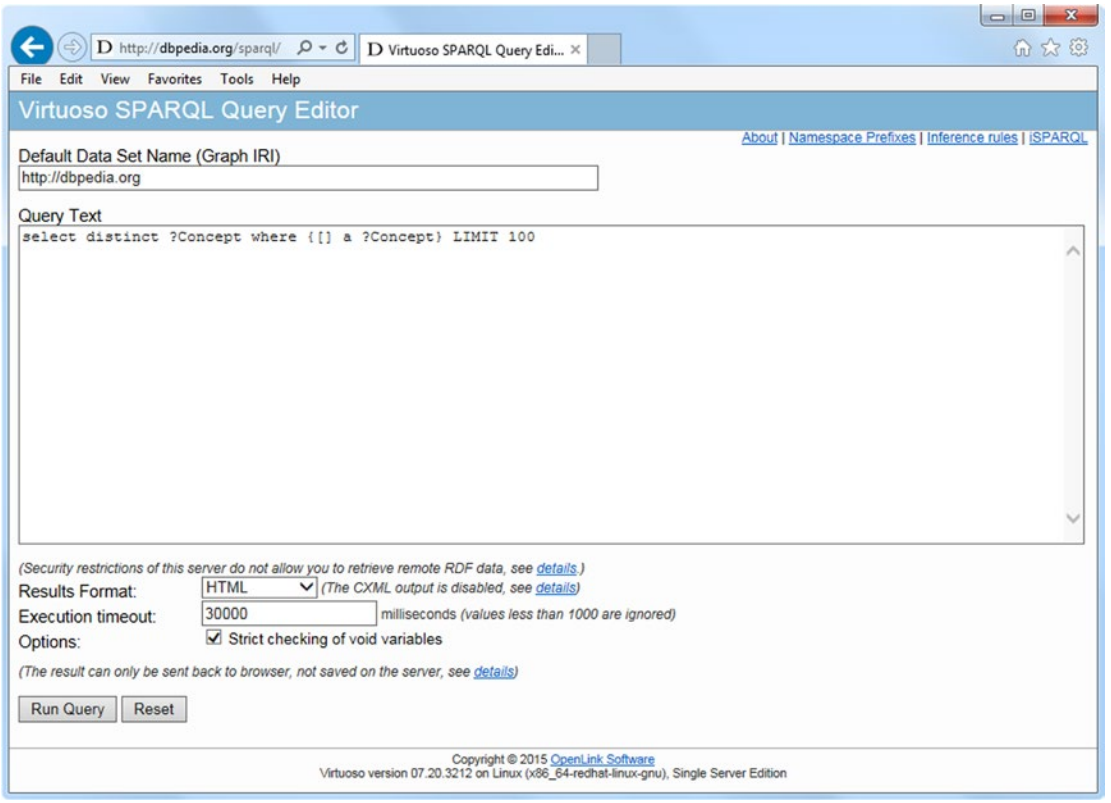


***Figure 7-2.*** *A Virtuoso SPARQL endpoint*

To install the OpenLink Virtuoso SPARQL Endpoint, follow these steps:

1. Download Virtuoso Open Source from `http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VOSDownload` or the commercial edition of Virtuoso from `http://virtuoso.openlinksw.com/download/`.
2. For the commercial Windows Open Source Edition, run the installer; otherwise, create a build.
3. Verify the installation and the configuration of the environmental variables by running the `virtuoso -?` Command.
4. Start the Virtuoso server with `virtuoso-start.sh`.
5. Verify the connection to the Virtuoso Server, using `isql localhost` (if using the default DB settings), `isql localhost:1112` (assuming demo database), or visit `http://<virtuoso-server-host-name>:[port]/conductor` in your browser
6. Open the SPARQL endpoint at `http://<virtuoso-server-host-name>:[port]/sparql`.
7. Run a test query such as `SELECT DISTINCT * WHERE {?s ?p ?o} LIMIT 50`.

# Fuseki

*Fuseki* is Apache Jena's SPARQL server that provides REST-style SPARQL HTTP Update, SPARQL Query, and SPARQL Update, using the SPARQL protocol over HTTP.

1. Download the binary distribution from https://jena.apache.org/download/.
2. Unzip the file.
3. Set file permission using the `chmod +x fuseki-server s-*` command.
4. Run the server by executing the command `fuseki-server --update --mem /ds`, which creates an in-memory, non-persistent dataset. If you want to create an empty, in-memory (non-persistent) dataset and load a file into it, use `--file=FILE` instead of `--mem`.

The default port number of Fuseki is 3030, which can be overridden by the port argument in the form `--port=number`. Fuseki not only supports SPARQL 1.1 queries, SPARQL 1.1 update operations, and file upload to a selected dataset but also provides validators for SPARQL Query and SPARQL Update, as well as for RDF serializations. To open the control panel of Fuseki, visit `http://localhost:3030` in your browser, click `Control Panel`, and select the dataset.

The URI scheme of the Fuseki server consists of the host, followed by the dataset and the endpoint, all of which are separated by a slash.

- `http://host/dataset/query` (SPARQL query endpoint)
- `http://host/dataset/update` (SPARQL UPDATE endpoint)
- `http://host/dataset/data` (SPARQL Graph Store Protocol endpoint)
- `http://host/dataset/upload` (file upload endpoint)

To load some RDF data into the default graph of the server, use the `s-put` command, as shown in Listing 7-42.

***Listing 7-42.*** Load RDF Data into the Default Graph of Fuseki

```
s-put http://localhost:3030/ds/data default books.ttl
```

To retrieve data from the default graph of the server, use the `s-get` command (see Listing 7-43).

***Listing 7-43.*** Retrieving Data, Using `s-get`

```
s-get http://localhost:3030/ds/data default
```

The default graph of the server can be queried with SPARQL, using the …/`query` endpoint employing the `s-query` command, as demonstrated in Listing 7-44.

***Listing 7-44.*** SPARQL Querying with Fuseki

```
s-query --service http://localhost:3030/ds/query 'SELECT * {?s ?p ?o}'
```

A SPARQL `UPDATE` query can be executed using the …/`update` endpoint with `s-update`. As an example, let's clear the default graph, as shown in Listing 7-45.

***Listing 7-45.*** A SPARQL `UPDATE` Query with Fuseki

```
s-update --service http://localhost:3030/ds/update 'CLEAR DEFAULT'
```

To use SPARQL 1.1 Query from Java applications, you can use the `QueryExecutionFactory.sparqlService` of Apache Jena's SPARQL query engine, ARQ. For the programmatic access of SPARQL Update, use `UpdateExecutionFactory.createRemote`. SPARQL HTTP can be used through `DatasetAccessor`.

## D2R

The *D2R Server* is a tool for publishing relational databases as Linked Data, providing access to the database content through a browser interface and querying the database using SPARQL. D2R performs on-the-fly transformation of SPARQL queries to SQL queries via a mapping. Among others, D2R is used as the SPARQL endpoint of Dailymed, a comprehensive, up-to-date dataset of marketed drugs in the United States. The D2R Server can be installed as follows:

1. Download the server from `http://d2rq.org`.
2. Run the server in one of the following ways:

   - From the command line (for development or testing), with the syntax shown in Listing 7-46.

     ***Listing 7-46.*** Running D2R From the Command Line

     ```
     d2r-server [--port port] [-b serverBaseURI][--fast] [--verbose]
     [--debug] mapping-file.ttl
     ```

     Because the default port number is `2020`, the default server URI is `http://localhost:2020`. The `fast` argument can optionally be used for performance optimization, the `verbose` argument for detailed logging, and `debug` for full logging. Optionally, you can declare the name of the D2RQ mapping file to use. If the mapping file is not provided, the database connection must be specified on the command line, so that a default mapping will be used.

   - Deploy the D2R Server web application into a servlet container, such as Apache Tomcat or Jetty (for production).

     a) Ensure that the mapping file includes a configuration block, setting the base URI in the form `http://servername/webappname/`. The `d2r:Server` instance in this file configures the D2R server (see Listing 7-47).

     ***Listing 7-47.*** D2R Configuration File Example

     ```
     @prefix d2r: <http://example.com/d2r-server/config.rdf#> .
     @prefix meta: <http://exampe.com/d2r-server/metadata#> .

     <> a d2r:Server;
       rdfs:label "My D2R Server";
       d2r:baseURI <http://localhost:2020/>;
       d2r:port 2020;
       d2r:vocabularyIncludeInstances true;

       d2r:sparqlTimeout 300;
       d2r:pageTimeout 5;
     ```

```
meta:datasetTitle "My Dataset" ;
meta:datasetDescription "This dataset contains Semantic Web ↩
 publication resources." ;
meta:datasetSource "The dataset covers publications from all related ↩
 datasets such as XY." ;

meta:operatorName "John Smith" ;
.
```

The `d2r:Server` instance supports a variety of configuration properties. The human-readable server name can be provided using `rdfs:label`. The base URI of the server can be declared using `d2r:baseURI` (the equivalent of the `-b` command line parameter). The port number of the server can be added as `d2r:port` (same as `--port` in the command line). By default, the RDF and HTML representations of vocabulary classes are also list instances, and the property representations are also list triples using the property. The `d2r:vocabularyIncludeInstances` configuration property accepts the `false` Boolean value to override this behavior. To specify automatic detection of mapping file changes, one can use the `d2r:autoReloadMapping` property. The default value is `true`. The maximum number of entities per class map can be set using `d2r:limitPerClassMap`. The default value is `50`, and the limit can be disabled with the property value set to `false`. The maximum number of values from each property bridge can be configured using `d2r:limitPerPropertyBridge`. The default value is `50`, and the limit can be disabled with the property value set to `false`. The timeout of the D2R server's SPARQL endpoint can be set in seconds as the property value of the `d2r:sparqlTimeout` property. If you want to disable the timeout for the SPARQL endpoint, set the value to `0`. The timeout for generating resource description pages can be similarly set in seconds, using the `d2r:pageTimeout`, which can also be disabled by setting the value to `0`. The default resource metadata template can be overridden using `d2r:metadataTemplate`, which specifies a literal value for the path name, either absolute or relative to the location of the server configuration file. The default dataset metadata template can be overridden by the value of `d2r:datasetMetadataTemplate`. The `d2r:disableMetadata` property enables the automatic creation and publication of all dataset and resource metadata, which accepts a Boolean value. The `true` value is assumed if the `d2r:disableMetadata` property is omitted.

b) The name of the configuration file declared as the `configFile` param in `/webapp/WEB-INF/web.xml` has to be changed to the name of your configuration file. The recommended location of the mapping file is the `/webapp/WEB-INF/` directory.

c) In the main directory of the D2R server, run ant war, which creates the `d2rq.war` file (requires Apache Ant).

d) The name of your web application can optionally be changed by renaming the file to `webappname.war`.

e) Deploy the `.war` file into your servlet container, such as by copying it into the `webapps` directory of Tomcat.

## 4store SPARQL Server

*4store* provides a SPARQL HTTP protocol server, which can answer SPARQL queries using the SPARQL HTTP query protocol. To run 4store's SPARQL server, use the `4s-httpd` command with the port number and KB name as shown in Listing 7-48.

***Listing 7-48.*** Running 4store's HTTP Server

```
4s-httpd -p port_number 4store_KB_name
```

---

■ **Note** Multiple 4store KBs have to run on separate ports.

---

Once the server is running, the overview page can be accessed in the web browser at `http://localhost:port_number/status/`, and the SPARQL endpoint at `http://localhost:port_number/sparql/` with an HTML interface at `http://localhost:port_number/test/`. From the command line, you can query the SPARQL server using the sparql-query tool available at https://github.com/tialaramex/sparql-query.

## PublishMyData

*PublishMyData* is a commercial Linked Data publishing platform. Because it is a *Software as a Service* (SaaS) in the cloud, you don't have to install anything to use it. Beyond the SPARQL endpoint, PublishMyData provides RDF data hosting, a Linked Data API, and customizable visualizations. It supports SPARQL 1.1. To submit a SPARQL query from your code, issue an HTTP GET request to the SPARQL endpoint, as demonstrated in Listing 7-49.

***Listing 7-49.*** SPARQL Query on PublishMyData

```
http://example.com/sparql?query=URL-encoded_query
```

For instance, to run the query `SELECT * WHERE {?s ?p ?o} LIMIT 10` and get the results in JSON, the URL to be used will have the structure shown in Listing 7-50.

***Listing 7-50.*** URL-Encoded SPARQL Query with PublishMyData

```
http://example.com/sparql.json?query=SELECT+%2A+WHERE+%7B%3Fs+%3Fp+%3Fo%7D+LIMIT+10
```

For demonstrating programmatic access, let's use JavaScript to request data from the SPARQL endpoint (see Listing 7-51).

***Listing 7-51.*** Using jQuery to Request Data Through the SPARQL Endpoint

```
<!DOCTYPE html>
<html>
  <head>
    <script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
  </head>
  <body>
    <script type="text/javascript">
      var siteDomain = "example.com";
      var query = "SELECT * WHERE {?s ?p ?o} LIMIT 10";
```

```
      var url = "http://" + siteDomain + "/sparql.json?query=";
      url += encodeURIComponent(query);
      $.ajax({
        dataType: 'json',
        url: url,
        success: function(data) {
        alert('success: ' + data.results.bindings.length + ' results');
        console.log(data);
        }
      });
    </script>
  </body>
</html>
```

When requesting the SPARQL output as JSON, a callback parameter can be passed, so that the results will be wrapped in the function, which can prevent cross-domain issues when running JavaScript under older browsers (see Listing 7-52).

***Listing 7-52.*** Using a Callback Function

```
http://example.com/sparql.json?callback=myCallbackFunction&query=SELECT+%2A+WHERE+%7B%3Fs ↩
+%3Fp+%3Fo%7D+LIMIT+10
```

Alternatively, you can make a JSON-P request with jQuery and can omit the callback parameter from the URL by setting the `dataType` to `jsonp`, as shown in Listing 7-53.

***Listing 7-53.*** Using JSON-P for SPARQL Querying

```
queryUrl = 'example.com/sparql.json?query=SELECT+%2A+WHERE+%7B%3Fs+%3Fp+%3Fo%7D+LIMIT+10'

$.ajax({
  dataType: 'jsonp',
  url: queryUrl,
  success: function(data) {
  // callback code
  alert('success!');
  }
});
```

You can also use Ruby to request data from the PublishMyData SPARQL endpoint, as shown in Listing 7-54.

***Listing 7-54.*** Make a Request to the PublishMyData SPARQL Endpoint in Ruby

```
require 'rest-client'
require 'json'

query = 'SELECT * WHERE {?s ?p ?o} LIMIT 10'
site_domain = "example.com"
url = "http://\#example.com/sparql.json"
```

```
results_str = RestClient.get url, {:params => {:query => query}}
results_hash = JSON.parse results_str
results_array = results_hash["results"]["bindings"]

puts "Total number of results: \#{results_array.length}"
```

The request in this case is written as JSON, and the result will be put in a Hash table.

# Summary

In this chapter, you learned the foundations of SPARQL, the standardized query language of RDF. You are now familiar with the query types and know how to write SPARQL queries to answer complex questions, display all nodes of an RDF graph with a particular feature, filter results, or add new triples to a dataset. By now you also recognize the most popular SPARQL endpoint interfaces and know how to set up your own endpoint.

The next chapter will show you how to handle high-volume, high-velocity datasets, leverage Semantic Web technologies in Big Data applications, and add structured data to your site, so that it will be considered for inclusion in the Google Knowledge Graph.

# References


1. The W3C SPARQL Working Group (2013) SPARQL 1.1 Overview. W3C Recommendation. World Wide Web Consortium. www.w3.org/TR/sparql11-overview/. Accessed 6 March 2015.
2. Prud'hommeaux, E., Seaborne, A. (2008) SPARQL Query Language for RDF. www.w3.org/TR/rdf-sparql-query/. Accessed 18 April 2015.
3. Harris, S., Seaborne, A. (2013) www.w3.org/TR/sparql11-query/. Accessed 18 April 2015.
4. Gearon, P., Passant, A., Polleres, A. (eds.) (2013) SPARQL 1.1 Update. www.w3.org/TR/sparql11-update/. Accessed 18 April 2015.
5. Ogbuji, C. (ed.) (2013) SPARQL 1.1 Graph Store HTTP Protocol. W3C Recommendation. World Wide Web Consortium. www.w3.org/TR/sparql11-http-rdf-update/. Accessed 6 March 2015.
6. Williams, G. T. (ed.) SPARQL 1.1 Service Description. www.w3.org/TR/sparql11-service-description/. Accessed 18 April 2015.
7. Glimm, B., Ogbuji, C. (eds.) (2013) SPARQL 1.1 Entailment Regimes. www.w3.org/TR/sparql11-entailment/. Accessed 18 April 2015.
8. Seaborne, A. (ed.) (2013) SPARQL 1.1 Query Results JSON Format. www.w3.org/TR/sparql11-results-json/. Accessed 18 April 2015.
9. Seaborne, A. (ed.) (2013) SPARQL 1.1 Query Results CSV and TSV Formats. www.w3.org/TR/sparql11-results-csv-tsv/. Accessed 18 April 2015.
10. Hawke, S. (ed.) (2013) SPARQL Query Results XML Format (Second Edition). www.w3.org/TR/rdf-sparql-XMLres/. Accessed 18 April 2015.

11. Prud'hommeaux, E., Buil-Aranda, C. (eds.) (2013) SPARQL 1.1 Federated Query. www.w3.org/TR/sparql11-federated-query/. Accessed 18 April 2015.

12. Feigenbaum, L., Williams, G. T., Clark, K. G., Torres, E. (eds.) (2013) SPARQL 1.1 Protocol. W3C Recommendation. World Wide Web Consortium. www.w3.org/TR/sparql11-protocol/. Accessed 9 March 2015.

13. Broekstra, J., Ansell, P., Visser, D., Leigh, J., Kampman, A., Schwarte, A. et al. (2015) The SeRQL query language. http://rdf4j.org/sesame/2.7/docs/users.docbook?view#chapter-serql. Accessed 22 April 2015.

14. Neo Technology, Inc. (2015) Intro to Cypher. http://neo4j.com/developer/cypher-query-language/. Accessed 22 April 2015.