4 Semantic Web Application Architecture

So far, we have seen how RDF can represent data in a distributed way across the web. As such, it forms the basis for the semantic web, a web of data in which Anyone can say Anything about Any topic. The focus of this book is modeling on the semantic web; describing and defining distributed data in such a way that the data can be brought back together in a useful and meaningful way. In a book just about modeling, one could say that there is no room for a discussion of system architecture – the components of a computer system that can actually use these models in useful applications. But this book is for the Working Ontologist, who builds models so that they can be used. Used for what? For building some application that takes advantage of information distributed over the web. In short, to put the Semantic Web to work we need to describe, at least at a high level, the structure of a Semantic Web application. In particular, the components it is made of, what kinds of inputs it gets (and from where), how it takes advantage of RDF, and why this is different from more familiar application architectures.

Many of the components of a Semantic Web application are provided both as supported products by companies specializing in Semantic Web technology, as well as by free software under a variety of licenses. New software is being developed both by research groups as well as product companies on an ongoing basis. We will not describe any particular tools in this chapter, but rather describe the types of components that make up a Semantic Web deployment, and how they fit together.

RDF Parser/Serializer: In section **Error! Reference source not found.**, we saw a number of serializations of RDF, including the W3C standard serialization in XML. An RDF Parser reads text in one (or more) of these formats, and interprets it as triples in the

RDF data model. An RDF serializer does it backwards; it takes a set of triples, and creates a file that expresses that content in one of the serialization forms

RDF Store. In section **Error! Reference source not found.**, we saw how RDF distributes data in the form of *triples*. An *RDF Store* (sometimes called a *triple store*) is a database that is tuned for storing and retrieving data in the form of triples. In addition to the familiar functions of any database, an RDF store has the additional ability to merge information from multiple data sources, as enabled by the RDF standard.

RDF Query Engine. Closely related to the RDF store is the RDF Query engine. The query engine provides the capability to retrieve information from an RDF store according to structured queries.

Application code. An application has some work that it does with the data is process; analysis, user interaction, archiving, etc. These capabilities are accomplished using some programming language that accesses the RDF store via queries (processed with the RDF Query Engine).

Most of these components have corresponding components in a familiar relational data-backed application. The relational database iteself corresponds to the RDF store, in that it stores the data. The database includes a query language that corresponds to the query engine for accessing this data. In both cases, the application itself is written using a general-purpose programming language that makes queries and processes their results. The parser/serializer has no direct counterpart in a relational data-backed system, at least as far as standards go. There is no standard serialization of a relational data base that will allow it to be imported into a competing relational data base system without change of semantics.

In the following sections, we will examine each of these capabilities in detail. Since new products in each of these categories are being developed on a regular basis, we will describe them generically, and not refer to specific products.

4.1 RDF Parser/Serializer

How does an RDF-based system get started? Where do the triples come from? There are a number of possible answers for this, but the simplest one is to find them directly on the web.

At the time of this writing, Google was able to find millions of files with extension ".rdf". Any of these could be a source of data for an RDF application. But these files are useless, unless we have a program that can read them. That program is an RDF parser.

RDF parsers take as their input a file in some RDF format. Most parsers support the standard RDF/XML format, which is compatible with the more widespread XML standard. An RDF Parser takes such a file as input, and converts it into an internal representation of the triples that are expressed in that file. At this point, the triples are stored in the triple store, and are available for all the operations of that store.

The triples at this point could also be serialized back out, either in the same text form, or another text form. This is done using the reverse operation of the parser, the serializer. It is possible to take a "round trip" with triples using a parser and serializer; if you serialize a set of triples, then parse the resulting string with a corresponding parser (e.g.., an N3 parser for an N3 serialization), then the result is the same set of triples that the process began with. Notice that this is not necessarily true if you start with a text file that represents some triples. Even in a single format, there can be many distinct files that represent the same set of triples. Thus it is not, in general, possible to read in an RDF file

and export it again, and be certain that the resulting file will be identical (character by character) to the input file.

4.1.1 Other data sources – converters and scrapers

RDF Parsers and Serializers based on the standard representations of RDF are useful for the systematic processing and archiving of data in RDF. But while there is considerable data available in these formats, even more data is not already available in RDF. Fortunately, for many common data formats (e.g., tabular data), it is quite easy to convert these formats into RDF triples.

In section 3.2XXX, we already saw how tabular data can be mapped into triples in a natural way. This approach can be applied to relational database or spreadsheets. Tools to perform a conversion based on this mapping, though not strictly speaking parsers, play the same role as a parser in a semantic solution; they connect the triple store with sources of information in the form of triples. Most RDF systems include a table input converter of some sort. Some tools specifically target relational databases, including appropriate treatment of foreign key references, while other work more directly with spreadsheet tables. Tools of this sort are called *converters*, since they typically convert information from some form into RDF, and often into a standard form of RDF like RDF/XML. This allows them to be used with any other tools that respect the RDF/XML standard.

Another rich source of data for the semantic web can be found in existing web pages, that is, in HTML pages. Such pages often include structured information, like contact information, descriptions of events, product descriptions, publications, etc. This information can be combined in novel ways on the semantic web, once it is available in RDF. There are two different approaches to the problem of making use of HTML sources

for RDF data. The first approach assumes that the original author of the HTML document might have no interest or knowledge of RDF and the semantic web, and will create content accordingly. This means that there are no annotations corresponding to predicates, no special structure of the HTML to make it especially "RDF-ready." The second approach assumes that the content author is willing to put in a bit of effort to mark up the content in such a way that in addition to its use for display as HTML, it can also include information that allows the data to be interpreted also as RDF.

Not surprisingly, the first approach received the most attention, especially as the semantic web began the bootstrapping process of gathering enough RDF data to begin the network effect. Legacy data had been represented in HTML before anyone knew anything about RDF. How could that information be made available to the semantic web as RDF triples?

The most "hands-off" approach to this problem is to use a program called a *scraper*. A scraper is a program that reads a source that was intended for human reading, typically an HTML page, and produces from it an RDF representation of that data. The name "scraper" was inspired by the image of "scraping" useful information from a complex display like a web page.

Scraper technology is continuing to develop. We will illustrate the basics with an early scraper system called Solver, which has been developed as part of the Simile project at MIT. Solvent provides a user interface for highlighting selected parts of a web page and translating the content into RDF. Figure 4-1 shows Solvent at work on a web page from the Los Angeles Metro Rail site. Solvent is implemented as a Firefox plug-in, and appears as an extra panel at the bottom of the Firefox window. Solvent provides the

basic functions of a scraper. First, it allows the user to select an item on the web page; in the figure, the user has selected the name and address of one station. The scraper then highlights all the items on the page that it determines to be “the same kind” of item; in this case, we see that Solvent has highlighted all the addresses of stations on the page in Yellow.

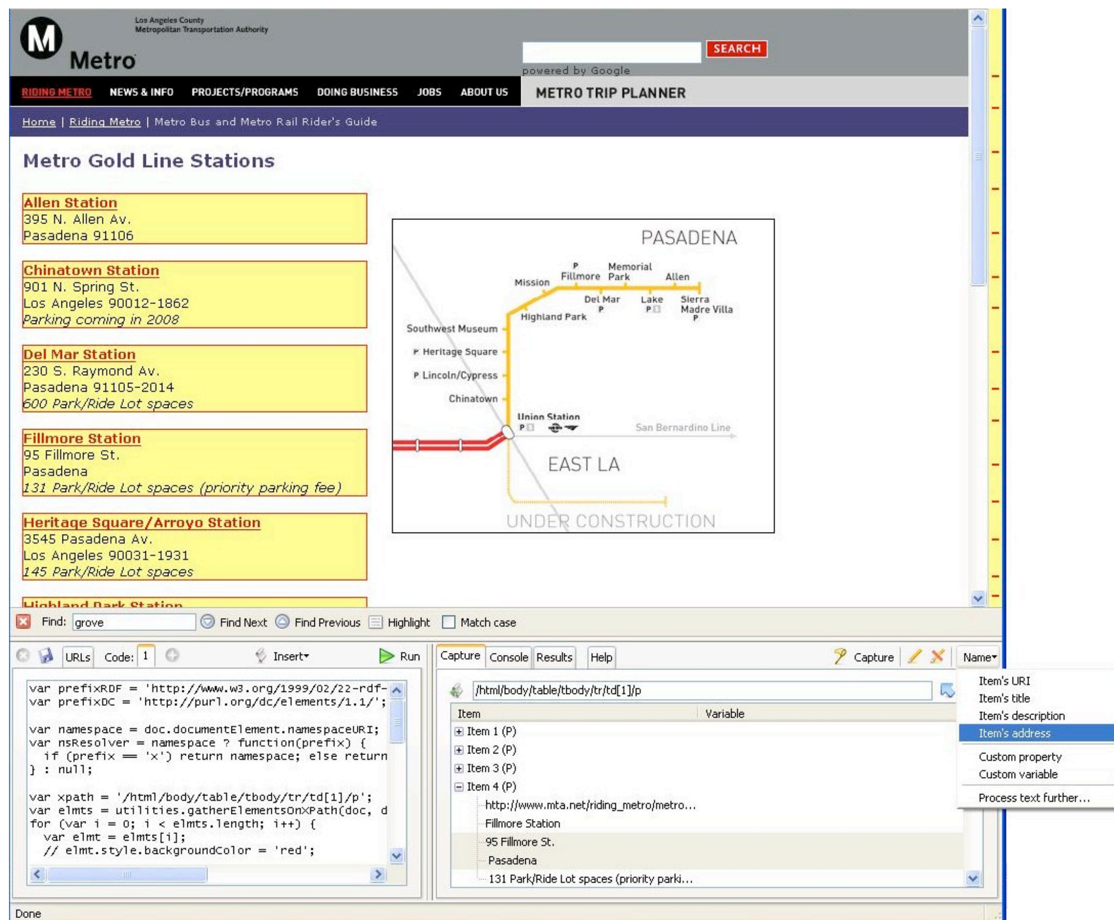

Figure 4-1 Example of the Solvent interface working with the Los Angeles Metro web page

The scraper then provides a way for the user to describe the selected data; in this example, the user specifies that “Allen Station” is the name of the first item, and that the next two lines “395 N> Allen Av. Pasadena 91106” is the address of the item. The scraper extrapolates this information to find the name and address of all the stations. The details of how the Solvent user interface does this are not important; the fundamental

ideas are that a user specifies information about a single item on a web page, and the system uses that information to mark-up all the information on the page. The result is then expressed in RDF; in this example, Solvent produces the following RDF triples:

```
metro:item0
  rdf:type metro:Metro ;
  dc:title "Allen Station" ;
  simile:address "395 N. Allen Av., Pasadena 91106" .


metro:item1
  rdf:type metro:Metro ;
  dc:title "Chinatown Station" ;
  simile:address "901 N. Spring St., Los Angeles
90012-1862" .


metro:item2
  rdf:type metro:Metro ;
  dc:title "Del Mar Station" ;
  simile:address "230 S. Raymond Av., Pasadena 91105-
2014" .

(etc.)
```

Scrapers differ in their user interfaces (for allowing users to specify items and their descriptions) and the sophistication with which they determine "similar" items on a page.

A new development in web page deployment is a trend that goes by the name of "microformats." The idea of a microformat is that some web page authors might be willing to put some structured information into their web page, to express its intended structure. To enable them to do this, a standard vocabulary (usually embedded in HTML as special tag attributes that have no impact on how a browser displays a page) is developed for commonly used items on a web page. Some of the first microformats were for business cards (including names, positions, companies and phone numbers in the controlled vocabulary) and events (including location, start time, and end time). The

growing popularity of microformats indicates that at least some web developers are willing to put in some extra effort to encode structured information into their HTML.

The W3C has outlined a specification called GRDDL (Gleaning Resource Descriptions from Dialects of Languages) that provides a standard way to express a mapping from a microformat to RDF. GRDDL makes it possible to specify, within an HTML document, a recipe for translating the HTML data into RDF resources. The transformations themselves are typically written in the XML stylesheet transformation language XSLT. Existing XHTML documents can be made available to the semantic web simply by marking up the preamble to the documents with a few simple references to transformations.

The W3C is also pursuing an alternate approach for allowing HTML authors to include semantic information in their web pages. One limitation of microformats is the need to specify a controlled vocabulary and write an XSLT script for GRDDL to use for that vocabulary. Wouldn't it be better, if instead someone (like the W3C) would simply specify a single syntax for marking up HTML pages with RDF data? Then there would be a single processing script for all microformats.

The W3C has proposed just such a format called RDFa. The idea behind RDFa is quite simple; make use of the attribute tags in HTML to embed information that can be parsed into RDF. Just like microformats, RDFa has no effect on how a browser displays a page. Current versions of RDFa are quite difficult to use. It is unclear whether the microformat approach or the RDFa approach to embedding RDF information into web pages will dominate (or indeed, if either of them will; there really is no reason for the

web page development industry to make a choice. And something else might turn out to catch on better than either of these).

All of these methods that allow web page developers to include structured information in their web pages have two advantages over scrapers and converters. First, from the point of view of the system developer, it is easier to harvest the RDF data from pages that were marked up with structure data extraction in mind. But more importantly, from the point of view of the content author, it ensures that the interpretation of the information in the document, when rendered as RDF, matches the intended meaning of the document. This really is the spirit of the word *Semantic* in the Semantic Web; that page authors be given the capability of expressing what they mean in a web page for a machine to read and use.

4.2 RDF Store

A database is a program that stores the data, making it available for future use. An RDF data storage solution is no different; the RDF data is kept in a system called an *RDF store*. It is typical for an RDF data store to be accompanied by a parser and serializer, to populate the store and publish information from the store respectively. Just as is the case for conventional (e.g., relational) data stores, an RDF store may also include a query engine as described in the next section. Conventional data stores are differentiated based on a variety of performance features, including the volume of data that can be stored, the speed with which data can be accessed or updated, and the variety of query languages supported by the query engine. These features are equally relevant when applied to an RDF store.

In contrast to a relational data store, an RDF store includes as a fundamental capability the ability to merge two data sets together. Because of the flexible nature of the RDF Data Model, the specification of such a merge operation is clearly defined. Each data store represents a set of RDF triples; a merger of two (or more) datasets is the single data set that includes all and only the triples from the source data sets. Any resources with the same URI (regardless of the originating data source) are considered to be equivalent in the merged data set. Thus, in addition to the usual means of evaluating a data store, an RDF store can be evaluated on the efficiency of the merge process.

RDF store implementations range from custom programmed database solutions to fully-supported off-the-shelf products from specialty vendors. Conceptually, the simplest relational implementation of a triple store is as a single table with three columns, one each for the Subject, Predicate and Object of the triple. The information about the Los Angeles Metro given in Section 4.1.1 would be stored in such a table as follows:

Subject	Predicate	Object
metro:item0	rdf:type	metro:Metro
metro:item0	dc:title	"Allen Station"
metro:item0	simile:address	"395 N. Allen Av., Pasadena 91106
metro:item1	rdf:type	metro:Metro
metro:item1	dc:title	"Chinatown Station"
metro:item1	simile:address	"901 N. Spring St., Los Angeles 90012-1862"
metro:item2	rdf:type	metro:Metro
metro:item2	dc:title	Del Mar Station
metro:item2	simile:address	230 S. Raymond Av., Pasadena 91105-2014"

. . .

This representation should look familiar, as it is exactly the representation we used to introduce RDF triples in Chapter 3. Since this is a relational database representation, it can be accessed using conventional relational database tools such as SQL. An experienced SQL programmer would have no problem writing a query to answer a

question like, "List the *dc:title* of every instance of *metro:Metro* in the table." As an implementation representation, it has a number of apparent problems, including the replication of information in the first column, and the difficulty of building indexes around string values like URIs. On the other hand, in situations in which SQL programming experience is plentiful, this sort of representation has been used to create a custom solution in short order.

It is not the purpose of this discussion to go into details of the possible optimizations of the RDF store. These details are the topic of the particular (often patented) solutions provided by a vendor of an off-the-shelf RDF store. In particular, the issue of building indices that work on URIs can be solved with a number of well-understood data organization algorithms. Serious providers of RDF stores differentiate their offerings based on the scalability and efficiency of these indexing solutions.

4.2.1 RDF Data Standards and Interoperability of RDF Stores

RDF stores bear considerable similarity to relational stores, especially in terms of how the quality of a store is evaluated. A notable distinction of RDF stores results from the standardization of the RDF data model and RDF/XML serialization syntax. Several competing vendors of relational data stores dominate the market today, and have for several decades. While each of these products is based on the same basic idea of the relational algebra for data representation, it is a difficult process to transfer a whole database from one system to another. That is, there is no standard serialization language with which one can completely describe a relational database, in such a way that it can be automatically imported into a competitor's system. Such a task is possible, but it typically

requires a database programmer to track down the particulars of the source database to ensure that they are represented faithfully in the target system.

The standardization effort for RDF makes the situation very different when it comes to RDF stores. Just as for relational stores, there are several competing vendors and projects. In stark contrast to the situation for relational databases, the underlying RDF Data model is shared by all of these products, and even more specifically, all of them can import and export their data sets in the RDF/XML format. This makes it a routine task to transfer an RDF data set – or indeed many RDF data sets – from one RDF store to another. This feature, which is a result of an early and aggressive standardization process, makes it much easier to begin with one RDF store, secure in the knowledge that the system can be migrated to another as the need arises. It also simplifies the issue of federating data that is housed in multiple RDF stores, possibly coming from different vendor sources.

4.3 RDF Query Engine

An RDF store may be differentiated based on its performance, but it is typically accessed using a query language. In this sense, an RDF store is similar to a relational database or an XML store. Not surprisingly, in the early days of RDF, a number of different query languages were available, each supported by some RDF based product or open-source project. From the common features of these query languages, the W3C has undertaken the process of standardizing an RDF query language called SPARQL. In this section, we will cover the highlights of the SPARQL query language. While these highlights are typical of RDF query languages in general, each query language has its own distinguishing features, some of which we expect will be incorporated in due course

into the W3C standard recommendation. We will describe SPARQL by example, based on the following set of 19 triples, shown here in N3 and in Figure 4-2 as a graph:

```
lit:Shakespeare lit:wrote lit:AsYouLikeIt ;
   lit:wrote lit:TwelfthNight;
   lit:wrote lit:KingLear;
   lit:wrote lit:LovesLaboursLost;
   lit:wrote lit:Hamlet;
   lit:wrote lit:TheTempest;
   lit:wrote lit:WintersTale;
   lit:wrote lit:HenryV;
   lit:wrote lit:MeasureForMeasure;
   lit:wrote lit:Othello ;
   bio:livedIn geo:Stratford .
bio:AnneHathaway bio:married lit:Shakespeare .
geo:Stratford geo:isIn geo:England .
geo:Scotland geo:partOf geo:UK .
geo:England geo:partOf geo:UK .
geo:Wales geo:partOf geo:UK .
geo:NorthernIreland geo:partOf geo:UK .
geo:ChannelIslands geo:partOf geo:UK .
geo:IsleOfMan geo:partOf geo:UK .
```

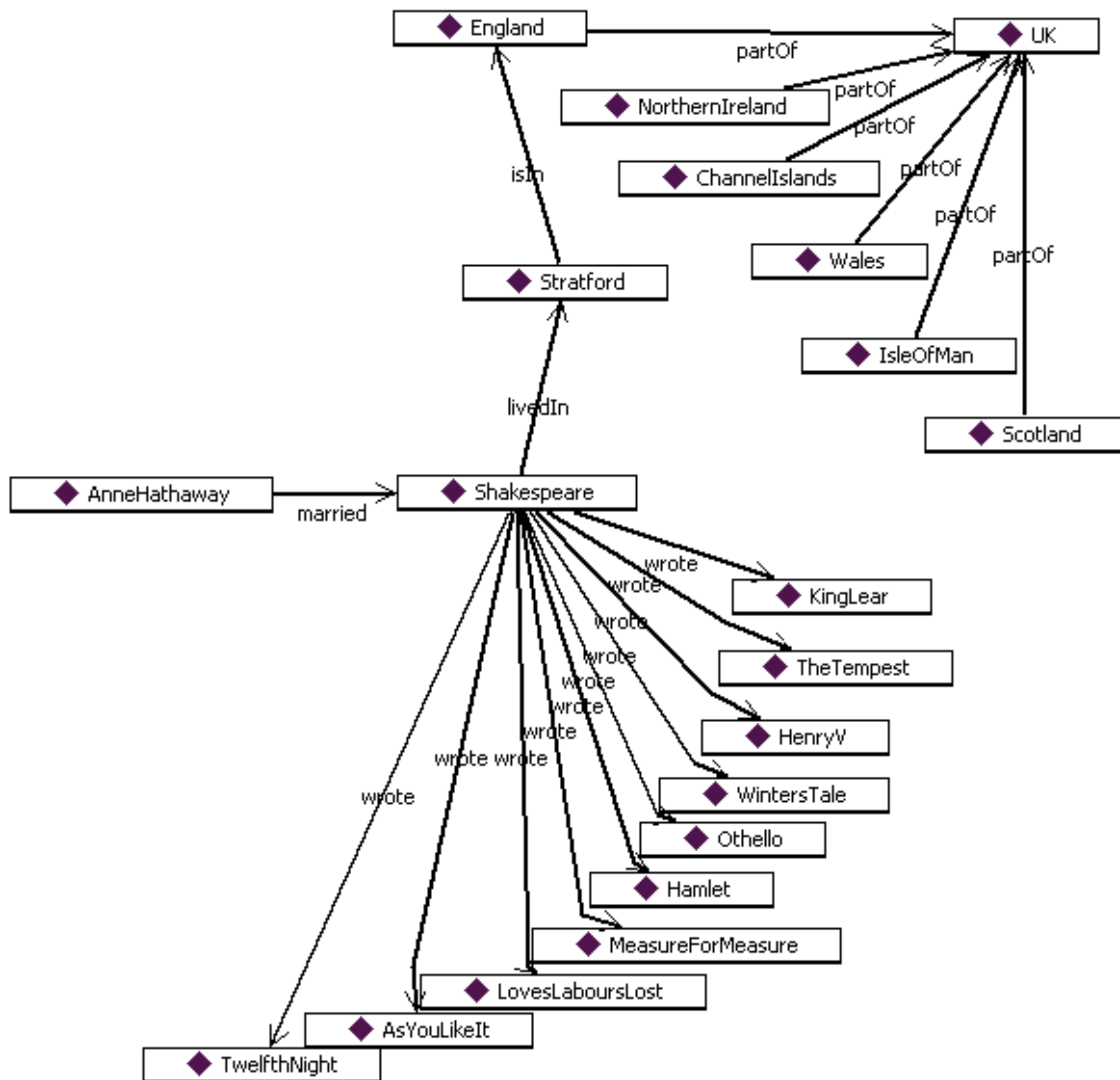

Figure 4-2 Sample triples for SPARQL examples

The basic building block of a SPARQL query is the *triple pattern*. A triple pattern looks just like a triple, but can have variables in place of resources in any of the three positions, Subject, Predicate, and Object. Variables are indicated a symbols preceded by the special character '?'. The following are all valid triple patterns:

```
?w lit:wrote lit:KingLear .
lit:Shakespeare ?r lit:KingLear .
lit:Shakespeare lit:wrote ?p .
```

The syntax for a triple pattern is intentionally very similar to the syntax for a triple in N3; a subject, predicate, object terminated by a period ("."). Each of these patterns can be interpreted as a question in a natural way, respectively:

```
Who wrote King Lear?
What relationship did Shakespeare have to King Lear?
What did Shakespeare write?
```

A SPARQL query engine, given each of these queries and the sample graph from Figure 4-2 as input, will determine the results as indicated in Table 4-1.

Triple pattern	SPARQL result
?w lit:wrote lit:KingLear .	?w = lit:Shakespeare
lit:Shakespeare ?r lit:KingLear .	?r = lit:wrote
lit:Shakespeare lit:wrote ?p .	?p = lit:AsYouLikeIt ?p = lit:TwelfthNight ?p = lit:KingLear ?p = lit:LovesLaboursLost ?p = lit:Hamlet ?p = lit:TheTempest ?p = lit:WintersTale ?p = lit:HenryV ?p = lit:MeasureForMeasure ?p = lit:Othello

Table 4-1 SPARQL results of various triple patterns on the sample input.

Since a set of RDF triples is viewed as a graph, a more interesting query is one in which the query specifies a *graph pattern*. A graph pattern is specified as a set of triple patterns, with the stipulation that any variable that appears in two or more triple patterns must match the same resource in the graph. In SPARQL syntax, graph patterns are given as a list of triple patterns, enclosed within braces ("{" and "}"). The following are valid graph patterns in SPARQL:

```
{?person lit:married ?s .
 ?person lit:wrote ?lit:KingLear . }

{?person bio:livedIn ?place .
 ?place geo:isIn geo:England .
 ?person lit:wrote lit:KingLear . }
```

Informally, these queries ask, “Find a person who married someone and who also wrote King Lear” and “Find a person who lived in a place that is in England, and who also wrote King Lear.” The meaning of a graph pattern is that *all* the triple patterns must match, and every occurrence of a single variable must match the same resource. Table 4-2 shows the results of a SPARQL query engine for each of these graph patterns on the sample input.

Graph pattern	SPARQL result
`{?person lit:married ?s .` `?person lit:wrote ?lit:KingLear .}`	no results

`{?person bio:livedIn ?place .`
`?place geo:isIn geo:England .`
`?person lit:wrote lit:KingLear . }`
	`?person=Shakespeare`
	`?place=Stratford`

Table 4-2 SPARQL results of various graph patterns on the sample input

The first result might seem a bit surprising; after all, Shakespeare wrote King Lear, and he married Anne Hathaway, right? This may well be true in the history books, but this information is not included in the sample graph. The sample graph only shows that Anne Hathaway married Shakespeare; it has no knowledge that marriage is a symmetric union, so that Shakespeare must have also married Anne Hathaway. We will see how to handle this sort of situation when we study the Web Ontology Language OWL in chapter 7XXX.

SPARQL also includes a facility for matching one triple *OR* another triple. The syntax in SPARQL is to use the keyword *UNION* to specify alternative graph patterns. We can use this facility to resolve the issue of who married whom in the previous example.

```
{ { {?spouse1 bio:married ?spouse2}
    UNION {?spouse2 bio:married ?spouse1} }
  {?spouse1 lit:wrote lit:KingLear}  }
```

The syntax gets a bit involved, but this query searches for two spouses, one has married the other (in either order), and the (arbitrarily determined) first spouse happens to have written King Lear, as shown in Table 4-3 .

Graph pattern	SPARQL result

`{{{?spouse1 bio:married ?spouse2}`
`UNION {?spouse2 bio:married ?spouse1}}`
`{?spouse1 lit:wrote lit:KingLear}}`
	`?spouse1=Shakespeare`
	`?spouse2=AnneHathaway`

Table 4-3 Graph pattern built on UNION, and its results

In these examples, we have shown the results of our SPARQL queries as binding lists, showing what value each variable is bound to. This mode of operation in SPARQL is called the *SELECT* form; that is, certain variables are selected from the graph pattern and all appropriate bindings for them are returned. In the context of an RDF store, the results of the query are returned in a more standard machine-readable form. The SPARQL standard includes the SPARQL Query Results XML Format for this purpose.

The *SELECT* form in SPARQL can be thought of as converting a graph to a table; the graph pattern matches parts of the graph, and the resulting bindings are returned as a table of values for the corresponding variables. SPARQL also supports another mode of operation called the *CONSTRUCT* form. The *CONSTRUCT* form uses two graph

patterns, and produces a new graph built from the matches in the input graph. Variable bindings in both graph patterns must match the same resources in the graph.

As an example of the use of the CONSTRUCT mode, let's consider the reification patter from Section **Error! Reference source not found.**, in which we represented the statement *Wikipedia says Shakespeare wrote Hamlet* with the triples

```
q:n1 rdf:subject lit:Shakespeare ;
     rdf:predicate lit:wrote ;
     rdf:object lit:Hamlet .
```

Then we can express the relation of Wikipedia to this statement as follows:

```
web:Wikipedia m:says q:n1 .
```

As we noted in Section **Error! Reference source not found.**, the presence of these triples does not mean that the triple

```
lit:Shakespeare lit:wrote lit:Hamlet .
```

is present, just as the statement *Wikipedia says Shakespeare wrote Hamlet* does not necessarily mean that we believe that *Shakespeare wrote Hamlet*. We can use a SPARQL construct query to pick out all of the reified statement asserted by *Wikipedia* as follows:

```
CONSTRUCT {?s ?p ?o}
WHERE {?r rdf:subject ?s .
       ?r rdf:predicate  ?p .
       ?r rdf:object ?o .
       web:Wikipedia m:says ?r .}
```

This SPARQL query will construct the graph made up of all the statements that *Wikipedia* says. This kind of query allows an application to process reified statements according to whatever policy it wants to implement; an application that trusts *Wikipedia* can use this query to add the *Wikipedia* statements into its graph. An application that does not will refrain from using this query.

An RDF Query Engine is intimately tied to the RDF store. In order to solve a query, the engine relies on the indices and internal representations of the RDF store; the more finely tuned the store is to the query engine, the better its performance. For large scale applications, it is preferable to have an RDF store and query engine that retains its performance even in the face of very large data sets. For smaller applications, other features (like cost, ease of installation, platform, and built-in integration with other enterprise systems) may dominate.

4.3.1 Comparison to Relational Queries

In many ways, an RDF query engine is very similar to the query engine in a relational data store; it provides a standard interface to the data, and defines a formalism by which data is viewed. A relational query language is based on the relational algebra of joins and foreign key references. RDF query languages look more like statements in predicate calculus. Unification variables are used to express constraints between the patterns.

A relational query describes a new data table that is formed by combining together two or more source tables. An RDF query (whether in SPARQL or another RDF query language) describes a new graph that is formed by describing a subset of a source RDF graph. That graph, in turn, may be the result of having merged together several other graphs. The inherently recursive nature of graphs simplifies a number of detailed issues that arise in table-based queries. For instance, there is no need in an RDF query language like SPARQL for a sub-query construct; the same effect can be achieved with a single query. Similarly, there is nothing special about a “self-join” in an RDF query language.

In the special case in which an RDF store is implemented as a single table in a relational data base, any graph pattern match in such a scenario will constitute a self-join on that table. Some end-developers choose to work this way in a familiar SQL environment. Oracle takes another approach to making RDF queries accessible to SQL programmers by providing its own RDF-based graph query language extension to its version of SQL, optimized for graph queries. The syntax of this language is graph-like (and hence more similar to SPARQL), but it is smoothly integrated with the table/join structure of SQL.

4.4 Application Interface

Database applications include more than just a database and query engine; they also include some application code, in an application environment, that performs some analysis on or displays some information from the database. The only access the application has to the database is through the query interface, as shown in Figure 4-3

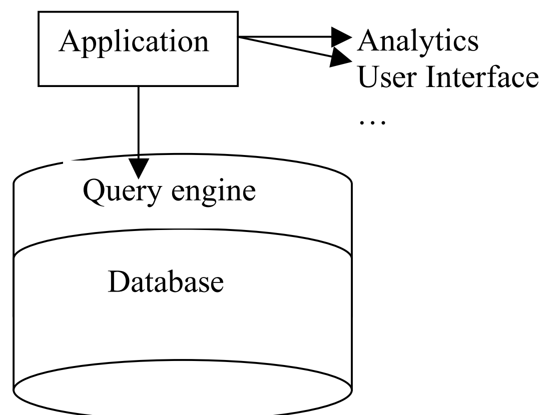

Figure 4-3 Application architecture for a database application

An RDF application has a similar architecture, but additionally includes all of the capabilities we have discussed so far:

- RDF Parser and Serializer
- Scrapers and Converters
- RDF Merge functionality
- RDF Query engine

These capabilities interact with the application itself and the RDF store as shown in Figure 4-4.

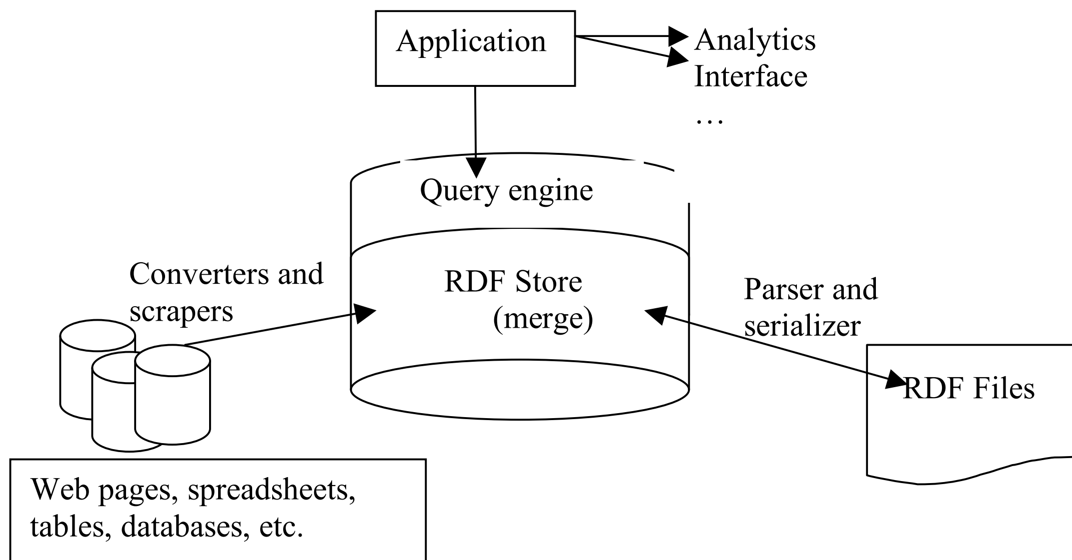

Figure 4-4 Application architecture for an RDF application

The application itself can take any of several forms. Most commonly, it is written in a conventional programming language (Java, C, Python and PERL are popular options). In this case, the RDF capabilities are provided as API bindings for that language. It is also common for an RDF store to provide a scripting language as part of the query system, which gives programmatic access to these capabilities, in a way that is not unlike

how advanced dialects of SQL provide scripting capabilities for relational database applications.

Regardless of the method by which the RDF store makes these functionalities available to the application, it is still the responsibility of the application to use them. Some examples of possible RDF applications include:

- Calendar integration – show appointments from different people and teams on a single calendar view,
- Map integration – show locations of points of interest gathered from different web sites, spreadsheets and databases all on a single map,
- Annotation – allow a community of users to apply keywords to information ("*tagging*") for others to consult, and
- Content management – make a single index of information resources (documents, web pages, databases, etc.) that are available in several content stores,

The application will decide what information sources need to be scraped or converted (e.g., diary entries in XML, lists of addresses from a web page, directory listings of content servers) . Depending on the volatility of the data, some of this process may even happen offline (e.g., the addresses of the Metro stations in Los Angeles are not likely to change for a while; this conversion could be done entirely outside the application context), while other data (like calendar data of team members) will have to be updated on a regular basis. Some data can remain in the RDF store itself (private information about this team); other data should be published in RDF form for other applications to use (information about the most popular documents in a repository).

Once all the required data sources have been scraped, converted, or parsed, the application uses the merge functionality of the RDF store to produce a single, federated graph of all the merged data. It is this federated graph that the application will use for all further queries. There is no need for the queries themselves to be aware of the federation strategy or schedule; the federation has already taken place when the RDF merge was performed.

From this point onward, the application behaves very like any other database application. A web page to display the appointments of any member of a team will include a query for that information. Even if the appointments came from different sources, and the information about team membership from still another source, the query is made against the federated information graph.

4.4.1 RDF-backed Web Portals

When the front-end of an application is a web server, the architecture shown in Figure 4-3 is the well-known architecture for a database-backed web portal. The pages are generated using any of a number of technologies (e.g., CGI, ASP, JSP, ZOPE) that allow web pages to be constructed from the results of queries against a database. In the earliest days of the web, web pages were typically stored statically as files in a file system. The move to database-backed portals was made to allow web sites to reflect the complex interrelated structure of data as it appears in a relational database.

The system architecture outlined in Figure 4-4 can be used the same way to implement a semantic web portal. The RDF store plays the same role that the database plays in database-backed portals. It is important to note that because of the separation between the presentation layer in both Figure 4-3 and Figure 4-4, it is possible to use all

the same technologies for the actual web page construction for a semantic web portal as are used in a database-backed portal. However, because of the distributed nature of the RDF store that backs a semantic web portal, information on a single web page typically comes from multiple sources. The merge capability of an RDF store supports this sort of information distribution as part of the infrastructure of the web portal. When the portal is backed by RDF, there is no difference between building a distributed web portal and one in with all the information is local. Federated web portals are as easy as siloed portals.

4.5 Data federation

The RDF data model was designed from the beginning with data federation in mind. Information from any source is converted into a set of triples, so that data federation of any kind – spreadsheets and XML, database tables and webpages – is accomplished with a single mechanism. As shown in Figure 4-4, this strategy of federation converts information from multiple sources into a single format, then combines all the information into a single store. This is in contrast to a federation strategy in which the application queries each source using a method corresponding to that format. RDF does not refer to a file format or a particular language for encoding data, but rather to the data model of representing information in triples. It is this feature of RDF that allows data to be federated in this way. The mechanism for merging this information, and the details of the RDF data model, can be encapsulated into a piece of software – the RDF store – to be used as a building block for applications.

The strategy of federating information first, then querying the federated information store, separates the concerns of data federation from the operational concerns of the application. Queries written in the application need not know where a particular triple

came from. This allows a single query to seamlessly operate over multiple data sources without elaborate planning on the part of the query author. This also means that changes to the application to federate further data sources will not impact the queries in the application itself.

This feature of RDF applications forms the key to much of the discussion that follows. In our discussion of RDFS and OWL, we will assume that any federation necessary for the application has already taken place; that is, all queries and inferences will take place on the *federated graph*. The federated graph is simply the graph that includes information from all the federated data sources, over which application queries will be run.

4.6 Chapter summary: RDF and Modeling

The components described in this chapter – RDF parsers, serializers, stores and query engines – are not semantic models themselves; they are the components of a systme that will include semantic models. Even the information represented in RDF does not necessarily a semantic model. These are the building blocks that go into making and using a semantic model. The model will be represented in RDF, to be sure. As we shall see, the semantic modeling languages of the W3C, RDFS and OWL, are built entirely in RDF, and they can be federated, just like any other RDF data.

Where do semantic models fit in to the application architecture of Figure 4-4? As data expressed in RDF, they will be housed in the RDF store, along with all other data. But semantic models are not simply data that will be used to answer a query, like the list of plays that shakespeare wrote or the places where paper machines are kept. Semantic models are meta-data; they are data that help to organize other data. When we federate

information from multiple sources, the RDF data model allows us to represent all the data in a single, uniform way. But it does nothing to resolve any conflicts of meaning between the sources. Do two states have the same definitions of “marriage”? Is the notion of “writing” a play the same as the notion of “writing” a song? It is the semantic models that give answers to questions like these. A semantic model acts as a sort of glue between disparate, federated data sources, so that we can describe how they fit together.

Just as anyone can say anything about any topic, so also can anyway say anything about a model; that is, anyone can contribute to the definition and mapping between information sources. In this way, not only can a federated, RDF-based, semantic application get its information from multiple sources, it can even get the instructions on how to combine information from mulitple sources. In this way, the semantic web really is a web of meaning, with multiple sources describing what the information on the web means.

4.7 Fundamental Concepts

The following fundamental concepts were introduced in this chapter:

RDF Parser/Serializer: A system component for reading and writing RDF in one of several file formats.

RDF Store. A database that works in RDF. One of its main operations is to merge RDF stores.

RDF Query Engine. Provides access to an RDF store, much as an SQL engine provides acccess to a relational store.

SPARQL. W3C standard query language for RDF.

Application interface. The part of the application that uses the content of an RDF store in an interaction with some user.

Scraper. A tool that extracts structured information from web pages.

Converter. A tool that converts data from some form (e.g., tables) into RDF.