This book is the result of a collaborative effort of a community of people like you, who believe that knowledge only grows if shared. We are waiting for you!

Get in touch with the rest of the team by visiting http://join.wikilearn.org

You are free to copy, share, remix and reproduce this book, provided that you properly give credit to original authors and you give readers the same freedom you enjoy. Read the full terms at https://creativecommons.org/licenses/by-sa/3.0/
# Contents

1 Introduction and Application Scenarios ............................................. 1
   1.1 Introduction .................................................................................. 1
   1.2 Semantic Technologies and Linked Data Foundations ................. 1
      1.2.1 Background standards ......................................................... 6
   1.3 Introduction to Linked Data .......................................................... 15

2 Querying Linked Data ........................................................................... 20
   2.1 Introduction .................................................................................. 20
   2.2 Introduction to SPARQL ............................................................... 20
   2.3 Updating Linked Data with SPARQL 1.1 ..................................... 36

3 Providing Linked Data .......................................................................... 50
   3.1 Introduction .................................................................................. 50
   3.2 The Linked Data Lifecycle ............................................................ 50
   3.3 Creation of Linked Data ............................................................... 52
   3.4 Interlinking Linked Data ............................................................. 54
   3.5 Publishing Linked Data ............................................................... 56
   3.6 Providing Linked Data: Checklist ................................................. 64
   3.7 Linked Data Catalogs and Tools for Providing Linked Data ........ 64
      3.7.1 3.9 Tools for Providing Linked Data .................................. 70

4 Interaction with Linked Data ............................................................... 87
   4.1 Introduction .................................................................................. 87
   4.2 Linked Data Visualization ............................................................ 88
   4.3 Linked Data Search ..................................................................... 112
   4.4 Methods for Linked Data Analysis ............................................. 119

5 Building Linked Data Applications ....................................................... 123
   5.1 Introduction .................................................................................. 123
   5.2 Characterization of Linked Data Applications ........................... 124
5.3 Architecture of Linked Data Applications .................................. 131
5.4 Using Web APIs ................................................................. 135

6 Text and image sources, contributors, and licenses ........................ 141
6.1 Text .................................................................................. 141
6.2 Images .............................................................................. 142
6.3 Content license ................................................................. 151
Chapter 1

Introduction and Application Scenarios

1.1 Introduction

The last decade has seen a growing interest in the Semantic Web, which extends the web of documents to a web of data. This technology applies web-based standards for encoding datasets and linking them to other published datasets, so that applications can exploit data from many different sources. It also provides standards for encoding general knowledge in ontologies, so allowing enhancements based on automatic reasoning (improved querying, for example).

This chapter introduces Linked Data and related semantic technologies, and shows how they can be deployed in web applications.

1.2 Semantic Technologies and Linked Data Foundations

We will describe a set of technologies that allows datasets to be published over the web, and queried effectively by applications. Compared with search engines such as Google and Yahoo, which are based on text-string matching, these technologies are “semantic”. This means that information is represented not in a natural language like English or Spanish, but in a graph-based data model that facilitates extension, integration, inference and uniform querying. As a realistic application of semantic technologies, we consider the provision of a portal through which users can retrieve resources and information in the world of music. Consider for example the following tasks:

- Retrieve a performance of the Beethoven violin concerto by a Chinese orchestra
- Retrieve a photograph of the conductor of this performance
- List male British rock musicians married to Scandinavians
Attempts to answer such queries through text-based search are unreliable: we might equally retrieve a performance in which the soloist was Chinese, or a rock musician that plays Scandinavian music. Using semantic technologies, resources such as the audio file of the performance, or the photograph of the conductor, can be annotated using the Resource Description Framework (RDF). In this framework, formal names can be assigned to what are called resources, which would include Beethoven, his violin concerto, the orchestra, and the conductor. Names can also be assigned to types (or classes) of resource (composers, concertos, etc.), and to relationships (or properties) that link resources (e.g., the “composed-by” relationship between composition and composer). By reasoning over facts encoded in this way, a query system can confirm that a performance was given by the Beijing Symphony Orchestra, that this orchestra is based in Beijing, that Beijing is located in China, and so forth – thus combining geographical and musical knowledge in order to retrieve an answer.

In designing these semantic technologies, a key design decision was to leave open the naming of resources and properties, provided that names conform to the format for web resource names – that is, provided they are Uniform Resource Identifiers or URIs.

http://rdf.freebase.com/ns/en.ludwig_van_beethoven

http://dbpedia.org/resource/Ludwig_van_Beethoven

http://musicbrainz.org/artist/1f9df192-a621-4f54-8850-2c5373b7eac9#

http://data.nytimes.com/N30866506154608358173

All four of the above could be names for Beethoven, illustrating that the URI need not be human-readable (e.g., it might be an arbitrary string of letters and numbers), although identifiers should be resolvable to RDF representations that include human-readable labels, as explained below. If data from different sources are to be combined, it is therefore important to establish links, for instance through statements indicating that the above four URIs are synonymous. These statements, which can also be expressed in RDF, provide a means by which data published by many people or organisations can be combined into linked data.

In the following chapters, we will show through practical examples how to describe resources in RDF, how to convert data from other formats to RDF, how to publish RDF data, and how to link published RDF to other datasets. We will also consider how to utilize existing linked data in applications for querying, analysis, mining, and visualisation.

**Background technologies**

Linked data results from a confluence of earlier ideas and technologies, including hypertext, databases, ontologies, markup languages, the Internet, and the World Wide Web. In this section we provide background information on these technologies.
Internet  The Internet is an extension of the technology of computer networks. The earliest computers operated independently. In the 1960s and 1970s, it became common for computers in an organisation (e.g., university, government, company) to be linked together in a network. At the same time, there were early experiments in linking whole networks together, including the ARPANET in the United States. In the early 1980s, the Internet Protocol Suite (TCP/IP) for the ARPANET was standardised, to provide the basis for a network of networks that could embrace the whole world. The Internet spread mostly to Europe and Australia during the 1980s, and to the rest of the world during the 1990s.

The technology supporting the Internet includes the IP (Internet Protocol) system for addressing computers, so that messages can be routed from one computer to another. Each computer on the Internet is assigned an IP number which can be written as four integers from 0–255 separated by dots, e.g. 185.56.200.4. (To be precise, this convention holds for version 4 of the IP, but not the more recent version 6.) The structure of messages is governed by application protocols that vary according to the service required (e.g., email, telephony, file transfer, hypertext). Examples of such protocols are FTP (File Transfer), USENET, and HTTP (HyperText Transfer).

Hypertext  The concept of hypertext is normally dated from Bush and Wang’s 1945 article “As we may think” [1], which proposed an organisation of external records (books, papers, photographs) corresponding to the association of ideas in human memory. By the 1960s, with more advanced computer technology, this concept was implemented by pioneers such as Douglas Engelbart and Ted Nelson in programs that allowed texts (or other media) to be viewed with some spans marked as hyperlinks, through which the reader could jump to another document.

World Wide Web  Informally people often use the terms “Internet” and “World Wide Web” (WWW) interchangeably, but this is inaccurate: the WWW is in fact just one of many services delivered over the Internet. The distinctive feature of the WWW is that it is a hypertext application, which exploits the Internet to allow cross-linking of documents all over the world.

The formal proposal for the WWW, and prototype software, were produced in 1990 by Tim Berners-Lee [2], and elaborated over the next few years. The basic idea is that a client application called a web browser obtains access to a document stored on another computer by sending a message, over the Internet, to a web server application, which sends back the source code for the document. Documents (or web pages) are written in the Hypertext Markup Language (HTML), which allows some spans to be marked as hyperlinks to a document at a specified location in the web, named using a Universal Resource Locator (URL). When the user clicks on a hyperlink, the browser finds the IP address associated with the URL, and sends a message to this IP address requesting the HTML file at the given location in the server’s file system; on receipt, this file is displayed in the browser.
Web 1.0 (static) In 1993 came a turning point for the WWW with the introduction of the Mosaic web browser, which could display graphics as well as text. From that date, usage of the web grew rapidly, although most users operated only as consumers of content, not producers. During this early phase of web development, sometimes called Web 1.0, web pages were mostly static documents read from a server and displayed on a client, with no options for users to contribute content, or for content to be tailored to a user’s specific demands.

Web 2.0 (dynamic) Around 2000 a second phase of web development began with the increasing use of technologies allowing the user of a browser to interact with web pages and shape their content. There are basically two ways in which this can be done, known as client-side scripting, and server-side scripting.

Client-side scripting is achieved through program code incorporated into the HTML source, typically written in Javascript. This code can be run on the user’s computer, without any need for further messages to be sent to the server: hence “client-side”.

Server-side scripting is achieved through messages to the server which invoke applications capable of creating the HTML source dynamically: the document eventually displayed to the user is therefore tailored in response to a specific request rather than retrieved from a previously stored file.

Social web These Web 2.0 technologies have made possible a wide range of social web sites now familiar to everyone, including chat rooms, blogs, wikis, product reviews, e-markets, and crowdsourcing. Previously a consumer of content provided by others, the web user has now become a prosumer, capable of adding information to a web page, and in this way communicating not only with the server, but through the server with other clients as well.
Web 3.0 (semantic) During the 1990s, Berners-Lee and collaborators developed proposals for a further stage of web development known as the Semantic Web. This far-reaching concept, first publicised in a 2001 article in the Scientific American [3], is partly implemented in the current stage of web development sometimes called Web 3.0. At present we cannot see clearly what lies beyond Web 3.0, but in Figure 1 we allow for future stages in Semantic Web development by including a loosely defined further stage “Web 4.0”.

In their 2001 article, Berners-Lee and co-authors pointed out that existing web content was usable by people but not by computer applications. There were many computer applications available for tasks like planning, or scheduling, or analysis, but they worked only on data files in some standard logical format, not on information presented in natural language text. A person could plan an itinerary by looking at web pages giving flight schedules, hotel locations, and so forth, but it was not yet possible (then as now) for programs to extract such information reliably from text-based web pages. The initial aim of the Semantic Web is to provide standards through which people can publish documents that consist of data, or perhaps a mixture of data and text, so allowing programs to combine data from many datasets, just as a person can combine information from many text documents in order to solve a problem or perform a task.

![Diagram](image.png)

Figure 2: From documents to data
Source: Own source.

Ontologies Datasets usually encode facts about individual objects and events, such as the following two facts about the Beatles (shown here in English rather than a database format):

The Beatles are a music group
The Beatles are a group

There is something odd about this pair of facts: having said that the Beatles are a music group, why must we add the more generic fact that they are a group? Must we list these two facts for all music groups – not to mention all groups of acrobats or actors etc.? Must we also add all other consequences of being a music group, such as performing music and playing musical instruments?
Ontologies allow more efficient storage and use of data by encoding *generic* facts about classes (or types of object), such as the following:

Every music group is a group
Every theatre group is a group

It is now sufficient to state that the Beatles (and the Rolling Stones, etc.) are music groups, and the more general fact that they are groups can be derived through inference. Ontologies thus enhance the value of data by allowing a computer application to infer, automatically, many essential facts that may be obvious to a person but not to a program.

To allow automatic inference, ontologies may be encoded in some version of mathematical logic. There are many formal logics, which vary in expressivity (the meanings that can be expressed) and tractability (the speed with which inferences can be drawn). To be useful in practical applications it is necessary to trade expressivity for tractability, and *description logic*, which is implemented in the Web Ontology Language OWL, does precisely this. However, despite these restrictions on expressivity, OWL cannot yet be used efficiently for inference over very large datasets, as required by Linked Data applications. For this reason, most reasoning for Linked Data relies on the far simpler logical resources of RDF-Schema, with OWL used sparingly if at all.

### 1.2.1 Background standards

The technologies described in the previous section are implemented through a number of standard protocols and languages, with probably familiar acronyms like HTTP, URI, XML, RDF, RDFS, OWL, SPARQL. You can look up details of these standards as needed, but as background it is useful to know a little about each one, and in particular what they are for. The later standards in this list build on the earlier ones, so they are often described as a *stack* of languages, as shown in Figure 3.

**HTTP**

From using the World Wide Web, most people are familiar with the HTTP prefix in front of web addresses such as [http://musicbrainz.org/](http://musicbrainz.org/). The meaning of this acronym is HyperText Transfer Protocol, and it refers to a set of conventions governing communication between a client and a server. More precisely, these conventions define the structure of request messages from client to server, and response messages from server to client. Message structure varies from one protocol to another: thus a different protocol such as FTP (File Transfer Protocol) will define a different message structure. A request messages in HTTP consists essentially of a *method* to be applied to a *resource*. The fundamental method is GET, which requests the server to send back a representation of the resource, typically an HTML file that can be displayed in a browser pane. However, there are several other methods including DELETE, which deletes the resource, and POST, which submits data to be processed with respect to the resource. The resource, specified through a relative document ID (often a filename/path on the
server), may be a document, or picture, or an executable that will generate data for the response.

**URI**

A Uniform Resource Identifier (URI) is defined in the standard [4] as “a compact sequence of characters that identifies an abstract or physical resource”. The word “compact” here means that the string must contain no space characters (or other white-space padding). “Abstract or physical” means that the URI may refer to an abstract resource such as the concepts “Beethoven” and “symphony”, as well as to a document or other file that can be retrieved from the WWW.

A URI that is linked to a retrievable resource is known also as a Uniform Resource Locator, or URL. For instance, the following URI for the MusicBrainz FAQ page is a URL:

```
http://musicbrainz.org/doc/Frequently_Asked_Questions
```

The definition of a correctly formed URI is quite complicated, with constituents that vary according to the scheme (the initial constituent before the colon), which specifies the relevant internet protocol, such as HTTP. For an HTTP URI, the other constituents most relevant for our purposes are the authority, and the path, which occur in that order. The authority specifies the server where the resource (if it really exists) is located. Finally, the path locates the resource precisely within the server’s directory structure.

Thus for the URL given above, “http” is the scheme, “musicbrainz.org” is the authority, and “/doc/Frequently_Asked_Questions” is the path; the other characters such as the colon are punctuation separating these constituents.

Note that the constituents following the scheme will be different for different schemes: thus the ‘tel’ scheme, for example, is followed simply by a telephone number. Here are some examples indicating this variety:

```
ldap://[2001:db8::7]/c=GB?objectClass?one
mailto:John.Doe@example.com
news:comp.infosystems.www.servers.unix
tel:+1-816-555-1212
telnet://192.0.2.16:80/
urn:oasis:names:specification:docbook:dtd:xml:4.1.2
http://dbpedia.org/resource/Karlsruhe
```

Since URIs are typically long, and hence difficult to read and write, it is convenient to make use of abbreviated forms known as “compact URIs” or “CURIEs”. A compact URI consists simply of a namespace and a local name, separated by a colon. Typically, the namespace includes the scheme, the authority, and perhaps the early part of the path; the local name contains the remainder of the URI, chosen so as to convey intuitively what the URI means, while observing some syntactic restrictions (e.g., there should be no further use of the characters “/” and “#”). Thus in the example just given, one could introduce a namespace “dbp” for http://dbpedia.org/resource/, so reducing the URI to ”dbp:Karlsruhe”, where the local name preserves the substring that is significant to human readers. We will use this convenient method of abbreviation often in the rest of this book.
XML

Extensible Markup Language (XML) is a refinement of Standard Generalised Markup Language (SGML), which was introduced in the 1980s as a meta-language suitable for defining particular mark-up languages – for instance, languages for adding formatting information to documents. The basic concept, now well known from widespread use of HTML, is that labelled tags are placed around spans of text, thus indicating perhaps that the span should be formatted in italics:

```
<i>text in italics</i>
```

The italic tag “i” is part of HTML, not SGML, but the convention of placing tags within angle brackets, and distinguishing the closing tag by a forward slash character, comes from SGML, as does the syntax for adding attributes to the opening tag, as in this example yielding blue text:

```
<font color="blue">blue text</font>
```

SGML is versatile because it can be used simply for encoding data, as well as for adding structure to text.

In the mid-1990s, the newly formed World Wide Web Consortium (abbreviated W3C) set up a working group to simplify and rework SGML to meet the requirements of the WWW. The result was the first XML specification, which became a W3C recommendation in 1998, and has become the standard convention for data exchange over the web. The essential advance on SGML is that XML is simpler and stricter: to give just one example, it is permissible in SGML (but not in XML) to omit closing tags, as in the common practice of inserting `<p>` without a closing `</p>` when writing HTML.
The Resource Description Framework (RDF) was introduced originally as a data model for metadata, which are attributes of a document, or image, or program, etc. such as its author, date, location, and coding standards. First published as a W3C recommendation in 1999 [5], the framework has since been updated, and generalised in its purpose to cover not only metadata (strictly interpreted) but knowledge of all kinds.

The basic idea of RDF is a very simple one: namely, that statements are represented as triples of the form subject–predicate–object, each triple expressing a relation (represented by the predicate resource) between the subject and object resources. Formally, the subject is expressed by a URI or a blank node, the predicate by a URI, and the object by a URI or a literal such as a number or string.

The original W3C recommendation for exposing RDF data was that it should be encoded in XML syntax, sometimes called RDF/XML. It is for this reason that the semantic web “stack” of languages has RDF implemented on top of XML. However, notations have also been proposed which are easier for people to read.
and write, such as Turtle, in which statements are formed simply by listing the elements of the triple on a line, in the order subject-predicate-object, followed by a full stop, with URIs possibly shortened through the use of namespace abbreviations defined by “prefix” and “base” statements, as in the following example:

```turtle
@base <http://musicbrainz.org/>.
@prefix mo:<http://purl.org/ontology/mo/>.
<artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d#_> a mo:MusicGroup.
```

Here the subject is abbreviated using the “base” statement, and the object is abbreviated using the “prefix” statement. The very simple predicate “a” relies on a further Turtle shorthand for very commonly used predicates, and refers to the “type” relation between a resource and its class. This can be seen from the following equivalent Turtle statement, in which all URIs are shown in their cumbersome unabbreviated form. Note that this statement should occupy a single line, although it is shown here with wrapping so that it fits on the page. The format in which every URI in a Turtle statement is fully expanded is also known as NTriples.

```turtle
```

Where multiple statements apply to the same subject, they can be abbreviated by placing a semi-colon after the first object, and then giving further predicate-object pairs separated by semi-colons, with a full stop after the final pair. For statements having the same subject and predicate, objects can be listed in a similar way separated by commas. These conventions are illustrated by the following statements:

```turtle
@base <http://musicbrainz.org/>.
@prefix mo:<http://purl.org/ontology/mo/>.
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl:<http://www.w3.org/2002/07/owl#>.
@prefix dbpedia:<http://dbpedia.org/resource/>.
@prefix bbc:<http://www.bbc.co.uk/music/artists/>.
<artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d#_>
  rdfs:label "The Beatles";
  owl:sameAs dbpedia:The_Beatles,
  bbc:b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d#_ artist.
```

**RDFS**

RDF Schema (RDFS) is an extension of RDF which allows resources to be classified explicitly as classes or properties; it also supports some further statements that depend on this classification, such as class-subclass or property-subproperty
relationships, and domain and range of a property. Some important resources in RDFS are as follows (for brevity we use the “rdfs” prefix defined above):

**rdfs:Class** A resource representing the class of all classes.

**rdfs:subClassOf** Used as a predicate to mean that the subject is a subclass of the object.

**rdfs:subPropertyOf** Used as a predicate to mean that the subject is a subproperty of the object.

**rdfs:domain** Used as a predicate when the subject is a property and the object is the class that is domain of this property.

**rdfs:range** Used as a predicate when the subject is a property and the object is the class that is range of this property.

The following statements in Turtle serve to illustrate these RDFS resources. Note that they use abbreviated URLs for which the prefixes are given above.

```turtle
mo:member rdf:type rdfs:Property.
mo:member rdfs:domain mo:MusicGroup.

mo:member rdfs:range foaf:Agent.
mo:MusicGroup rdfs:subClassOf foaf:Group.
```

In these statements, the resource "mo:member" denotes the property that relates a music group to each of its members – for instance, the Beatles to John, Paul, George and Ringo, as in the following triple:

```turtle
dbpedia:The_Beatles mo:member dbpedia:Ringo_Starr.
```

The second and third statements above give the domain and range of the property "mo:member". Intuitively, their meaning is that if "mo:member" is employed as predicate in a triple, its subject will belong to the class "mo:MusicGroup", and its object to the class "foaf:Agent". The fourth statement means that any resource belonging to the class "mo:MusicGroup" will also belong to the (more general) class "foaf:Group".

An important gain in adding such statements is that they allow new facts to be inferred from existing ones. Consider for instance how they may be combined with the statement (just given) that Ringo is a member of the Beatles. Using the domain and range statements for the property "mo:member", it follows directly that the Beatles are a music group, and that Ringo is an agent; using the subClassOf statement, it follows further that the Beatles are a group. Encoded in Turtle, these inferred facts are as follows:

```turtle
dbpedia:Ringo_Starr rdf:type foaf:Agent.
dbpedia:The_Beatles rdf:type foaf:Group.
```

RDFS also contains some predicates for linking a resource to information useful in presentation and navigation, but not for inference. These include the following:
**rdfs:comment** Associates a resource with a human-readable description of it.

**rdfs:label** Associates a resource with a human-readable label for it.

**rdfs:seeAlso** Associates a resource with another resource that might provide additional information about it.

**rdfs:isDefinedBy** A sub-property of "rdfs:seeAlso", indicating a resource that contains a definition of the subject resource.

**OWL**

The Web Ontology Language (OWL) extends RDFS to provide an implementation of a description logic, capable of expressing more complex general statements about individuals, classes and properties.

OWL was developed in the early 2000s and became a W3C standard (along with RDFS) in 2004. The acronym OWL was preferred to the more logical WOL because it is easier to pronounce, provides a handy logo, and is suggestive of wisdom. Of course the name also reminds us of the character in “Winnie the Pooh” who misspells his name “Wol”.

The reason for choosing description logic, rather than a more expressive kind of mathematical logic, has already been mentioned: the aim was to achieve fast scalable reasoning services, and hence to use a logic for which efficient reasoning algorithms were already available. In fact description logics are more a family of languages than a single language. They can be thought of as a palette of operators for constructing classes, properties and statements, from which the user can make different selections, so obtaining fragments with different profiles of expressivity and tractability.
The OWL standard is under constant development, and the current version OWL 2.0 provides for the fragments shown in Figure 4; their meanings are as follows:

**OWL 2 Full** Used informally to refer to RDF graphs considered as OWL 2 ontologies and interpreted using the RDF-Based Semantics.

**OWL 2 DL** Used informally to refer to OWL 2 ontologies interpreted using the formal semantics of Description Logic (“Direct Semantics”).

**OWL 2 EL** A simple fragment limited to basic classification, allowing reasoning in polynomial time.

**OWL 2 QL** A fragment designed to be translatable to querying in relational databases.

**OWL 2 RL** A fragment designed to be efficiently implementable using rule-based reasoners.

As already explained, a detailed understanding of OWL is not necessary for working with Linked Data. When reasoning over huge amounts of data, only the simplest reasoning processes are computationally efficient, and these can for the most part be implemented using only the resources of RDFS. Very briefly, the additional resources in OWL are terms providing mainly for the following:

- Class construction: forming new classes from existing classes, properties and individuals (e.g., ObjectIntersectionOf);
- Property construction: distinguishing object properties (resources as values) from data properties (literals as values);
- Class axioms: statements about classes, describing sub-class, equivalence and disjointness relationships;
- Property axioms: statements about properties, including relationships such as equivalence and sub-property, and also attributes such as whether a property is functional, transitive, and so forth;
- Individual axioms: statements about individuals, including class membership, and whether two resources represent the same individual or different individuals.

**SPARQL**

The SPARQL Protocol and RDF Query Language (a recursive acronym, since it contains itself) is a language for formulating queries over RDF data. It is the Semantic Web’s counterpart to SQL (Structure Query Language), which has been a standard language for querying relational databases since the 1980s. SPARQL
is a recent addition to the Semantic Web stack of languages, having been recommended as a W3C standard in 2008 [6]. Since chapter 3 of this book is dedicated to SPARQL, we limit ourselves here to an example that illustrates its purpose. Comparing SPARQL with SQL, the key difference is that it is designed for retrieving information from sets of triples, rather than from data organised into relations (i.e., tables). Queries are therefore formulated using lists of RDF triples in which some URIs or literals are replaced by variables, as in the following:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX music-ont: <http://purl.org/ontology/mo/>

SELECT ?album_name ?track_title
WHERE {
  dbpedia:The_Beatles foaf:made ?album .
  ?track dc:title ?track_title
}
```

Translated into English, the meaning of this query is as follows:

Retrievalist of all album names AN and track titles TT in the data for which the following conditions hold:
1. There is an album A made by the Beatles.
2. Album A has the title AN.
3. There is a track T on album A.
4. Track T has the title TT.

Or more colloquially: retrieve the titles of all tracks on albums by the Beatles, along with the corresponding album titles. The response should be a list of pairs, each containing an album name and a track title.

This example shows the simplest kind of query, in which the WHERE statement is simply a list of triples (containing variables). SPARQL also provides some more sophisticated constructs: these include FILTER, which allows conditions on the values of variables (e.g., that a number should be between 1990 and 2000); also OPTIONAL, which specifies data that should be retrieved if available, while allowing the query to succeed even when they are unavailable. For more information on these more complex constructs, see Chapter 3.

Practically, to pose a query to a dataset you need to use a program or website that serves as a SPARQL endpoint. For a list of endpoints see the W3C site at http://www.w3.org/wiki/SparqlEndpoints. Typically, an endpoint interface provides text fields where you can type the URL of the dataset you wish to query, and the query itself (e.g., the SELECT query in the example above). On hitting the “Submit” button, you obtain a dynamically generated webpage listing the values of the query variables in a table. There are also libraries allowing
you to incorporate SPARQL queries into your programs, such as the Java library Jena at http://jena.apache.org/ or php BOTK http://ontology.it/tools/botk.

**HIGHLIGHT:** LinkedData.Center provides you with a fully configured and fully managed server that exposes a private SPARQL endpoint. Thanks to high capacity RDF storage (included in the service) you are able to load any number of RDF triple and query the resulting graph using SPARQL.

### 1.3 Introduction to Linked Data

In 2006 Berners-Lee wrote an influential note suggesting principles for the publication of data on the semantic web. The original text can be found at this web address:

http://www.w3.org/DesignIssues/LinkedData

Since then the volume of data has grown from around 2 billion triples in 2007 to over 30 billion in 2011, interconnected by over 500 million RDF links, the main purpose of which is to establish chains of URIs that refer to the same individuals. Through such links, published datasets are combined into a vast body of data known as a “cloud”.

**Figure 5:** Linked data cloud (2007)

Source: http://lod-cloud.net

Citation: Linking Open Data cloud diagram (2007), by Richard Cyganiak and Anja Jentzsch. http://lod-cloud.net

License: CC-BY-SA

Figure 5 shows a diagram of the linked data cloud for 2007, in which nodes represent published datasets, and links represent sets of RDF triples through which the URIs in one dataset are paired with their counterparts in another dataset. Thus the link from DBpedia to MusicBrainz means that DBpedia includes not only RDF triples that give information about the world, but also triples that link some DBpedia names to their synonyms in MusicBrainz. We have seen examples of such statements in the last section, including the following triple which links the two names for the Beatles.
<http://musicbrainz.org/artist/b10bbbfccf9e-42e0-be17e2c3ed600d>
<http://www.w3.org/2002/07/owl#sameAs>
<http://dbpedia.org/resource/The_Beatles>.

Note that since the “sameAs” relation is transitive and commutative, two statements of the form “X sameAs Y” and “Y sameAs Z” (or equivalently “Z sameAs Y”) can be combined to infer “X sameAs Z”; in this way, lists of synonymous names can be derived from the cloud.

Principles In his 2006 note, Berners-Lee set out four simple principles for publishing data on the web. These are best seen as rules of best practice rather than rules that must be obeyed: the idea is that the more people follow these principles, the more their data will be usable by others.

In brief, the principles are as follows:

1. Use URIs to identify things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, RDFS, SPARQL).
4. Include links to other URIs, so that they can discover more things.

The rationale for these principles is probably obvious. By using URIs to identify individuals, classes, and properties, we obtain names that perform a double duty: as well as referring to the relevant thing, they give us a location on the web where we may look for information about that thing. Other naming schemes accomplish only the first of these duties. However, to obtain benefit from a name that also serves as a web address, the URI should not be a broken link. It should point to relevant information, encoded in one of the expected formats. This benefit will be enhanced further if the information includes URIs that point to other locations on the web from which additional relevant information might be recovered.

Rating published datasets In 2010 Berners-Lee extended the note referenced above to propose a system for rating datasets, based on the five-star rating system used for hotels. Closely related to the principles just listed, the system is as follows:

- One-star (*): The data is available on the web with an open license.
- Two-star (**): The data is structured and machine-readable.
- Three-star (***): The data does not use a proprietary format.
- Four-star (****): The data uses only open standards from W3C (RDF, SPARQL).
- Five-star (*****): The data is linked to that of other data providers.
Note that every level here includes the previous levels: thus for instance three-star data must also be available on the web in machine-readable form.

**Growth of linked data on the web**  We have shown above a diagram of the linked data cloud for 2007 (Figure 5). For comparison, Figure 6 shows the corresponding diagram for 2014 (the last year for which we have data), showing the expansion that has taken place during these years. The picture is just considering dataset reachable by [] crawler and today it is VERY partial.

![Figure 6: Linked data cloud (2014)](http://lod-cloud.net)

*Source:* [http://lod-cloud.net](http://lod-cloud.net)

*Citation:* Linking Open Data cloud diagram (2014), by Richard Cyganiak and Anja Jentzsch. [http://lod-cloud.net](http://lod-cloud.net)

*License:* CC-BY-SA

The colours on this diagram provide a broad categorisation of the domains of the various datasets.

**Examples**

To explore the possibilities of linked data browsers and data mashups (which combine data from many sources), look at these examples of working websites.

**Illustreets**  Illustreets is a web application developed by Manuel Timita and Kateryna Koval, available at [http://illustreets.co.uk](http://illustreets.co.uk) free of charge

This application is particularly useful when looking to buy or rent property anywhere in England, UK. illustreets puts deprivation, crime, education, transport, environment, and census data on an interactive, searchable map, helping you to compare between locations on the fly.

You can hover over the map to get useful information about any location, in real time. Then, as you click on a coloured area, you get full details about that particular neighbourhood.

Also, users can filter the neighbourhoods by a number of criteria. For example, house hunters can filter the results by the number of bedrooms and the property
price. Unaffordable areas are then shaded out on the map and users can view detailed summaries of the areas where they can afford to live.

Figure: Illustreets screenshot
Source: https://data.gov.uk/
Citation: Manuel Timita and Kateryna Koval.

Illustreets uses following datasets from available in https://data.gov.uk/:

- English Indices of Deprivation
- Lower Layer Super Output Area (LSOA) boundaries
- 2011 Census
- Street level crime
- National Public Transport Access Nodes (NaPTAN)
- Independent school inspections and outcomes
- Maintained school inspections and outcomes

Figure: Illustreets screenshot
Source: https://data.gov.uk/
Citation: Manuel Timita and Kateryna Koval.
Some other sites For further examples of sites using Linked Data, see the following.

**BBC Music** The BBC has launched a music portal based on Linked Data at [http://www.bbc.co.uk/music](http://www.bbc.co.uk/music).

**LinkedGeoData** The University of Leipzig has a community project providing street map information based on Linked Data, at [http://linkedgeodata.org/](http://linkedgeodata.org/).

**US government data** In 2009 the US and UK governments made commitments to open data. The US government data site is at [http://www.data.gov/](http://www.data.gov/).

**UK government data** Available at [http://data.gov.uk/](http://data.gov.uk/) with over 8000 datasets published at the time of writing.

**Exercise**

Study the following RDF statements, expressed in the Turtle syntax, then attempt the exercises that follow.

```turtle
@base <http://LinkedData.Center/home/about#> .
@prefix org: <http://www.w3.org/ns/org#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

vocab:ResearchProject rdfs:subClassOf foaf:Group .

vocab:consortiumMember rdfs:subPropertyOf foaf:member .

<enrico> a foaf:Person ;
  foaf:givenName "Enrico" ;
  foaf:familyName "Fagnoni" ;

<http://LinkedData.Center/> a org:Organization ;
  rdfs:label "LinkedData.Center company" ;
  rdfs:comment "Linked Data as Service provider"@en, "fornitore di servizi di Linked Data"@it .
```

1. Re-express the statements in NTriples (i.e. remove all prefixes and abbreviations to give full triples in absolute URIs).

2. Add a resource representing yourself, attaching your name using the FOAF properties.

3. Add a property to this resource asserting that you know Enrico (i.e. foaf: knows).
Chapter 2

Querying Linked Data

2.1 Introduction

Having surveyed the main concepts and standards for Linked Data, we now look in detail at SPARQL (SPARQL Protocol And RDF Query Language). As its name suggests, SPARQL is fundamentally a language for formulating queries, through which information can be retrieved from datasets. However, since it is targetted at datasets published on the World Wide Web, it also provides a protocol for specifying SPARQL commands. Moreover, in the latest version SPARQL 1.1 it has been extended to allow updating as well as querying, so that a SPARQL command can require data to be added, revised or deleted.

We also look in more detail at RDFS and OWL, which allow developers to formulate conceptual knowledge that can be exploited by automatic reasoning services in order to enhance the semantics of queries.

2.2 Introduction to SPARQL

SPARQL was proposed as a standard by the World Wide Web Consortium (W3C) in November 2008. It is maintained and developed by the W3C SPARQL Working Group, who in November 2012 recommended an upgraded version SPARQL 1.1 with new features including an update language (allowing users to change as well as consult RDF datasets). The latest recommendation can be found at these two sites, one for the query language and one for update:

http://www.w3.org/TR/sparql11-query
http://www.w3.org/TR/sparql11-update

Along with RDF and OWL, SPARQL is one of the three core standards of the Semantic Web. Its location in the Semantic Web “stack of languages” is shown in Figure 1. One point to note in the figure is that SPARQL does not depend on RDFS and OWL. However, as will be shown later in the chapter, knowledge encoded in RDFS and OWL may enhance the power of querying.
Figure 1: SPARQL in Semantic Web stack

SPARQL, as a database query language, resembles the well-known Structured Query Language (SQL). The syntax of SPARQL is shaped by the fact that it operates over graph data represented as RDF triples, as opposed to SQL’s tabular data organised in a relational database.

The essence of querying is shown by the following illustration, using for the time being English rather than RDF. Imagine an RDF dataset with statements containing the following information:

The Beatles made the album “Help”.

The Beatles made the album “Abbey Road”.
The Beatles made the album “Let it be”.
The Beatles includes band-member Paul McCartney.
Wings made the album “Band on the run”.
Wings made the album “London Town”.
Wings includes band-member Paul McCartney.

The Rolling Stones made the album “Hot Rocks”.

One can imagine various queries that a music portal might need to run over such a dataset. For instance, the portal might construct web pages on demand for any album or group nominated by the user. This would require retrieval of information from the dataset for questions such as the following:

Who made the album “Help”?
Which albums did the Beatles make?

These are so-called WH-questions (“who”, “what”, “where”, etc.), for which the first would receive a single answer (“The Beatles”), and the second a list of three answers (“Help”, “Abbey Road”, “Let it be”). The SPARQL counterparts to these questions use RDF triples that contain variables; these correspond to the WH-words in the English queries. The general form for such questions (still working in English) is as follows:

Give me all values of X such that X made the album “Help”.
Give me all values of X such that the Beatles made X.
We can go further than this by introducing more than one variable, thus generalising the query:

Give me all values of X and Y such that X made Y.

This is like asking a question with two WH-words, such as “Which bands made which albums?”. The answer is not a list of values, as before, but a list of X-Y pairs that could be conveniently presented in a table:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Beatles</td>
<td>“Help”</td>
</tr>
<tr>
<td>The Beatles</td>
<td>“Abbey Road”</td>
</tr>
<tr>
<td>The Beatles</td>
<td>“Let it be”</td>
</tr>
<tr>
<td>Wings</td>
<td>“Band on the run”</td>
</tr>
<tr>
<td>Wings</td>
<td>“London Town”</td>
</tr>
<tr>
<td>The Rolling Stones</td>
<td>“Hot Rocks”</td>
</tr>
</tbody>
</table>

In all these examples, the question is represented by a single statement with one or more variables; however, we can also construct more complex queries containing several statements:

Give me all values of X and Y such that: (a) X made Y, and (b) X includes band member Paul McCartney.

The answer would be the first five pairs from the previous answer, excluding “Hot Rocks” since the dataset does not list Paul McCartney as a band member of the Rolling Stones.

Moving now from English to SPARQL, here is the encoding for the simple query “Which albums did the Beatles make?” for the MusicBrainz dataset. For now don’t worry about learning the exact syntax; the important thing is to understand what the various bits and pieces are doing.

```sparql
PREFIX dc: <http://purl.org/dc/elements/1.1/>  
PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
PREFIX mo: <http://purl.org/ontology/mo/>

SELECT ?album_title  
WHERE {  
}
```

The query begins with PREFIX statements that define abbreviations for namespaces. The query proper begins in the line starting SELECT, which also contains a variable (corresponding to X and Y in our English examples) starting with the question mark character '?'. Choose any word you like for the rest of the variable name, provided that you use it consistently. The remainder of the query, starting WHERE, contains a list of RDF triple patterns. These are like RDF triples except that they include variables. They are expressed in Turtle, which we introduced in Chapter 1.
The WHERE clause in the example has two RDF triple patterns, separated by a full stop. The first pattern matches resources made by the Beatles; the second requires that these resources belong to a class mo:SignalGroup (this rather weird name distinguishes albums, which are “signal groups”, from their constituent tracks, which are also encoded as resources made by the Beatles).

The response to a query is computed by a process known as graph matching.

SPARQL terminology Before proceeding to the detailed structure of queries, it is worth pausing to review the concepts introduced so far:

**RDF triple** An RDF triple is a statement of the form subject-predicate-object expressed in one of the RDF formalisms.

**RDF triple pattern** An RDF triple pattern is the same as an RDF triple except that any or all of its three constituents may be replaced by a variable.

**RDF graph** An RDF graph is a set of RDF triples. You probably know that “graph” in mathematics has two distinct meanings: (1) a diagram showing points arranged by their relationship to an X axis and a Y axis; (2) a set of vertices (or nodes) linked by edges (or arcs). In the case of RDF the second meaning applies, where the subject and object in a triple are vertices, and the predicate is an edge that links them by pointing from subject to object. Formally, an RDF graph can be described as a directed labelled multigraph, which means (a) that edges are directional (you cannot switch subject and object without changing the statement), (b) that edges are named (by the predicate identifier), and (c) that there can be multiple edges linking two vertices (resources may be related in different ways).

**RDF dataset** An RDF dataset is a set of RDF triples comprising a default RDF graph, which by definition is unnamed, and zero or more named RDF graphs. The idea behind this segmentation is that SPARQL queries can be explicitly confined to a named subset rather than running over the whole dataset.

**Graph pattern** We use this term to refer to a conjunction of RDF triple patterns. It is therefore the same as an RDF graph, except that its constituents are RDF triple patterns (which contain variables) as opposed to normal RDF triples (which don’t). Note that in a query, the expression following the keyword WHERE is a graph pattern; this is why graph patterns are important.

**SPARQL Protocol client** A SPARQL Protocol client is an HTTP client that sends requests for SPARQL Protocol operations. As you probably know, “client” here refers to a program that sends a request to another program, possibly running on another computer, over a network; the other computer is known as the “server”.

**SPARQL Protocol service** A SPARQL Protocol service is an HTTP server that services requests for SPARQL Protocol operations.

**SPARQL endpoint** A SPARQL endpoint is a SPARQL Protocol service, identified by a given URL, which listens for requests from SPARQL clients.
Querying with SPARQL

Submitting a query  In developing an application, you will need to build queries into your application code. There are APIs that help you to do this, like the ones provided by LinkedData.Center. However, before learning to use APIs, you can learn the syntax for queries (and their responses) by using an interactive SPARQL endpoint web application to enter the query by hand.

As an example of this procedure Figure shows a snapshot of the LinkedBrainz endpoint at http://sparql.linkedbrainz.org/ with a query about the albums made by the Beatles.

![Figure 3: Typing a query into a SPARQL endpoint](image)

If you type in this query and hit the “Run Query” button you will obtain a table giving values for the variable ?album_title:

<table>
<thead>
<tr>
<th>album_title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;'69 Rehearsals&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
</tr>
<tr>
<td>&quot;'Quote' Unquote: The Sixties Interviews&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
</tr>
<tr>
<td>&quot;1&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
</tr>
<tr>
<td>&quot;16 Superhits, Volume 1&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
</tr>
<tr>
<td>&quot;16 Superhits, Volume 2&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
</tr>
<tr>
<td>&quot;16 Superhits, Volume 3&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
</tr>
<tr>
<td>&quot;16 Superhits, Volume 4&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
</tr>
<tr>
<td>&quot;1962 Live Recordings&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
</tr>
<tr>
<td>&quot;1962 Live at Star Club in Hamburg&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
</tr>
<tr>
<td>&quot;1962-1970&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
</tr>
</tbody>
</table>

Please note that the clause LIMIT 10 in the SPARQL query limits the results to the first ten solutions that match the “WHERE” clause portion.

It is always a good idea to limit the number of the results in a SPARQL query, because some time returned solution can be really a lot. ORDER BY, LIMIT and OFFSET constructs, like in SQL, help you in paginating results.

Types of query  SPARQL defines the following query types for data retrieval:

ASK  An ASK query is a test of whether there are any resources in the dataset
matching the search pattern; the response is either true or false. Intuitively, it poses the question: “Are there any X, Y, etc. satisfying the following conditions ...?”

**SELECT** A SELECT query returns a table in which columns represent variables and rows represent variable bindings matching the search pattern. Intuitively: “Return a table of all X, Y, etc. satisfying the following conditions ...”.

**CONSTRUCT** A CONSTRUCT query returns an RDF graph (i.e., set of triples) matching a template, using variable bindings obtained from the dataset using a search pattern. Intuitively: “Find all X, Y, etc. satisfying the following conditions ... and substitute them into the following template in order to generate (possibly new) RDF statements ...”.

**DESCRIBE** A DESCRIBE query returns an RDF graph, extracted from the dataset, which provides all available information about a resource (or resources). The resource may be identified by name, or by a variable accompanied by a graph pattern. Intuitively: “Find all statements in the dataset that provide information about the following resource(s) ... (identified by name or description)”.

**Queries using ASK** An ASK query corresponds intuitively to a Yes/No question in conversational language. For example, the following query corresponds to the Yes/No question “Is “All You Need Is Love” a song recorded by the Beatles?”:

```sql
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

ASK
WHERE {
  ?band foaf:name "The Beatles" ; foaf:made [ 
    a mo:Track ; dc:title "All You Need Is Love"^^xsd:string 
  ].
}
```

If this query is submitted as described above, the answer given will be “true”. Dissecting the syntax of this query, we note the following:

- The PREFIX statements are not an essential part of a query, but here as elsewhere they are useful as a means of abbreviating RDF triples or patterns. Be careful not to put a full stop at the end of a PREFIX statement, since this will cause a syntax error.
- The query proper begins with ASK, which specifies the relevant query type.
WHERE introduces a graph pattern, which at its simplest is a conjunction of RDF triples or patterns, presented in curly brackets and separated by full stops (or by commas or semicolons, as mentioned in the section on Turtle in Chapter 1). The patterns may use abbreviations defined in the PREFIX statements, and may include one or more variables. (More complex graph patterns will be described later on.)

- Layout is free provided that terms are separated by white space. For instance, if you wished you could type the whole query on one line, or at the other extreme type a new-line character after every term. The layout given above with new lines for the key words PREFIX, ASK, WHERE, is adopted only for human readability.

- The keywords of SPARQL syntax – PREFIX, ASK, WHERE, etc. – are not case-sensitive, so if you prefer you can use prefix, ask, where, and so on. In the examples we consistently capitalise these words for reasons of readability, but this has no effect on how the query engine interprets the query.

If you want to ask whether there are any X, Y, etc. such that certain conditions hold – e.g., “Are there any X such that X was made by the Beatles” – you need to use RDF patterns, which are like triples except that they contain variables. These are represented by names beginning with a question mark '?', as in this example:

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX mo: <http://purl.org/ontology/mo/>  
PREFIX dc: <http://purl.org/dc/elements/1.1/>

ASK
WHERE { ?band foaf:name "The Beatles" ; foaf:made ?something }
```

Queries using SELECT To show the basic syntax of a SELECT query, let us return to the example given in section 2.3:

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX mo: <http://purl.org/ontology/mo/>

SELECT ?album
       ?album a mo:SignalGroup }
```

Note the following:

- After the (optional) PREFIX statements, the query proper begins with SELECT, which specifies the query type.
After `SELECT` you list all the variables that you would like to see tabulated in the response. Variables should be separated by spaces if there are more than one. Alternatively, you can simply put an asterisk after `SELECT`, meaning that all variables should be tabulated. In the example, this would yield the same result.

- As before, `WHERE` introduces a graph pattern including one or more variables.
- As before, layout is free provided that terms are separated by white space.

Try running the query in the interactive linkedbrainz endpoint. You should obtain in response a list of URIs that use arbitrary codes rather than recognisable words.

**Ordering the rows in the query result** For some queries you might want the results to be presented in a particular order. For instance, if your music portal retrieves albums made by the Beatles, using the query given above, you might want to present these in alphabetical order of title. This can be done using the keywords `ORDER BY`, as in the following example:

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

               ?album a mo:SignalGroup .
               ?album dc:title ?title }
ORDER BY ?title
```

The key element in this query is the `ORDER BY` component at the end, which stipulates that rows should be presented in alphabetical order of the values in the `?title` column. Note also one other change from the previous example, the use of the asterisk shorthand in the `SELECT` clause, which asks for all variables in the `WHERE` clause to be tabulated. This shorthand is not always used, since often the selected variables are a strict subset of those mentioned in the `WHERE` clause.

Try replacing `ORDER BY ?title` by `ORDER BY DESC(?title)`, which will present the rows in `descending` rather than ascending order of title (i.e., ascending is the default).

What exactly do these tables show? The columns, as we have seen, correspond to variables in the query for which tabulation is requested after `SELECT`, either explicitly by name, or implicitly by the `*` option. The rows give all variable bindings that match the graph pattern in the `WHERE` clause. A binding is an assignment of identifiers or literals to the variables which, when instantiated in these patterns, will yield a subgraph of the dataset. Note that when computing...
these bindings, the query engine makes the key assumption that variables occurring in more than one pattern are bound to the same resource. In this way it avoids returning a row in which an album is paired with the title of a different album.

**Returning results page by page** For some queries, including our example, the output table returns too many result at once. In such cases it would be useful if the music portal included a paging facility allowing users to view the information in manageable portions – perhaps ten rows at a time. This can be done through the query engine if you use the keywords LIMIT and OFFSET. To see how this works, trying extending our previous query as follows:

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT *
    ?album a mo:SignalGroup .
    ?album dc:title ?title }
ORDER BY ?title
LIMIT 10 OFFSET 0
```

You should get back the same table, but cut off after the first ten rows; the result will be the same if you just put LIMIT 10 leaving the offset unspecified. Now try raising the number after OFFSET to 10. You should get back the next ten-row segment of the table, covering rows 11-20. In general, if LIMIT is L and OFFSET is S, the query will return L rows starting at S+1 and continuing up to S+L.

**Using tests to filter the results** We have seen queries in which the WHERE clause contains specific resources (the Beatles) or variables (?album). But what if we want to obtain results for any variable that satisfies a certain condition – e.g., albums beginning with the letter ‘B’, or band members born before 1960? Such conditions are called “filters”, and to illustrate them, let us switch to an example in which our aim is to retrieve tracks (not albums) with a duration between 300 and 400 seconds. Since in MusicBrainz durations are encoded in milliseconds, the relevant filter condition can be stated as follows:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?title ?duration
    ?track a mo:Track .
    FILTER (?duration >= 300) AND (?duration < 400) }
ORDER BY ?title
LIMIT 10 OFFSET 0
```
FILTER (?duration>300000 && ?duration<=400000) }
ORDER BY ?duration
LIMIT 10

The tables in Figures 5 and 6 show some other operators from which filter conditions can be constructed. Both tables are taken from the W3C specification at http://www.w3.org/TR/rdf-sparql-query/, which can be consulted for further details.

Conceptually, filters define a boolean condition on a graph pattern binding. We have already seen that the graph pattern in a WHERE clause has a set of solutions, each corresponding to a binding of the variables mentioned in the graph pattern. The filter submits these solutions to a boolean condition, letting through only those variable bindings for which the condition is met. In a naive implementation, the projection would be computed in just this way: first find the set of solutions matching the graph pattern; then select from this set the solutions that satisfy the filter condition, for inclusion in the final result.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type of A</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>*/,+,-</td>
<td>numeric</td>
<td>numeric</td>
</tr>
<tr>
<td>BOUND(A)</td>
<td>Variable</td>
<td>xsd:boolean</td>
</tr>
<tr>
<td>isURI(A)</td>
<td>RDF term</td>
<td>xsd:boolean</td>
</tr>
<tr>
<td>isBLANK(A)</td>
<td>RDF term</td>
<td>xsd:boolean</td>
</tr>
<tr>
<td>isLITERAL(A)</td>
<td>RDF term</td>
<td>xsd:boolean</td>
</tr>
<tr>
<td>STR(A)</td>
<td>literal / URI</td>
<td>simple literal</td>
</tr>
<tr>
<td>LANG(A)</td>
<td>literal</td>
<td>simple literal</td>
</tr>
<tr>
<td>DATATYPE(A)</td>
<td>literal</td>
<td>simple literal</td>
</tr>
</tbody>
</table>

Figure 5: Unary operators for filter expressions

<table>
<thead>
<tr>
<th>Type of operator</th>
<th>Operator</th>
<th>Data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>And (&amp;&amp;), Or (</td>
<td></td>
</tr>
<tr>
<td>Comparison</td>
<td>Equal (=), Not equal (!=), Less than (&lt;), Greater than (&gt;)</td>
<td>Numeric datatypes: xsd:dateTime, xsd:string, xsd:boolean</td>
</tr>
<tr>
<td></td>
<td>Less than or equal (&lt;=), Greater than or equal (&gt;=)</td>
<td>xsd:boolean</td>
</tr>
<tr>
<td>Comparison</td>
<td>Equal (=), Not equal (!=)</td>
<td>Other datatypes</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>+, -, *, /</td>
<td>Numeric datatypes</td>
</tr>
</tbody>
</table>

Figure 6: Binary operators for filter expressions

**Avoiding duplicate rows in the output table**  For some queries you may find that the output table has duplicated rows. The reason for this is usually that the selected (tabulated) variables are a strict subset of the variables in the
graph pattern. Consider for instance the following table, which you will see if you submit the previous query:

<table>
<thead>
<tr>
<th>title</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;All You Need Is Love&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
<td>300000^^<a href="http://www.w3.org/2001/XMLSchema#int">http://www.w3.org/2001/XMLSchema#int</a></td>
</tr>
<tr>
<td>&quot;Misery&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
<td>300293^^<a href="http://www.w3.org/2001/XMLSchema#int">http://www.w3.org/2001/XMLSchema#int</a></td>
</tr>
<tr>
<td>&quot;[improvisation 5]&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
<td>301080^^<a href="http://www.w3.org/2001/XMLSchema#int">http://www.w3.org/2001/XMLSchema#int</a></td>
</tr>
<tr>
<td>&quot;Hey Jude&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
<td>302040^^<a href="http://www.w3.org/2001/XMLSchema#int">http://www.w3.org/2001/XMLSchema#int</a></td>
</tr>
<tr>
<td>&quot;I Lost My Little Girl&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
<td>302666^^<a href="http://www.w3.org/2001/XMLSchema#int">http://www.w3.org/2001/XMLSchema#int</a></td>
</tr>
<tr>
<td>&quot;[Think for Yourself studio chatter]&quot;^^<a href="http://www.w3.org/2001/XMLSchema#string">http://www.w3.org/2001/XMLSchema#string</a></td>
<td>303000^^<a href="http://www.w3.org/2001/XMLSchema#int">http://www.w3.org/2001/XMLSchema#int</a></td>
</tr>
</tbody>
</table>

In the begin of this table we find two rows with track title “All You Need Is Love (live on “Our World” TV Show: 1967-06-25)” and duration 300293; and there are many more examples further down the table. This happens because there might be multiple resources instantiating ?track having the same values for ?title and ?duration. Here, for example, the track “All You Need Is Love (live on “Our World” TV Show: 1967-06-25)” is present in two different albums, so it shows up twice. If all three variables were tabulated, the rows would differ in the ?track column, but since this column is not requested, we obtain rows that appear identical. To avoid this you can include the keyword DISTINCT in the SELECT clause, as follows:

```sparql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT DISTINCT ?title ?duration
WHERE {
  ?track a mo:Track .
  FILTER (?duration>300000 && ?duration<400000)
}
ORDER BY ?duration LIMIT 10
```

The output table, you should now find only one row pairing “All You Need Is Love (live on “Our World” TV Show: 1967-06-25)” with 300293.

Since DISTINCT is computationally expensive there is an efficient alternative REDUCED which eliminates some duplicates but not necessarily all (e.g., it fails to eliminate the duplication of “Within You Without You” mentioned above); however, DISTINCT is more widely used, and should not be computationally expensive when ORDER BY is also used.

**Retrieving aggregate data for groups of bindings** Suppose that the dataset contains triples specifying the tracks on each album, and associating...
a duration in milliseconds with each track, represented by an integer literal. (In fact the MusicBrainz dataset associates records with albums, and tracks with records, which complicates the query slightly – see below.) Fully listed this is a lot of data, and you might wish instead to report, for each album, the total duration obtained by summing the durations of the tracks. This is an example of aggregate data, and it requires two operations: first, we must segment the variable bindings into groups, corresponding in this case to all bindings relating to a given album; second, for each group, we must submit the values of a specified variable to an aggregation function – in this case, sum the track durations.

For instance, suppose the dataset has just two albums matching the query, namely Revolver and Abbey Road; and suppose that for Revolver just three tracks are included, namely “Eleanor Rigby”, “I’m only sleeping”, and “Doctor Robert”, with durations respectively of 200000, 240000, and 160000 milliseconds. (These are just round numbers made up for the example.) This will mean that for Revolver we have three bindings of the variables ?album, ?track, and ?track_duration; and if Abbey Road has four specified tracks, it will correspondingly have four bindings. We now separate the bindings into groups – three for Revolver, four for Abbey Road – and within each group we want to sum the track durations, and return only a table that gives albums along with their total durations. This will mean that the first row of the table will specify Revolver in the ?album column, and 600000 in the second column for which we need a new name – perhaps ?album_duration.

Here is a query that achieves this result. In addition it imposes a condition on the total duration of the album, reporting only albums with duration exceeding 3600000 milliseconds (i.e., one hour), selecting (and hence grouping by) album title rather than album.

```sql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX mo: <http://purl.org/ontology/mo/>

SELECT ?album_title (SUM(?track_duration) AS ?album_duration)
WHERE {
}
GROUP BY ?album_title
HAVING (SUM(?track_duration) > 3600000)
ORDER BY ?album_duration
LIMIT 10
```

Note two new keywords here: AS introducing a variable name for the sum of durations (this becomes the heading of the second column of the output table); and HAVING introducing a filter over the group of variable bindings that has just been specified by the GROUP BY component.
Queries using CONSTRUCT When building an application, you might need to retrieve some information from a queried dataset and re-express it in new RDF triples, perhaps using new names for resources. This might, for example, allow more efficient integration with triples from another dataset.

To meet this need, SPARQL provides a CONSTRUCT query which uses information retrieved from a dataset in order to build new RDF statements. Note that the query does not update the dataset. The new RDF triples are returned to the user as output, to be used in any way desired, the dataset itself remaining unchanged. (In SPARQL 1.1 there are query types for updating a dataset, as described in a later section of this chapter.)

To show the basic syntax of a CONSTRUCT query, consider the following example where the user wishes to assert “creator” relationships between artists and their products – of any kind; perhaps the aim is to construct a dataset in which this predicate is used consistently, replacing more specific predicates in MusicBrainz.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX mo: <http://purl.org/ontology/mo/>

}
}
```

The key to understanding this query is that the variables employed in the CONSTRUCT list must occur also in the WHERE list. When the query is run, the query engine begins by retrieving the variable bindings satisfying the description in the WHERE list – just as it would for a SELECT query. For each variable binding, it then instantiates the triple patterns in the CONSTRUCT list and so creates (in this case two) new RDF triples. The result of the query is a merged graph including all the created triples.

Figure 8 shows this outcome diagrammatically for an even simpler CONSTRUCT query, with the relevant part of the dataset and the constructed triples both shown as graphs.
Figure 8: Result of a simple CONSTRUCT query

**Ordering and limiting the constructed triples** When building new RDF triples with CONSTRUCT, you can use the operators described in section 2.4.4 in order to organise and delimit the variable bindings retrieved from the dataset. The following query will build “creator” relationships only for the first 10 albums recorded by the Beatles, following alphabetical order of the titles, along with their tracks:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX mo: <http://purl.org/ontology/mo/>

} ORDER BY ?album_title
LIMIT 10
```

**Query patterns containing disjunction** Suppose that for some reason you want to construct triples for albums made *either* by the Beatles *or* by the Smashing Pumpkins (or both). Including both of these constraints in the WHERE list will not work, because implicitly the list represents a conjunction of statements, each of which must be satisfied. To allow disjunctions, SPARQL contains a UNION pattern; this is formed by placing the keyword UNION between two subsets of statements, each subset delimited by curly brackets. The meaning is that variable bindings should be retrieved if they satisfy either the statements on the left or the statements on the right (or both). Thus our target query is formed as follows:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX mo: <http://purl.org/ontology/mo/>

          UNION
          }
} ORDER BY ?album_title
LIMIT 10
```
Note that the first three statements of the WHERE list lie outside the scope of the UNION operator.

**Retrieving resources for which information is MISSING from the dataset**

In the examples we have seen so far, variable bindings must be retrieved for all patterns listed after WHERE. This means that if we retrieve several facts about an album (say), the album will only be included in the output if all these facts are presented in the dataset: if just one is missing, the others will be ignored. SPARQL deals with this problem by allowing any graph pattern in the list to be preceded by the keyword OPTIONAL. This means that when computing variable bindings, the query engine should accept incomplete bindings provided that the unspecified variables occur only in optional patterns.

In the following query, optional patterns are used ingeniously to select only variable bindings for which a particular variable is not bound. The variable in question records an artist’s place of death, and it is assumed that if this information is missing from the dataset, the artist will still be alive. If variables in the CONSTRUCT clause are not bound in the OPTIONAL clause, the triple patterns with these variables are not generated. As a result, “creator” relationships are constructed only for artists who are alive (or more precisely, artists for whom there is no death place recorded in the dataset).

Note that in the filter expression ‘!’ denotes negation, so that the whole expression means that the variable is not bound.
You should take care using this kind of query, since it depends on a risky inference sometimes called the closed-world assumption — namely, that any relevant statement not found in the dataset must be false. Thus if the dataset contains information about places of death, but no statement giving the place of death of Paul McCartney, we infer by this assumption that Paul McCartney must still be alive, since otherwise his place of death would have been recorded.

**Assigning variables**  If you want to construct RDF triples using a variable that is derived from retrieved data, e.g., through an arithmetical operation, you can add a BIND statement to the WHERE clause as follows:

```sparql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX mo: <http://purl.org/ontology/mo/>

CONSTRUCT { ?track mo:runtime ?secs }
  BIND ((?duration/1000) AS ?secs) .
} LIMIT 10
```

In this way the object of mo:runtime will be given in seconds rather than milliseconds.

**Constructing new triples using aggregate data**  We have already discussed a SELECT query that returns aggregate data by summing the durations of tracks in each album. You may recall that such a query uses the AS keyword in the expression following SELECT, to introduce a variable name for the aggregate value — in this case, the album duration. In the context of a CONSTRUCT query we therefore have a problem: how to introduce this new variable for the aggregate?

The solution used in SPARQL is to allow a sub-query after the keyword WHERE, in place of the usual graph pattern. This is achieved by the following rather convoluted syntax:

```sparql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX mo: <http://purl.org/ontology/mo/>

CONSTRUCT { ?album mo:duration ?album_duration }
WHERE {
  SELECT ?album (SUM(?track_duration) AS ?album_duration )
  } GROUP BY ?album
  HAVING (SUM(?track_duration) > 3600000)
}
```
Queries using DESCRIBE Like CONSTRUCT, DESCRIBE delivers as output an RDF graph – i.e., a set of RDF triples. It differs from CONSTRUCT in that these triples are not constructed according to a template, but returned as found in the dataset. The reasons for doing this are similar to those for CONSTRUCT – you might, for instance, want to add these triples to another dataset – but you would prefer DESCRIBE if you were satisfied with the original encoding and had no reason to re-express the information using different resource names.

Resources can be specified more generically as bindings to a variable. Thus the following query requests all triples that mention for an album.

```
PREFIX mo: <http://purl.org/ontology/mo/>
DESCRIBE ?track WHERE { ?track a mo:Track } LIMIT 1
```
when the Beatles were at their most active and famous, then this information would be out-of-date; we might want to describe Peter Best as a former band member, but not a current one. Here is a query that will achieve the first part of this objective, by describing Peter Best as a former band member:

```sql
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX db-ont: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

INSERT { dbpedia:The_Beatles db-ont:formerMember ?x }
WHERE { dbpedia:The_Beatles db-ont:currentMember ?x .
    ?x foaf:name "Peter Best"
}
```

This query is very similar in syntax to the queries we examined in section 2.4, especially CONSTRUCT and DESCRIBE queries. The query form in this case is INSERT, and like CONSTRUCT it is followed by a graph pattern that specifies a set of RDF triples, given bindings for any variables that are included in the pattern. These bindings, as before, are obtained through a WHERE clause. In this simple example, we assume that the person formulating the query does not know the IRI naming the resource Peter Best, and accordingly retrieves it from the dataset using the foaf:name relation to the literal “Peter Best”. Once the relevant IRI is plugged into the single triple following INSERT, we obtain an exact specification of an RDF triple (with no variables). The query engine then checks whether this triple is present in the default graph, and if not, adds it. If the triple was already present in the graph, there would be no change.

If you knew the URI for Peter Best, you could use an alternative form of the insert query with the keywords INSERT DATA. This is followed not by a graph pattern (including variables) but a set of fully specified triples; accordingly, a WHERE clause is not needed.

If you wish the triples to be added to a named graph rather than the default graph, you can name the relevant graph at the beginning of the pattern after INSERT, as in the following slightly modified example:

```sql
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX db-ont: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

INSERT { GRAPH <http://myFavGroups/The_Beatles>
    { dbpedia:The_Beatles db-ont:formerMember ?x }
    }
WHERE { dbpedia:The_Beatles db-ont:currentMember ?x .
    ?x foaf:name "Peter Best"
}
```

Deleting data from a graph As in the case of inserting, you have two options: either use DELETE DATA followed by a set of RDF triples (without variables), or use DELETE followed by a graph pattern and a WHERE clause. In either case, you again have the option of indicating, within the DELETE clause,
the graph from which the specified triples will be removed. If no graph is named, the triples will be removed from the default graph.

In the following example, which would be a natural sequel to the query in 2.6.1, Peter Best is removed as a current band member from the default graph.

```sparql
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX db-ont: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

DELETE { dbpedia:The_Beatles db-ont:currentMember ?x }
WHERE { dbpedia:The_Beatles db-ont:currentMember ?x .
  ?x foaf:name "Peter Best"
}
```

Combining insertion and deletion  In the examples for INSERT and DELETE the aim was to replace an outdated statement (Peter Best as current member) by an up-to-date one (Peter Best as former member). When the INSERT and DELETE operations use the same variable bindings, they can be combined in a single query, sharing the same WHERE clause, as follows:

```sparql
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX db-ont: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

DELETE { dbpedia:The_Beatles db-ont:currentMember ?x }
INSERT { dbpedia:The_Beatles db-ont:formerMember ?x }
WHERE { dbpedia:The_Beatles db-ont:currentMember ?x .
  ?x foaf:name "Peter Best"
}
```

Loading an RDF graph  You might wish to extend your dataset by loading triples from an external dataset for which you have the URL. This can be done using the LOAD query, which has the following simple basic syntax:

```sparql
LOAD <http://xmlns.com/foaf/spec/20100809.rdf>
```

This will retrieve all triples from the specified external web source – if it exists – and copy them into the default graph of your query engine. If you want instead to copy them into a named graph you can use the keyword INTO to specify the graph name:

```sparql
LOAD <http://xmlns.com/foaf/spec/20100809.rdf>
INTO <http://myFavGroups/The_Beatles>
```

Clearing an RDF graph  If you want to retain a graph in the store as a location into which you can load and insert data, but to repopulate it from scratch by clearing out its current triples, you can use a CLEAR query with the following simple syntax:
The keyword GRAPH is employed when you identify the graph to be cleared using its IRI. Alternatives are CLEAR DEFAULT, which clears the default graph, CLEAR NAMED, which clears the graph with the specified name, and CLEAR ALL, which clears all graphs in the store. Thus the following would be a complete query removing all triples from the query engine’s default graph:

```
CLEAR DEFAULT
```

Adding and removing graphs  SPARQL 1.1 allows you to manage the graphs in the query engine’s store by the query forms CREATE and DROP. It is important to understand that these operate on graphs, not on triples. CLEAR removes all triples from a graph, but does not remove the graph itself; it remains in place as a location into which triples may be inserted or loaded. DROP, by contrast, removes the graph altogether from the store, so that any attempt to load or insert into it will yield an error – unless you first reinstate it using CREATE. Thus CLEAR is like removing all text from a Word document while leaving the document in place, whereas DROP is like trashing the document altogether.

To create a new graph, just specify its IRI as follows:

```
CREATE GRAPH <http://myFavGroups/The_Beatles>
```

To drop an existing graph, use the same syntax with DROP instead of CREATE:

```
DROP GRAPH <http://myFavGroups/The_Beatles>
```

Other graph management operations  If you are working with several graphs in a store, you might wish to copy or move information from one to another. For instance, when starting a new version of a named graph, you might first create a new (empty) graph for version 2, and then wish to copy across the entire contents of version 1. This is done as follows:

```
COPY GRAPH <http://myFavGroups/The_Beatles/v1>
TO GRAPH <http://myFavGroups/The_Beatles/v2>
```

Note that version 1 remains unchanged by this operation, which is like copying text from one document and pasting it into another document, *overwriting any text already in the second document*. This means that after the COPY operation is performed, the two graphs will have exactly the same triples.

If the source graph is a draft that you have no more use for, you can use the MOVE command, which first copies all triples from the first graph to the second, and then removes the first graph altogether – equivalent to performing COPY followed by DROP:

```
MOVE GRAPH <http://myFavGroups/The_Beatles/draft>
TO GRAPH <http://myFavGroups/The_Beatles/v1>
```
This is like copying text from one document to another, overwriting any text in the second document, and then trashing the first document.

If you want to retain triples already present in the destination graph, you can use the ADD command, which is the same as COPY except that triples from the source graph are added to the destination graph without overwriting. You might do this, for example, if you have been constructing separate graphs for different rock groups (The Beatles, The Rolling Stones, etc.), and now wish to add your Beatles data to a larger graph covering all your favourite rock groups:

```
ADD GRAPH <http://myFavGroups/The_Beatles>
TO GRAPH <http://myRockGroups>
```

Like COPY, this operation leaves the source graph unchanged. It is like copying text from a document and pasting it into another document without overwriting the text already present there.

For all these operations, a destination graph will be created automatically if it does not exist already.

**SPARQL protocol**

The SPARQL 1.1 Protocol comprises two operations, one for querying, one for updating. An operation specifies the following:

- An HTTP method (either GET or POST)
- HTTP query parameters
- Message content in the HTTP request body
- Message content in the HTTP response body

A request is sent from the SPARQL client to the SPARQL server (or endpoint), which sends back a status code, possibly accompanied by other data (e.g., tables with the results of the query). For a query operation, the response is a table of data in one of various formats (XML, CSV, etc.) for a SELECT query, and triples formatted in RDF/XML or Turtle for a CONSTRUCT query. Status codes are three-digit integers whose meanings are defined by the HTTP specification; for instance, the standard response to a successful request has the code “200”.

Figure 9 shows a possible encoding of the HTTP GET request for a simple query. The details are not important for this chapter, but it is worth noting the final line (starting with “Accept”), which lists the media types that the sender can understand, with preference ratings expressed as numbers between 0.0 and 1.0. This is called content negotiation, and is also part of the HTTP specification.
Reasoning over linked data

Reasoning enhances the information contained in a dataset by including results obtained by inference from the triples already present. As a simple example, suppose that the dataset includes the following triples, shown here in Turtle (assume the usual prefixes):

\[
\begin{align*}
\text{dbpedia:} & \text{The_Beatles a mo:MusicGroup .} \\
\text{mo:MusicGroup rdfs:subClassOf mo:MusicArtist .}
\end{align*}
\]

Recall that the predicate a in the first triple is Turtle shorthand for rdf:type, denoting class membership. Now, suppose that we submit the following SELECT query designed to retrieve all triples in which the Beatles occur as subject:

\[
\begin{align*}
\text{PREFIX dbpedia: <http://dbpedia.org/resource/>} \\
\text{SELECT *} \\
\text{WHERE \{ dbpedia:The_Beatles ?predicate ?object \}}
\end{align*}
\]

Assuming there are no other relevant triples, the response to this query under a regime with no entailment will be the following single-row table:

\[
\begin{array}{cc}
\text{?predicate} & \text{?object} \\
a & \text{mo:MusicGroup}
\end{array}
\]

An obvious inference has been missed here, since if every music group is a music artist (as asserted by the second triple), the Beatles will also be a music artist. (It might sound odd to call a group an artist, but this is how the information is encoded in the Music Ontology.) If we execute the query on an engine that implements the RDFS entailment regime (a set of rules governing inference based on RDFS resources like rdfs:subClassOf), the output table will be enriched by a second variable binding inferred with the aid of the second triple.

\[
\begin{array}{cc}
\text{?predicate} & \text{?object} \\
a & \text{mo:MusicGroup} \\
a & \text{mo:MusicArtist}
\end{array}
\]
This is a very simple case, but in general the formulation of workable entailment regimes is a complex task, still under investigation and discussion. One hard problem is what to do if the dataset is inconsistent, since a well-known result of logic states that from a contradiction, anything can be inferred. (This follows from the definition of material implication, for which the truth-conditions state that if the antecedent is false, the implication holds whatever the consequent. Hence a contradiction, being false by definition, implies any statement whatever.)

Another problem is that some queries might return an infinite set of solutions. For instance, there is a convention in RDF that allows for a family of predicates rdf:_1, rdf:_2, etc. ending in any positive integer; such predicates are used when constructing lists. This means that if all possible inferences are performed, a query of the form ?x rdf:type rdf:Property (intuitively, return all properties) would yield an infinite set.

For practical purposes we can ignore such cases, but they illustrate that that query engines may not always return all the entailments that one might expect.

In the rest of this section we look first at typical entailments that arise from RDFS, and then at the much richer entailments that can result from an ontology in OWL.

**Reasoning using RDFS** RDF Schema introduces a very limited range of logical terms based mostly on the concept of class, or rdfs:Class, a concept absent from RDF. It is possible to state directly that a resource denotes a class, using a triple such as the following (to save space we omit prefixes):

```
mo:MusicGroup rdf:type rdfs:Class .
```

We have already seen an example of the predicate rdfs:subClassOf, through which we can assert that the resource in subject position is a subclass of the resource in object position – a relationship often expressed in English by sentences of the form *All X are Y* (e.g., “All pianists are musicians”). A similar predicate is provided for properties rather than classes, illustrated by the following triple in which subject, predicate and object are all property resources:

```
ex:hasSinger rdfs:subPropertyOf ex:hasPerformer .
```

This means that if a subject and object are related by the ex:hasSinger predicate, they must also be related by the ex:hasPerformer predicate. Note that even though these are properties, they may occur in subject and object position – in other words, we can make statements about properties, as well as using them to express relationships among other resources. Two important facts about any property are its *domain* and *range*, which constrain the resources that can be employed in subject and object position when the property is used as a predicate; these can also be defined using RDFS resources, as follows:

```
ex:hasSinger rdfs:domain mo:Performance .
ex:hasSinger rdfs:range foaf:Agent .
```

This means that any resource occurring as subject of ex:hasSinger must belong to the class mo:Performance (only performances have singers), while any resource
occurring as object of ex:hasSinger must belong to the class foaf:Agent (only agents are singers).

**RDFS entailment rules**  The RDFS entailment regime is defined by thirteen rules, which are listed here. In brief, their import is as follows:

- **rdfs1** Allocates a blank node to a literal, and classifies this node as a member of the class rdfs:Literal.
- **rdfs2** Uses a rdfs:domain statement to classify a resource found in subject position (in a triple containing the relevant property as predicate).
- **rdfs3** Uses a rdfs:range statement to classify a resource found in object position (in a triple containing the relevant property as predicate).
- **rdfs4** Classifies a resource found in subject or object position in a triple as belonging to the class rdfs:Resource.
- **rdfs5** Inference based on the transitivity of rdfs:subPropertyOf.
- **rdfs6** Infers that any resource classified as a property is a sub-property of itself.
- **rdfs7** Uses a rdfs:subPropertyOf statement to infer that two resources are related by a property P if they are related by a subproperty of P.
- **rdfs8** Infers that any resource classified as a class is a subclass of rdfs:Resource.
- **rdfs9** Infers that a resource belongs to a class if it belongs to its subclass.
- **rdfs10** Infers that any class is a subclass of itself.
- **rdfs11** Uses the transitivity of rdfs:subClassOf to infer that the subclass of a class C is a subclass of the superclass of C.
- **rdfs12** Infers that any resource belonging to rdfs:ContainerMembershipProperty is a subproperty of rdfs:member.
- **rdfs13** Infers that any resource belonging to rdfs:Datatype is a subclass of rdfs:Literal.

In what follows, we look more closely at how these RDFS resources and their associated rules can support inferences when information is retrieved from a dataset.

**Inferring subclass and class membership relationships**  We have already seen an example of an inference based on rdfs:subClassOf, which we repeat below omitting Prefix statements:

```sparql
dbpedia:The_Beatles a mo:MusicGroup .
mo:MusicGroup rdfs:subClassOf mo:MusicArtist .
```

The general form of this inference is that if a resource belongs to a class, which in turn belongs to a broader class, then the resource must also belong to this broader class (see rdfs9). This pattern will be familiar as a variant of the Aristotelian syllogism:
Socrates is a man.

All men are mortal.

Therefore, Socrates is mortal.

A similar inference can be drawn from two subclass statements in which the object of one statement is the subject of the other (see rdfs11):

\[
\begin{align*}
\text{mo:MusicGroup} & \text{ rdfs:subClassOf } \text{mo:MusicArtist} . \\
\text{mo:MusicArtist} & \text{ rdfs:subClassOf } \text{foaf:Agent} .
\end{align*}
\]

If music groups are a subclass of music artists, who are in turn a subclass of agents, then music groups must be a subclass of agents. This corresponds to the so-called Barbara pattern of the syllogism:

All Greeks are men.

All men are mortal.

Therefore, all Greeks are mortal.

**Classifying a resource from its relationships to other resources**

Suppose a dataset contains the following triple asserting that Paul McCartney belongs to the Beatles:

\[
\text{dbpedia:The_Beatles} \text{ mo:bandMember } \text{dbpedia:Paul_McCartney} .
\]

Can we infer anything further about the resources in subject and object position? To a human, of course, it will be obvious from the resource names that the Beatles is a group, and that Paul McCartney is an agent, or more specifically a person. However, these names are meaningless to the query engine, which can only utilise information encoded in RDF triples. To put ourselves in this position, we could try using arbitrary IRIs, as in this version of the triple:

\[
\text{ex:X} \text{ mo:bandMember } \text{ex:Y} .
\]

Even with subject and object anonymised in this way, a person could infer from the predicate that X will be a band, and Y a person. The query engine can perform this inference too if the dataset includes triples defining the domain and range of the property (see rdfs2 and rdfs3):

\[
\begin{align*}
\text{mo:bandMember} & \text{ rdfs:domain } \text{mo:MusicGroup} ; \\
& \text{ rdfs:range } \text{foaf:Agent} .
\end{align*}
\]

( Remember the use of the semi-colon here to abbreviate two statements with the same subject.) If these triples are present in the dataset, and the query engine is deploying the RDFS entailment regime, then two further triples can be directly inferred:

\[
\begin{align*}
ex:X & \text{ a mo:MusicGroup} . \\
ex:Y & \text{ a foaf:Agent} .
\end{align*}
\]
A special case of this inference arises when a triple uses an RDFS predicate for which domain and range statements are axiomatic, meaning that they are defined in RDFS itself and need not be included in the dataset over which the query is run. Thus for rdfs:subClassOf, the following triples are axiomatic and need not be defined by developers:

```
    rdfs:subClassOf rdfs:domain rdfs:Class .
    rdfs:subClassOf rdfs:range rdfs:Class .
```

It follows that on encountering any subclass statement, the query engine can infer that its subject and object must be classes. Thus from –

```
    mo:MusicGroup rdfs:subClassOf mo:MusicArtist .
```

– it can infer:

```
    mo:MusicGroup a rdfs:Class .
    mo:MusicArtist a rdfs:Class .
```

**Inferences based on sub-property relationships**  Finally, whenever SP has been defined as a sub-property of P, it can be inferred that any subject and object having an SP relationship must also have the (broader) P relationship (see rdfs5):

```
    ex:hasSinger rdfs:subPropertyOf ex:hasPerformer .
    ex:Yesterday ex:hasSinger dbpedia:PaulMcCartney .
```

From these two statements, the query engine may infer:

```
    ex:Yesterday ex:hasPerformer dbpedia:PaulMcCartney .
```

Such inferences are common when a dataset uses RDFS to organise classes and properties into hierarchies. Thus classes like Corgi, Dog, Canine, Mammal, Animal, Living thing, comprise an obvious hierarchy in which membership of any class implies membership of all its superclasses; as we have just shown, a similar hierarchical organisation can be defined for properties.

**More advanced inferences**  Compared with OWL, RDFS has two main limitations: (a) it provides no operators for constructing complex classes or properties out of simpler ones (e.g., “artist that belongs to at least two bands” from artist, band, and member-of); (b) it lacks some important resources for describing the logical properties of classes and properties, such as disjointness (for classes) and inverse (for properties). Limitations of the second kind are particularly common when working with linked data, and it is worth illustrating some of them now, before we look in detail at OWL.

Suppose that a dataset contains resources named ex:Man and Woman, both classes, which can be used for classifying resources that represent individual people:

```
    dbpedia:Paul_McCartney a ex:Man .
    dbpedia:Cilla_Black a ex:Woman .
```
Now, suppose that in addition to these two triples, a third triple is either present in the dataset, or can be inferred:

```sparql
dbpedia:Paul_McCartney a ex:Woman .
```

Plainly something has gone wrong here, and we would like a query engine capable of even elementary reasoning to signal an inconsistency. It might surprise you the learn that there is no way of doing this, because RDFS provides no predicate allowing you to state that men and women are disjoint (i.e., that no man is a woman, or nothing is both a man and a woman). The necessary predicate, owl:disjointWith, is found in OWL only, and allows statements such as the following:

```sparql
ex:Man owl:disjointWith ex:Woman .
```

To give one more example, suppose that we introduce a property resource allowing us to state that one person is married to another. We might for instance apply it to the McCartneys:

```sparql
```

Now, suppose that someone formulates the following query, corresponding to the question “Who is married to Paul McCartney?” (we omit PREFIX clauses as before):

```sparql
SELECT ?person WHERE { ?person ex:marriedTo dbpedia:Paul_McCartney }
```

Obviously the answer should be “Linda McCartney”, but this binding will not be found because the following triple is missing from the dataset:

```sparql
```

To infer this we need a statement to the effect that if X is married to Y, Y must be married to X. In mathematics, this fact is expressed by saying that the property ex:marriedTo is symmetric. This can be stated directly using OWL, as follows:

```sparql
ex:marriedTo a owl:SymmetricProperty .
```

No such resource exists in RDFS, so this kind of inference cannot be performed under an RDFS entailment regime.

**Reasoning using OWL** As a reminder, Figure 10 shows the location of the Web Ontology Language (OWL) in the Semantic Web stack. As can be seen, OWL depends on RDF and RDFS, from which it draws crucial resources such as rdf:type and rdfs:Class. Like RDFS, it can enhance the information in RDF datasets, accessible to SPARQL queries.
The current OWL standard OWL-2 is complex, providing for a number of fragments with different computational properties (see chapter 1). Most of these are subsets of OWL2-DL, the OWL description logic, but there is also a more expressive variant called OWL2-Full which ventures outside description logic. We will not try to cover all this material in this section. Instead, we look at some of the main logical resources in OWL2-DL from the viewpoint of their role in inference.

**Inferences based on characteristics of properties** We have illustrated in the last section an inference based on one of the property characteristics that can be represented in OWL, namely symmetry (if X p Y, then Y p X). A number of others are provided, of which the following are used most often:

- **Transitive** properties support the inference: if X p Y, and Y p Z, then X p Z. An example is the property “longer than” applied to track durations: if “Hey Jude” is longer than “Help”, and “Help” is longer than “Yesterday”, then we can infer that “Hey Jude” is longer than “Yesterday”.

- **Functional** properties support the inference that if X p Y and X p Z, Y and Z must be identical: in other words, X can bear the relation p to only one thing. An example is the property “has mother”, assuming that this means biological mother. To say that this property is functional means that a person can have at most one mother. Thus if Charles has Elizabeth as mother, and Charles also has Mrs Windsor as mother, then Elizabeth and Mrs Windsor are two names of the same person.

- **Inverse-functional** properties support the inference that if Y p X and Z p X, then Y and Z must be identical. An example would be the property “is mother of”. Thus if Elizabeth is mother of Charles, and so is Mrs Windsor, then Elizabeth and Mrs Windsor are the same person.

- Two properties p1 and p2 are **inverse** if X p1 Y means exactly the same as Y p2 X (i.e., they are equivalent, and each can be inferred from the other). Thus in our last two examples, “has mother” and “is mother of” are inverse properties, since if Charles has mother Elizabeth, then Elizabeth is mother of Charles, and vice-versa.

Let us see how these characteristics are used in practice, starting with a transitive property.
Here the classification of ex:locatedIn as a transitive property allows a query engine with an OWL2-DL regime to draw the following inference in response to a query concerning the location of the Abbey Road Studio (or concerning what is located in the United Kingdom):

```
ex:AbbeyRoadStudios ex:locatedIn ex:UnitedKingdom .
```

Moving on to functional properties, consider this set of triples:

```
ex:hasFather a owl:FunctionalProperty .
dbpedia:Julian_Lennon ex:hasFather dbpedia:John_Lennon .
dbpedia:Julian_Lennon ex:hasFather ex:J_Lennon .
```

The classification of ex:hasFather as functional permits the inference that Julian Lennon can have only one father, and hence that the objects of the second and third triples must be co-referential. In OWL this is expressed using owl:sameAs:

```
dbpedia:John_Lennon owl:sameAs ex:J_Lennon .
```

Exactly the same inference can be drawn if the same information is presented using the inverse-functional property ex:isFatherOf:

```
ex:isFatherOf a owl:InverseFunctionalProperty .
dbpedia:John_Lennon ex:isFatherOf dbpedia:Julian_Lennon .
ex:J_Lennon ex:isFatherOf dbpedia:Julian_Lennon .
```

Note that owl:sameAs is symmetric, and also transitive; these characteristics are axiomatic in OWL, so they need not be specified by the ontology developer.

Finally, if we include a triple stating that ex:hasFather and ex:isFatherOf are inverse, we obtain two ways of expressing fatherhood, with subject and object switched. Imagine for instance a dataset with just two triples –

```
ex:hasFather owl:inverseOf ex:isFatherOf .
dbpedia:Julian_Lennon ex:hasFather dbpedia:John_Lennon .
```

– and suppose we run the following query (prefixes omitted) on who is the father of Julian Lennon:

```
SELECT ?person
WHERE { ?person ex:isFatherOf dbpedia:Julian_Lennon }
```

Without OWL-based reasoning the query engine has no way of returning John Lennon as an answer to this query, but using the inverse property statement it can derive the following entailment:

```
dbpedia:John_Lennon ex:isFatherOf dbpedia:Julian_Lennon .
```
Inferences based on equivalence among terms  We have already seen an OWL property owl:sameAs which allows us to state that two IRIs name the same individual:

```
dbpedia:John_Lennon owl:sameAs ex:J_Lennon .
```

This property is very important in linked data, since it provides a means by which datasets with different naming conventions can be connected. OWL also provides predicates for stating that two classes, or two properties, have the same meaning, as follows:

```
mo:MusicArtist owl:equivalentClass ex:musician .
foaf:made owl:equivalentProperty ex:creatorOf .
```

In essence we have three ways here of stating that two terms mean the same thing, and in each case inferences can be drawn from the fact that equivalence of meaning is symmetric and transitive. The practical consequence is that if a dataset already contains two names for John Lennon (say), owing to the owl:sameAs statement above, we can add a third name by relating it to only one of these names –

```
dbpedia:John_Lennon owl:sameAs new:JL666 .
```

– whereupon by symmetry and transitivity, a similar statement can be inferred for any subject-object pair drawn from these three names:

```
ex:J_Lennon owl:sameAs new:JL666 .
new:JL666 owl:sameAs ex:J_Lennon .
new:JL666 owl:sameAs dbpedia:John_Lennon .
```

Similarly, a single owl:sameAs statement introducing a fourth name will suffice to make it equivalent in meaning to any of the other three – and so on.
Chapter 3
Providing Linked Data

3.1 Introduction

In the previous chapter we described how SPARQL is used to formulate queries and retrieve information from a dataset. In this chapter we look at how such datasets can be created and made available in the first place. We will describe how data can be extracted from different sources such as texts and databases and then represented as linked data using the RDF data model. We also show how relationships can be used to express connections between datasets as well define concepts used in a dataset in terms of a different vocabulary. We will also describe the steps involved in making the dataset accessible and discoverable.

In this chapter we once again use MusicBrainz as a motivating example. MusicBrainz is used to show how different music data sources, stored in for example, databases and texts can be prepared, matched to the MusicBrainz schema and made available as a linked data source.

The final sections provide a practical introduction to a number of tools that assist in the process of providing linked data.

3.2 The Linked Data Lifecycle

The process of providing linked data is often characterised as a Linked Data lifecycle in which data is created, reusing other data sources, prepared and then made accessible for use. The new data can then become one of a number of data sources used in the preparation of further datasets. This illustrates the cyclic nature of linked data creation and publishing. Before looking in more detail at different perspectives on the stages of the linked data lifecycle, we will recap the four Linked Data Principles introduced in Chapter 1. In summary, these principles are as follows:

1) Use URIs as names for things.

2) Use HTTP URIs so that users can look up those names.

3) When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL).

4) Include links to other URIs, so that users can discover more things.
As described in Chapter 1, a clear motivation for these principles is to publish “things” as URIs in a way that is accessible (principle 2), useful (principle 3) and connects to other published things (principle 3). Essentially this means publishing data in a way that enhances its practical value and facilitates the use of that dataset as input to other datasets. This therefore creates a cycle in which data is created, published and then used in the creation of new data.

Different ways of understanding the linked data lifecycle and its constituent steps have been proposed. Sören Auer [1] proposes an eight stage linked data lifecycle. This involves searching or finding sources from which data can be extracted and stored, interconnecting these sources of data, establishing the quality of the new dataset, maintaining the data as it evolves and making it available as a data source for further browsing, search or exploration.

![Figure 1: Sören Auer (2011) “The Semantic Data Web” [1].](image1)

The linked data lifecycle proposed by José M. Alvarez compresses this to four cyclical stages: Produce, Publish, Consume and Feedback/Update. A process of validation may be applied to any or each of these four stages.

![Figure 2: José M. Alvarez. (2012) “My Linked Data Lifecycle” [2](image2)]

Finally, Michael Hausenblas presents a linked data life cycle in which the four
central stages cover similar processes in which data is modelled, published, dis-
covered and then integrated with other sources of data.

![Linked Data Lifecycle Diagram]

Figure 3: Michael Hausenblas (2011) “Linked Data lifecycle” [3]

In the following sections we will focus on three main stages that can be found throughout the various formulations of the linked data lifecycle. These three stages can also be mapped to the linked data principles.

**Creating Linked Data:** This involves data extraction, the creation of HTTP URIs and vocabulary selection. This relates to linked data principles 1 and 2 as it involves finding or creating HTTP URI names for things.

**Interlinking Linked Data:** This involves finding and expressing associations across datasets. This relates to linked data principle 3 as it involves providing links to other things.

**Publishing Linked Data:** This involves creating metadata about a dataset and making the dataset available for use. It therefore relates to linked data principle 3 in that it ensures useful information is returned about the dataset.

### 3.3 Creation of Linked Data

New data may be initially authored in linked data format. However, often data of interest may already be stored in some alternative format. The most common formats are:

- Spreadsheets or tabular data
- Databases
- Text

Several tools have been developed to support data extraction from these sources. We will look at a number of these in the final sections of this chapter. OpenRefine is a tool for translating spreadsheet or tabular data into linked data. R2RML [4] is a W3C recommendation for specifying mappings between relational databases and linked data. A number of tools exist to support extraction of data from free text including GATE, Zemanta and DBpedia Spotlight.

The aim of all these tools is to allow disparate, possibly messy data sources to be viewed in terms of the RDF data model. As described in chapter 1, RDF represents knowledge in the form of subject-predicate-object triples. The subject and object are the nodes of the triple. Nodes represent the concepts or entities within the data. A node is labelled with a URI, blank node or a literal. Relations between the concepts or entities are modelled as arcs, which correspond to predicates within the data model. Predicates are expressed by a URI.
As described above, Linked Data principle number 1 states that URIs should be used as names for things. URIs can be used to name both nodes and arcs within RDF triples. More specific guidance can be found on how to design effective URIs. These are referred to as Cool URIs. The guidelines for the design of Cool URIs [5] can be summarised as follows:

- **Simplicity**: short, mnemonic URIs
- **Stability**: maintain the URIs as long as possible
- **Manageability**: issue the URIs in a way that you can manage

Vocabularies model concepts and relationships between them in a knowledge domain. These can be used to classify the instance-level nodes expressed in the RDF triples. You should avoid defining your own vocabulary from scratch unless absolutely necessary. Try to make use of (and therefore link to) well-known vocabularies that are already available. A large number of vocabularies are available as Linked Open Data. Many can be discovered through the Linked Open Vocabularies (LOV) dataset [6]. Using LOV you can free text search for vocabularies (for example vocabularies related to the term “music”) and filter the results according a number of facets such as the associated domain (e.g. science, media) of the vocabulary.

LOV also allows a number of vocabularies to be visualized in terms of how interconnected they are with other vocabularies. For example, we can see that the Music Ontology (referred to as “mo” in the figure) references 25 other vocabularies, the principal ones including Friend Of A Friend (FOAF) [7] and DCMI Metadata Terms (dcterms) [8]. FOAF is useful in this context at it provides a way of describing the people who feature as music artists. For example, FOAF can be used to represent their name, gender and contact details. The namespace dcterms can be used to describe core features of the products of music artists (i.e. albums and tracks) such as their title and description.

The Music Ontology is itself referenced by seven other vocabularies in the LOV dataset. These visualizations help in gauging how widely used a vocabulary is and how it aligns with other available vocabularies.

Other lists of well-known vocabularies are available. The W3C SWEO Linking Open Data community project maintains a list of the most commonly used vocabularies [9]. The Library Linked Data Incubator Group maintains a list of vocabularies related to the linked open library data [10].
3.4 Interlinking Linked Data

Besides making use of existing vocabularies, the author or maintainer of a dataset should investigate how entities in the dataset can be linked out to entities in other datasets. This follows Linked Data principle number 4 by linking to other URIs so that the user can discover more things. RDF links between entities in different datasets can be specified on two levels: the instance level and the schema level. On the instance level links can be made between individual entities (e.g. people, places, objects) using the properties rdfs:seeAlso and owl:sameAs. The property owl:sameAs is used to express that two URI references actually refer to the same thing. The property rdfs:seeAlso indicates that more relevant information can be found by following the link.

In MusicBrainz the property owl:sameAs is used to connect resources referring to the same music artist. The property rdfs:seeAlso is used to connect albums produced by artists. The reason rdfs:seeAlso is used instead of owl:sameAs is that a URI may refer to a particular release of an album (such as the US rather than UK release). It would be incorrect to express that an express an owl:sameAs relationship between them as they may differ in terms of release date and geographical market. They may also differ in other ways such as their track listings and album covers. An approach to modelling different album releases and their characteristics will be described in section 3.7.2.

On the schema level, which contains the vocabulary used to classify the instance-level items, a number of relationships can be expressed using RDFS, OWL and the SKOS Mapping vocabulary. The RDFS properties rdfs:subPropertyOf and rdfs:subClassOf can be used to declare relationships between two properties or two classes from different vocabularies.

As described in chapter 2, OWL also provides predicates for stating that two classes, or two properties, have the same meaning, as follows:

```
mo:MusicArtist owl:equivalentClass ex:musician .
foaf:made owl:equivalentProperty ex:creatorOf .
```

The first of these triples means that all instances of one of these classes are also instances of the other. The second of these triples means that if two resources are connected by one of the properties then they are also connected by the other property. SKOS mapping properties can also be used to express alignment between concepts from different vocabularies. These will be discussed later in the
The process of detecting links between datasets is known as link discovery. Datasets are heterogeneous in terms of their vocabularies, format and data representation. This makes the process of link discovery far more complex. Determining whether two entities from different datasets refer to the same thing is an example of what is know as the entity resolution problem. Two types of ambiguity can make the process more challenging. Name ambiguities can result from typos, or the use of different languages or homonyms to describe a thing. Structural ambiguities result from entities having possibly inconsistent relationships to other entities in their respective datasets. These are resolved using ontology and schema matching techniques.

Mappings between datasets (either on the instance or schema level) can be discovered and expressed both manually and automatically. The manual comparison of pairs of entities from different datasets is impractical for larger datasets. SILK is a tool that can be used to discover and express relationships between datasets. This is will be described in section 3.7.4.

SKOS (Simple Knowledge Organisation System) is a data model for expressing and linking Knowledge Organisation Systems such as thesauri, taxonomies, and classification schemes [12]. SKOS is expressed as RDF triples. Here will we consider three SKOS mapping properties in particular:

- skos:closeMatch expresses that two concepts are sufficiently similar that they could possibly be used interchangeably.
- skos:exactMatch expresses that two concepts can be used interchangeably. This property is transitive.
- skos:relatedMatch expresses that there is an associative mapping between the two concepts.

We will now give some examples that make use of the following prefixes.

@prefix mo: <http://purl.org/ontology/mo/>
@prefix dbpedia-ont: <http://dbpedia.org/ontology/>
@prefix schema: <http://schema.org/>

The example below expresses that the concept MusicArtist from the Music Ontology is an exact match for MusicalArtist from the DBpedia ontology.

mo:MusicArtist skos:exactMatch dbpedia-ont:MusicalArtist.

Here are some other examples:


The following triples express that the concept SignalGroup (meaning the group of musical or audio signals captured in a recording session) from the Music Ontology is a close match for both the MusicAlbum concept from schema.org and the Album concept from DBpedia.

mo:SignalGroup skos:closeMatch schema:MusicAlbum.
Certain integrity conditions apply in order to ensure a consistent mapping between two vocabularies. Two concepts cannot be both a related and exact match. The properties skos:closeMatch, skos:relatedMatch and skos:exactMatch are all symmetric. Only skos:exactMatch is transitive.

Figure 6: Partial Mapping Relation diagram with integrity conditions.

3.5 Publishing Linked Data

Once the RDF dataset has been created and interlinked, the publishing process involves the following tasks:

1. Metadata creation for describing the dataset
2. Making the dataset accessible
3. Exposing the dataset in linked data repositories
4. Validating the dataset

These will be described in the following four subsections.

3.6.1 Providing metadata about the dataset

A published RDF dataset should have metadata about itself that can be processed by search engines. This metadata allows for:

- Efficient and effective search of datasets.
- Selection of appropriate datasets (for consumption or interlinking).
- Acquiring general statistics about the dataset such as its size.
The frequently used vocabulary for describing RDF datasets is VoID (Vocabulary of Interlinked Datasets) [13]. An RDF dataset is expressed as being of the type void:Dataset.

The VoID schema covers four types of metadata:

1. General metadata
2. Structural metadata
3. Descriptions of linksets
4. Access metadata

**General metadata**

General metadata is intended to help users identify appropriate datasets. This contains general information such as the title, description and publication date. It also identifies contributors, creators and authors of the dataset. The VoID schema makes use of both Dublin Core and FOAF predicates. A list of general VoID properties is shown in Figure 7.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcterm:title</td>
<td>literal</td>
<td>Name of the dataset</td>
</tr>
<tr>
<td>dcterm:description</td>
<td>literal</td>
<td>Description of the dataset</td>
</tr>
<tr>
<td>dcterm:source</td>
<td>RDF resource</td>
<td>Responsible for creating the dataset</td>
</tr>
<tr>
<td>dcterm:creator</td>
<td>RDF resource</td>
<td>Responsible for creating the dataset and for making changes to the dataset</td>
</tr>
<tr>
<td>dcterm:availability</td>
<td>void:availability</td>
<td>Date of creation of the dataset</td>
</tr>
<tr>
<td>dcterm:status</td>
<td>void:status</td>
<td>Date of last update of the dataset</td>
</tr>
<tr>
<td>dcterm:usage</td>
<td>void:usage</td>
<td>Date of last update of the dataset and of the dataset</td>
</tr>
<tr>
<td>dcterm:publisher</td>
<td>RDF resource</td>
<td>Entity responsible for making the data set available</td>
</tr>
<tr>
<td>dcterm:contributor</td>
<td>RDF resource</td>
<td>Entity responsible for making contributions to the dataset</td>
</tr>
</tbody>
</table>

**Figure 7: VoID General metadata.**

The VoID general metadata also describes licencing terms of the dataset using the dcterm:licence property (see [14] for a discussion of licensing issues). The topics and domains of the data are expressed using the dcterm:subject property. The property void:feature can be used to express technical features of the dataset such as its serialisation formats (e.g. RDF/XML, Turtle).

**Structural metadata**

This provides high-level information about the internal structure of the dataset. This metadata is useful when exploring or querying the dataset and includes information about resources, vocabularies used in the dataset, statistics and examples of resources in the dataset.

In the example below a URI (which happens to represent The Beatles) is identified as being an example resource in the MusicBrainz dataset.

```plaintext
:MusicBrainz a void:Dataset;
                           void:exampleResource
                           <http://musicbrainz.org/artist/b10bb5e-cf9e-42e0-be17-c2c3e1d2600d> .
```
It is also possible to specify the string that prefixes all new entity URIs created in the dataset. Below, all new entities in the MusicBrainz dataset are specified as beginning with the string http://musicbrainz.org/.

:MusicBrainz a void:Dataset;
void:uriSpace "http://musicbrainz.org/".

The property void:vocabulary identifies the most relevant vocabularies used in the dataset. It is not intended to be an exhaustive list. The example below states that the Music Ontology is a vocabulary used by the MusicBrainz dataset. This property can only be used for entire vocabularies. It cannot be used to state that a subset of the vocabulary occurs in the dataset.

:MusicBrainz a void:Dataset;

A further set of properties are used to express statistics about the dataset such as the number of classes, properties and triples. These statistics can also be expressed for any subset of the dataset.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void:triples</td>
<td>Number</td>
<td>Total number of triples contained in the data set.</td>
</tr>
<tr>
<td>void:entities</td>
<td>Number</td>
<td>Total number of entities that are described in the data set.</td>
</tr>
<tr>
<td>void:classes</td>
<td>Number</td>
<td>An entity must have a URI, and match the void:uriRegExPattern</td>
</tr>
<tr>
<td>void:properties</td>
<td>Number</td>
<td>Total number of distinct properties in the data set.</td>
</tr>
<tr>
<td>void:distinctSubjects</td>
<td>Number</td>
<td>Total number of distinct subjects in the data set.</td>
</tr>
<tr>
<td>void:distinctObjects</td>
<td>Number</td>
<td>Total number of distinct objects in the data set.</td>
</tr>
<tr>
<td>void:documents</td>
<td>Number</td>
<td>Total number of documents, in case that the data set is published as a set of individual documents.</td>
</tr>
</tbody>
</table>

Figure 8: VoID statistics about a dataset.

The void:subset property is defines parts of a dataset. The example below states that MusicBrainzArtists is a subset of the MusicBrainz dataset.

MusicBrainz a void:Dataset;

void:subset :. MusicBrainzArtists

The properties void:classPartition and void:propertyPartition are subproperties of void:subset. A subset that is the void:classPartition of another dataset contains only triples that describe entities that are individuals of this class. A subset that is the void:propertyPartition of another dataset contains only triples using that property as the predicate. A class partition has exactly one void:class property. Similarly, a property partition has exactly one void:property property. The example below asserts that there is a class partition of MusicBrainz containing triples describing individuals of mo:Release. It also asserts that there is a property partition that contains triples using mo:member as the predicate.

:MusicBrainz a void:Dataset;
void:classPartition [ void:class mo:Release .] ;

Descriptions of linksets
A **linkset** is a set of RDF triples in which the subject and object are described in different datasets. A linkset is therefore a collection of links between two datasets. The RDF links in a linkset often use the owl:sameAs predicate to link the two datasets. In the example below, LS1 is declared as a subset of the DS1 dataset. LS1 is a linkset using the owl:sameAs predicate. The linkset declares sameAs relations to entities in another dataset (DS2).

```
@PREFIX void: <http://rdfs.org/ns/void#>
@PREFIX owl: <http://www.w3.org/2002/07/owl#>

:DS1 a void:Dataset .
:DS2 a void:Dataset .
:LS1 a void:Linkset;
  void:linkPredicate
  owl:sameAs;
  void:target :DS1, :DS2 .
```

**Figure 9:** A collection of links between two datasets. Based on [15].

In the MusicBrainz example below a class partition named MBArtists is defined. This is a linkset that has skos:exactMatch links between MusicBrainz and DBpedia.

```
@PREFIX void: <http://rdfs.org/ns/void#>
@PREFIX skos: <http://www.w3.org/2002/07/owl#>

:MusicBrainz a void:Dataset .
:DBpedia a void:Dataset .
:MusicBrainz void:classPartition :MBArtists .
:MBArtists void:class mo:MusicArtist .
:MBArtists a void:Linkset;
  void:linkPredicate
  skos:exactMatch;
  void:target :MusicBrainz, :DBpedia .
```

**Access metadata**

The VoID schema can also be used to describe methods for accessing the dataset, for example the location of a URI where entities in the dataset can be inspected, a SPARQL endpoint or file containing the data. The predicate void:rootResource can be used to express the top terms in a hierarchically structured dataset.

<table>
<thead>
<tr>
<th>Method</th>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI look up endpoint</td>
<td>void:urlLookupEndpoint</td>
<td>Specifies the URI of a service for accessing the data set (different from the SPARQL protocol)</td>
</tr>
<tr>
<td>Root resource</td>
<td>void:rootResource</td>
<td>URI of the top concepts (entry for data sets structured in trees)</td>
</tr>
<tr>
<td>SPARQL endpoint</td>
<td>void:spatialEndpoint</td>
<td>Provides access to the data set via the SPARQL protocol*</td>
</tr>
<tr>
<td>RDF data dumps</td>
<td>void:database</td>
<td>Specifies the location of the dump file. If the data set is split into multiple files, then several values of this property are provided.</td>
</tr>
</tbody>
</table>

**Figure 10:** Methods for accessing metadata.
3.6.2 Providing access to the dataset

A dataset can be accessed via four different mechanisms:

- Dereferencing HTTP URIs
- RDFa
- SPARQL endpoint
- RDF dump

These will be described below.

As we saw earlier, the first two linked data principles state that URIs should be used as names for things and that HTTP URIs should be used so that users can look up those names. *Dereferencing is the process of looking up the definition of a HTTP URI.*

The URI http://dbpedia.org/resource/The_Beatles is used to name The Beatles. It is not possible to send The Beatles over HTTP. However if you access this URI you will be forwarded to a document at some other location that can provide you with information about The Beatles. The HTTP conversation goes as follows:

1. You request data from a URI used to name a thing such as The Beatles (e.g. http://dbpedia.org/resource/The_Beatles). You may request data in a particular format such as HTML or RDF/XML.
2. The server responds with a 303 status (meaning redirect) and another location from which the data in the preferred format can be accessed.
3. You request data from the location to which you were redirected.
4. The server responds with a 200 status (meaning your request has been successful) and a document in the preferred format.

If you request data about The Beatles in HTTP format you will be redirected to a web page (http://dbpedia.org/page/The_Beatles). If you request the data in RDF/XML format then you will be redirected to an alternative document (http://dbpedia.org/data/The_Beatles.rdf). If you are providing rather than requesting the data then you need to decide which RDF triples should be returned from your dataset in response to dereferencing a HTTP URI about an entity (such as The Beatles) which cannot itself be returned. Guidance on what to return can be found in [16]. This can be summarised as follows:

- **Immediate description:** All of the triples in the dataset in which the originally requested URI was the subject.
- **Backlinks:** All triples in which the URI is the object. This allows browsers or crawlers to traverse the data in two directions.
- **Related descriptions:** Triples not directly linked to the resource but likely to be of interest. For example, information about the author could be sent with information about a book as this is likely to be of interest.
• **Metadata:** Information about the data, along the lines described in 3.4.1 such as the author of the data and licensing information.

• **Syntax:** There are a number of ways of serializing RDF triples. The data source may be able to provide RDF in more than one format, for example as Turtle as well as RDF/XML.

**RDFa** stands for “RDF in attributes”. RDFa is an extension to HTML5 for embedding RDF within HTML documents. The advantage of RDFa is that a single document can be used for both human and machine consumption of the data. A human accessing the data via a web browser need not be aware that an alternative RDF representation is embedded within the page. RDFa can be thought of a bridge between the Web of Data and the Web of (human readable) Documents.

Figure 11 lists the main attributes of RDFa. The about attribute specifies the subject that the metadata is about. The typeof attribute specifies the rdf:type of the subject. The property attribute specifies the type of relationship between the subject and another resource. The vocab and prefix attributes specify the default vocabulary and prefix mappings.

<table>
<thead>
<tr>
<th>Attribute role</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>prefix</td>
<td>List of prefix-name IRI pairs</td>
</tr>
<tr>
<td></td>
<td>vocab</td>
<td>IRI that specifies the vocabulary where the concept is defined</td>
</tr>
<tr>
<td>Subject</td>
<td>about</td>
<td>Specifies the subject of the relationship</td>
</tr>
<tr>
<td>Predicate</td>
<td>property</td>
<td>Express the relationship between the subject and the value</td>
</tr>
<tr>
<td></td>
<td>rel</td>
<td>Defines a relation between the subject and a URI</td>
</tr>
<tr>
<td></td>
<td>rev</td>
<td>Express reverse relationships between two resources</td>
</tr>
<tr>
<td>Resource</td>
<td>href</td>
<td>Specifies an object URI for the rel and rev attributes</td>
</tr>
<tr>
<td></td>
<td>resource</td>
<td>Same as href [used when href is not present]</td>
</tr>
<tr>
<td></td>
<td>src</td>
<td>Specifies the subject of a relationship</td>
</tr>
<tr>
<td>Literal</td>
<td>datatype</td>
<td>Express the datatype of the object of the property attribute</td>
</tr>
<tr>
<td></td>
<td>content</td>
<td>Supply machine-readable content for a literal</td>
</tr>
<tr>
<td></td>
<td>xml:lang, lang</td>
<td>Specifies the language of the literal</td>
</tr>
<tr>
<td>Macro</td>
<td>typeof</td>
<td>Indicate the RDF type(s) to associate with a subject</td>
</tr>
<tr>
<td></td>
<td>inlist</td>
<td>An object is added to the list of a predicate</td>
</tr>
</tbody>
</table>

**Figure 11: RDFa attributes.**

Below we can see a portion of HTML+RDFa contained in a HTML <div> element. The subject of this fragment of RDFa is specified using the about attribute. Here the subject is the MusicBrainz URI for The Beatles. In the line below we see the typeof property which is used to specify the rdf:type of the subject. The type of The Beatles is specified as the MusicGroup concept from the Music Ontology.

```xml
<div class="artistheader"
  about="http://musicbrainz.org/artist/b10bhbcd-cf9e-42e0-be17-2c3e1d2600d#_
  typeof="http://purl.org/ontology/mo/MusicGroup">
  ...
</div>
```

Below we can be see this RDF triple extracted from the HTML+RDFa.
Figure 12 shows an example of a page in HTML+RDFa format. This is the MusicBrainz page about The Beatles (http://musicbrainz.org/artist/b10bbbf-cf9e-42e0-be17-e2c3e1d2600d). As mentioned earlier, the human reader need not be aware of the RDF embedded within the page.

An RDFa distiller and parser can be used to extract the RDF representation. In Figure 3.13 the URL for the MusicBrainz page about The Beatles has been entered into the form (http://www.w3.org/2007/08/pyRdfa).

Figure 13: Extracting RDF from a MusicBrainz page in HTML+RDFa format.

Figure 14 shows a fragment of the RDF contained in the page, represented in the N-Triples format.

Figure 14: Extracted RDF in N-Triples format.

An RDF Dump is a file that contains the whole or some subset of an RDF dataset.
A dataset may be split over several data dumps. An RDF dump may use one of a number of formats. RDF/XML encodes RDF in XML syntax. N-Triples is a subset of the Turtle format in which the RDF is represented as a list of dot-separated triples. The format N-Quads is an extension of N-Triples in which a fourth element specifies the context or named graph of each triple. A site that maintains a list of available RDF data dumps can be found in [17].

SPARQL is a language that can be used to query an RDF dataset. A SPARQL endpoint is a service that processes SPARQL queries and return results. SPARQL queries can be used to retrieve particular subsets of the dataset. See chapter 2 for more information on SPARQL. Lists of publicly available SPARQL endpoints are maintained at [18] and [19].

3.6.3 Exposing the dataset in linked data repositories

Data catalogs, markets or repositories are platforms that provide access to a wide range of contributed datasets. They assist data consumers in finding and accessing new datasets. Catalogs generally offer relevant metadata about the dataset. The open source platform CKAN can be used for managing and providing access to a large number of datasets. CKAN would be recommended for a large institution that wanted to manage access to a number of datasets. The Data Hub is a public linked data catalog to which datasets can be contributed. CKAN and Data Hub will be described in more detail in section 3.7.

3.6.4 Validating the dataset

There are three different ways in which an RDF dataset can be validated. The first set (labelled accessibility in the Figure 15) checks that URIs are dereferenced correctly, using the HTTP client-server dialogue as described in section 3.5.2. A second set (labelled parsing and syntax) is used to validate the syntax of the RDF that is returned. Separate services are available for validating RDF/XML and RDFa markup. Finally, RDF:Alerts is a general purpose validation service for checking syntax and other problems such as undefined properties and classes and data type errors.

Figure 15: Ways of validating an RDF dataset.
3.6 Providing Linked Data: Checklist

Creating Linked Data

- Are all the relevant entities/concepts were effectively extracted from the raw data?
- Are all of the URI you have created dereferenceable?
- Are you reusing terms from widely accepted vocabularies and only invented terms when one did not already exist?

Interlinking Linked Data

- Is the dataset linked to other RDF datasets?
- Are the created vocabulary terms linked to other vocabularies?

Publishing Linked Data

- Do you provide dataset metadata?
- Do you provide information about licensing?
- Do you provide additional access methods such as data dump or SPARQL endpoint?
- Is the dataset registered in LD catalogs, using CKAN within an organisation or publicly via the Data Hub?

3.7 Linked Data Catalogs and Tools for Providing Linked Data

3.8.1 Linked data catalogs

As we saw in the previous section, data catalogs are platforms that provide access to a wide range of datasets from different domains. Below we describe CKAN that can be used to build data catalogs and The Data Hub, which is a public catalog of datasets.

CKAN [20] is an open source platform for developing a catalog for a number of datasets. CKAN may be used by an organisation for internally managing their datasets. These datasets need not be publically available as part of the Linking Open Data cloud. CKAN features a number of tools for data publishers to support:

- Data harvesting
- Creation of metadata
- Access mechanisms to the dataset
• Updating the dataset
• Monitoring the access to the dataset

Figure 16. CKAN.

CKAN has a schema for describing contributed datasets. This is similar to the VoID schema described in section 3.5.1.

Figure 17: Overview of the CKAN portal (from [20]).

The Data Hub [21] is a community-run data catalog that contains more than 5,000 datasets. Data Hub is implemented using the CKAN platform and can be used to find public datasets. In The Data Hub, datasets can be organised into groups each having their own user permissions. Groups may be topic based (e.g. archaeological datasets) or datasets in a particular language or originating from a certain country.
Figure 18: The Data Hub.

The group “Linking Open Data Cloud” catalogs datasets that are available on the Web as Linked Data.

Figure 19: The group Linking Open Data Cloud on The Data Hub.

Every bubble in the Linking Open Data cloud (shown in Figure 3.20) is registered with the Data Hub. For a dataset to be included in this cloud it must satisfy the following criteria:

- The dataset must follow the Linked Data principles (see section 3.3)
- The dataset must contain at least 1,000 RDF triples
- The dataset must contain at least 50 RDF links to a dataset that is already in the diagram
- Access to the dataset must be provided
Once these criteria are met, the data publisher must add the dataset to the Data Hub catalog, and contact the administrators of the Linking Open Data Cloud group.

Figure 20: Linking Open Data Cloud.

3.8.2 Linked data and commercial search engines

Search engines collect information about web resources in order to enrich how search results are displayed. Snippets are the few lines of text that appear underneath a search result link to give the user a better sense of what can be found on that page and how it relates to the query. Rich snippets provide more detailed information, by understanding something about the content of the page featured in the search results. For example, a rich snippet of a restaurant might show a customer review or menu information. A rich snippet for a music album might provide a track listing with a link to each individual track.

Figure 21: Example of a Rich Snippet.

Rich snippets are created from the structured data detected in a web page. The structured data found in a web page may be represented using RDFa as the mark up format and schema.org as the vocabulary. schema.org is a collaboration between Google, Microsoft and Yahoo! to develop a markup schema that can be used by search engines to provide richer results.

schema.org provides a collection of schemas for describing different types of resource such as:

- Creative works: Book, movie, music recording, ...
- Embedded non-text objects
- Event
- Health and medical types
- Organization
- Place, local business, restaurant
- Product, offer, aggregate offer
- Review, aggregate rating

Data represented using schema.org is recognized by a number of search engines such Bing, Google, Yahoo! and Yandex. schema.org also offers an extension mechanism that a publisher can use to add more specialised concepts to the vocabularies. The aim of schema.org is not to provide a top-level ontology, rather it puts in place core schemas appropriate for many common situations and that can also be extended to describe things in more detail.

**Google knowledge graph** uses structured data from Freebase to enrich search results. For example a search for the Beatles could include data about the band and its membership. In the snapshot of Figure 3.22 this can be seen in what is called a **disambiguation pane** to the right of the search results. This additional information could help the user to disambiguate between alternative meanings of their search terms. Google knowledge graph can also allow users to access directly related web pages that would otherwise be one or more navigation steps away from their search results. For example, the search for a Beatles album could provide links giving direct access to tracks contained on the album.

![Figure 22. Google search for The Beatles showing a disambiguation pane.](image)

Bing is now providing similar functionality to Google Knowledge Graph but built on the Trinity graph engine. A Bing search for “leopard” would produce structured data and disambiguation as shown in Figure 23.
Chapter 3. Providing Linked Data

Figure 23: Bing search for Leopard showing disambiguation.

The above examples use data graphs that connect entities to enrich search results. The Open Graph Protocol, originally developed by Facebook, can be used to define a social graph between people and between people and objects. The Open Graph Protocol can be used to express friend relationships between people and also relationships between people and things that they like: music they listen to, books they have read, films they have watched. These links between an object and person are expressed by clicking Facebook “like” buttons that can be added by publishers to websites outside the Facebooks domain. RDFa embedded in the page provides a formal description of the “liked” item. The Open Graph Protocol supports the description of several domains including music, video, articles, books, websites and user profiles.

Figure 24: Example Open Graph relationships (from [22]).

The Open Graph Protocol can be used to express different types of actions for different types of content. For example, a user can express that they want to watch, have watched or give a rating for a movie of TV programme. For a game, a user may record an achievement or high score.
This social graph of people and objects can then be used in *Facebook Graph Search*. This can be used to search not only for objects, but for objects liked by friends that have other properties such as living in a particular location.

![Facebook Graph Search](image)

**Figure 25: Facebook Graph Search.**

### 3.7.1 3.9 Tools for Providing Linked Data

In this section we will look at some of the tools that can assist with the creation and interlinking of linked data, introduced in sections 3.3 and 3.4.

- Extracting data from spreadsheets: OpenRefine.
- Extracting data from RDBMS: R2RML.
- Extracting data from text: OpenCalais, DBpedia Spotlight, Zemanta, GATE.
- Interlinking datasets: Silk.

#### 3.9.1 Extracting data from Spreadsheets: OpenRefine

First, we will look at how RDF data can be created from tabular data such as that found in spreadsheets. This relates to the part of the architecture shown in Figure 3.26. Tabular data can be represented in a number of common formats. CSV (Comma Separated Values) and TSV (Tab Separated Values) are two common plain text formats for representing tables. Tables can also be represented in HTML and in spreadsheet applications. Tabular data can also be represented in JSON (Javascript Object Notation), originally developed for use with the Javascript language and now a common data interchange format on the Web.

The transformation of tabular data to an RDF dataset will involve mapping items mentioned in tables to existing vocabularies, interlinking to entities for other datasets and to some extent data cleansing where alternative names or mistyped names for items mentioned in the tables need to be handled.
Figure 26: Integrating chart data.

Below we can see an example of data represented in CSV format. This shows sales data for a number of music artists. Each row is divided into two cells by the comma. So, for example, the first row tells us that The Beatles have sold 50 million records. Elvis Presley is not far behind with 203.3 million records.

The Beatles, 250 million
Elvis Presley, 203.3 million
Michael Jackson, 157.4 million
Madonna, 160.1 million
Led Zeppelin, 135.5 million
Queen, 90.5 million

Below we can see the first row of sales data represented in JSON format. Here the rank order of the artists based on sales is made explicit as an additional column in the data. The cells in the data also have labels, reflecting the column labels you might find in a HTML table, and which are also implicit in the CSV format.

```json
{
  "artist": {
    "class": "artist",
    "name": "The Beatles"
  },
  "rank": 1,
  "value": 250 million
},
...
Finally, in Figure 27 we see sales data represented as a HTML table. Here we have explicit column labels and also additional information related to the active period and first release of each artist.

![Figure 27: List of best-selling music artists (from [23]).](image)

We will now look at how OpenRefine [24] can be used to translate tabular formats to RDF. The OpenRefine tool was originally developed by the company Metaweb as a way of extracting data for Freebase, a vast collaborative knowledgebase of structured data. This tool later became GoogleRefine and was then renamed OpenRefine when released as an open source project. When using OpenRefine, the first step is to create a project and import tabular data in a format such as Microsoft Excel or CSV.

![Figure 28: Importing data to OpenRefine.](image)

Your browser does not support the video tag.

*Movie 5: Screencast of OpenRefine.*
As illustrated in Figure 3.29, OpenRefine assists us in transforming a tabular format such as CSV to RDF data. A number of processes are involved in this transformation. First, we can see that the serialisation has changed from comma-separated rows of data to RDF data in Turtle format. Second, the artists listed in the first column have been transformed from text strings to MusicBrainz URIs. Third, the sales figures have been transformed to a number with an integer data type. Finally, we have a relation (totalSales) linking the artist to the number. We will now look step-by-step at how this transformation is carried out.

Figure 29: Translating CSV data to RDF.

The first step involves defining the rows and columns of the dataset. This can involve deleting columns not required in the RDF data and splitting columns based on specified conditions. To help with this process OpenRefine provides a powerful expression language called OpenRefine Expression Language. This has a number of functions for dealing with different types of data such as Booleans, strings, and mathematical expressions. The OpenRefine Expression Language is also still often known by the acronym GREL. This dates back to when it was known as the Google Refine Expression Language.

In the example of Figure 3.30, we used GREL to split the word “million” from the number in Column 2 to create Columns 2.2 and 2.3. We then multiply Column 2.2 by the number 1,000,000 to create the Total Sales column.
We then need to map the artists listed in Column 1 to MusicBrainz URIs. For this we can use the RDF Refine plugin developed by DERI (www.deri.ie). The process of mapping between multiple representations of the same thing (in this case artists represented by a string and a MusicBrainz URI) is known as entity reconciliation. Entity reconciliation can be carried out against the SPARQL endpoint of an RDF dataset. Textual names for things (such as The Beatles) are matched against the text labels associated with the entities in the dataset. In the figure the artist names have been reconciled with the MusicBrainz URIs listed in a column headed musicbrainz-id.

---

**Figure 30: Data transformation in OpenRefine.**

**Figure 31: Entity reconciliation in OpenRefine.**
The data is then transformed into RDF triples. In the example of Figure 3.32, we have specified that a triple should be generated for each row in which the MusicBrainz URI is connected to the Total Sales data by the totalSales property. The RDF Preview tab shows what the first 10 rows of data will look like as RDF triples.

![Previewing RDF triples in OpenRefine.](image)

3.9.2 Extracting data from relational databases: R2RML

For data stored in multiple tables in a relational database we need a more expressive way of defining mappings to an RDF dataset. R2RML (Relational Database to RDF Mapping Language) [4] can be used to express mappings between a relational database and RDF that can then be handled by an R2RML engine. R2RML can be used to publish RDF from relational databases in two ways. First, the data could be transformed in batch as an RDF dump. This dump could then be loaded into a RDF triplestore. The endpoint of the triplestore could then be used to run SPARQL queries against the RDF dataset. Second, the R2RML engine could be used to translate SPARQL queries on-the-fly into SQL queries that can be run against the relational database.
In 2012, the W3C made two recommendations for mapping between relational databases and RDF [25]. The first recommendation defines a *direct mapping* between the database and RDF. This does not allow for vocabulary mapping or interlinking, just the publishing of database content in an RDF format without any additional transformation. The direct mapping recommendation is not relevant here as we also wish to map items in the database (for example music artists) to existing URIs even though those URIs are not included in the database itself.

The second recommendation is R2RML that provides a means of assigning entities in the database to classes and mapping those entities in subject-predicate-object triples. It also allows the construction of new URIs for entities and interlinking them with the rest of the RDF graph.

Figure 3.34 shows the core database tables and relationships in the *MusicBrainz Next Generation Schema (NGS)* that was released in 2011. In the diagram, the Primary Key (PK) of a table indicates that each entry in that column can be used to uniquely reference a row. A Foreign Key (FK) uniquely identifies a row in another table. MusicBrainz NGS provides a more complex way of modelling musical releases. For example, before NGS it was not possible to relate together multiple releases of the same album at different times and in different territories.

The NGS defines an Artist Credit that can be used to model variations in artist name. This can describe multiple names for an individual and different names for various groups of artists. For example, the song “Get Back” is credited to “The Beatles with Billy Preston” rather than “The Beatles”. This would be difficult to represent in MusicBrainz without NGS.

Another major change in MusicBrainz NGS is how musical releases are modelled. A *Release Group* is an abstract “album” entity. A Release is a particular product you can purchase. A Release has a release date in a particular county and on a particular label. Many releases can belong to the same Release Group. A Release
may have more than one Medium (such as MP3, CD or vinyl). On each Medium, the Release will have a tracklist comprising a number of tracks. Each track has a Recording that is a unique audio version of the track. This could be used, for example, to distinguish between the single and album versions of a track. Artist Credit can be assigned to each individual track as well as the Recording, Release and Release Group. Artist Credit can also be assigned to a Work, which represents the composed piece of music as distinct from its many recordings.

Figure 34: Relational Schema for the Music Database.

In Figure 3.35 we can see a few core classes in the Music Ontology to which we can map when generating RDF data from data represented in the MusicBrainz NGS Relational Database. The Music Ontology models a MusicArtist as composing a Composition, which is then produced as a MusicalWork. A Performance of a MusicalWork can be recorded as a Signal that can be Produced from Recordings of that Performance.

Figure 35: The Music Ontology (from [26]).

Mapping a database table to a class in an ontology is relatively straightforward. Here we will map the Artist table in MusicBrainz NGS to the MusicArtist class in the Music Ontology. Mappings in R2RML are mostly specified as instances of what are referred to TriplesMaps. The TriplesMap specified below has the
identifier lb:Artist. The logicalTable is the source data table from which the triples are derived. In this case the logicalTable is the table named “artist” in the relational database. As we shall see in the next example, the logicalTable can also be the result of a SQL query across a number of tables rather than a single table in the database.

The subjectMap defines the subject of the triple. The subject is constructed as a MusicBrainz URI with the entry from the gid column of the Artist table being inserted between http://musicbrainz.org/artist/ and the # symbol. The specified predicate is mo:musicbrainz_guid which links a MusicBrainz URI to its ID in the form of a string. The object of the triple is also the entry from the gid column but represented as a string.

lb:Artist a rr:TriplesMap;
rr:logicalTable [rr:tableName “artist”];
rr:subjectMap [rr:class mo:MusicArtist;
rr:template “http://musicbrainz.org/artist/\protect\char"007B\relaxgid\protect\char"007D\relax#_”];
rr:predicateObjectMap [rr:predicate mo:musicbrainz_guid;
rr:objectMap [rr:column “gid”]
rr:datatype xsd:string]] .

Database columns can also be mapped to properties. In the example below we supply a name property to each of the MusicBrainz URIs generated in the previous example. In this case the logicalTable used in the TriplesMap is an SQL query that returns a table of results. This query joins two tables to link the gid of an artist to the artist’s name. The subject of the triple is the same as the subject specified using the TriplesMap from the previous example. The predicate is foaf:name. The object is the name column from the logicalTable.

lb:artist_name a rr:TriplesMap;
rr:logicalTable [rr:sqlQuery
""""SELECT artist.gid, artist_name.name
FROM artist
INNER JOIN artist_name ON artist.name = artist_name.id""""];
rr:subjectMap lb:sm_artist;
rr:predicateObjectMap [rr:predicate foaf:name;
rr:objectMap [rr:column “name”]] .

MusicBrainz Next Generation Schema (NSG) also provides Advanced Relationships as a way of representing various inter-relationships between key MusicBrainz entities such as Artist, Release Group and Track. The table l_artist_artist is
used to specify relationships between artists. Each pairing of artists would be represented as a row in the `l_artist_artist` table. Each pairing of artists refers to a Link. One link would be member_of that would specify a relation between an artist and a band of which they were a member.

**Figure 36: NGS Advanced Relations.**

The R2RML triplesmap below shows how we would specify that an artist is a member of a band. Here the logicalTable is the result of a complex query that associates artists with a band. The Music Ontology member_of predicate is used to associate an artist with a MusicBrainz URI identifying the band.

```r2rml
lb:artist_member a rr:TriplesMap ;
rr:logicalTable [rr:sqlQuery
"""
SELECT a1.gid, a2.gid AS band
FROM artist a1
INNER JOIN l_artist_artist ON a1.id = l_artist_artist.entity0
INNER JOIN link ON l_artist_artist.link = link.id
INNER JOIN link_type ON link_type = link_type.id
INNER JOIN artist a2 on l_artist_artist.entity1 = a2.id
WHERE link_type.gid = '5be4c609-9afa-4ea0-910b-12ffb71e3821'
AND link.ended = FALSE"""] ;
rr:subjectMap lb:sm_artist ;
rr:predicateObjectMap
[rr:predicate m:member_of ;
rr:objectMap [rr:template "http://musicbrainz.org/artist/\protect\char'007b\relaxband\protect\char'007d/relax#"] ;
rr:termType rr:IRI]] .
```
3.9.3 Extracting data from text: DBpedia Spotlight, Zemanta and OBIE

The previous tools worked on data that was already in some tabular or relational structure. Work carried out by the tools largely involved transforming this existing structure to a triple structure as well as some mapping and interlinking. Text is more open and ambiguous and involves more than transformation. As we shall see later, text extraction is only correct to some level of precision and recall.

OpenCalais [27] can be used to automatically identify entities from text. OpenCalais uses natural language processing and machine learning techniques to identify entities, facts and events. OpenCalais is difficult to customise and has variable domain-specific coverage.

DBpedia Spotlight [28] can be used to identify named entities in text and associate these with DBpedia URIs. In the snapshot of Figure 3.38, recognised entities in the submitted text have been hyperlinked to their DBpedia URIs. DBpedia Spotlight is not easy to customise or extend and is currently only available in English.

Zemanta [29] is another general-purpose semantic annotation tool. Zemanta is used by bloggers and other content publishers to find links to relevant articles and media. Best results require bespoke customization.
Figure 39: Zemanta.

GATE (General Architecture for Text Engineering) [30] is an open-source framework for text engineering. GATE started in 1996 and has a large developer community. GATE can be more readily customized for text annotation in different domains and for different purposes. GATE is used worldwide to build bespoke solutions by organisations including the Press Association and National Archive. Information extraction is supported in many languages. GATE can also parse text as well as recognise entities and can therefore be used to identify entities depending on their function in the sentence. For example GATE could be used to extract an entity only when used as the noun phrase, rather than the verb phrase, of the sentence.

LODIE [31] is an application built with GATE using DBpedia. LODIE uses Wikipedia anchor texts, disambiguation pages and redirect pages to help find alternative versions of things.

Figure 40: LODIE.

Precision and recall measures can be used to compare text annotation tools. Precision and recall are both values from 0 to 1. Precision indicates what proportion of annotations are correct. Recall indicates what proportion of possible correct annotations in the text are identified by the tool.

Figure 3.41 shows precision and recall figures for these tools. Precision and recall figures are shown in pairs, precision on the left, recall on the right separated by a slash. Precision and recall are shown for types of entity (person, location and organisation) as well as in total. We see that DBpedia Spotlight has relatively
good precision but relatively low recall, identifying 39% of entities in the text. Zemanta has similar precision but higher recall. LODIE has the highest recall but lower precision.

Good results can also be achieved by combining the methods. Only annotating entities suggested by both Zemanta and LODIE (i.e. the intersection of Zemanta and LODIE) gives very high precision. The union of Zemanta and LODIE gives high recall but lower precision.

<table>
<thead>
<tr>
<th></th>
<th>PER</th>
<th>LOC</th>
<th>ORG</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB Spotlight</td>
<td>0.97 / 0.40</td>
<td>0.82 / 0.46</td>
<td>0.85 / 0.31</td>
<td>0.85 / 0.39</td>
</tr>
<tr>
<td>Zemanta</td>
<td>0.96 / 0.84</td>
<td>0.89 / 0.62</td>
<td>0.82 / 0.57</td>
<td>0.90 / 0.68</td>
</tr>
<tr>
<td>LODIE</td>
<td>0.81 / 0.92</td>
<td>0.73 / 0.76</td>
<td>0.55 / 0.59</td>
<td>0.71 / 0.74</td>
</tr>
<tr>
<td>Zemanta ∩ LODIE</td>
<td>1.00 / 0.74</td>
<td>0.95 / 0.45</td>
<td>0.97 / 0.42</td>
<td>0.97 / 0.54</td>
</tr>
<tr>
<td>Zemanta U LODIE</td>
<td>0.94 / 0.53</td>
<td>0.77 / 0.76</td>
<td>0.72 / 0.71</td>
<td>0.82 / 0.81</td>
</tr>
</tbody>
</table>

Figure 41: Comparison of DBpedia Spotlight, Zemanta and LODIE.

An alternative to the generic services provided by DBpedia Spotlight and Zemanta is to build a GATE processing pipeline specifically for your domain. For this we take an RDF dataset and use this to produce what is called a GATE Gazetteer, which is a list of entities in a domain and associated text labels used to refer to those entities. We can produce a gazetteer using the RDF data produced from the R2RML transformation of the MusicBrainz NGS Relational Database (see section 3.7.2). A SPARQL endpoint to this data can be used to populate a custom gazetteer for the music domain. For example, the query in Figure 3.43 returns solo artists and music groups and their foaf:name. It also returns albums (represented by the SignalGroup class in the music ontology) and their dc:title. This provides a vocabulary of artists and albums with associated labels that can be used in the gazetteer.
Figure 42: Producing a GATE Gazeteer.

A GATE pipeline can be run locally or uploaded to the GATE cloud. Once set up, text can be submitted and then annotated using the MusicBrainz data. The annotated text can then be output in a format such as RDFa.

Figure 43: GATE Cloud.

3.9.4 Interlinking datasets: SILK

As mentioned in section 3.4, manually interlinking large datasets is not feasible. SILK [32] is a tool that has been developed to support the interlinking of datasets. In our case we may wish to define links between artists mentioned in MusicBrainz and the same entities in DBpedia. This process fits specifically in the interlinking phase of the diagram of Figure 44.

Figure 44: Interlinking datasets with SILK.

SILK is an open source tool for discovering RDF links between data items within
different Linked Data sources. The Silk Link Specification Language (Silk-LSL) is used to define rules for linking entities from two different datasets. For example, a rule may express that if two entities belong to specified classes and have matching labels then they should be linked by a certain property. This property could be owl:sameAs or some other property such as skos:closeMatch (see section 3.4).

SILK can run in different variations. It can be run locally on a single machine, on a server or distributed across a cluster. The SILK workflow is shown in Figure 3.44. The first step is to select the two datasets. Generally, one dataset would be your own and another including some of the same entities but named with different URIs. In the second step, we specify the two datasets either by loading an RDF dump or pointing to a SPARQL endpoint. We also specify the types of entities to be linked. In the third step, we express the linkage rules in Silk-LSL. The discovered links can then be published as a linkset (see section 3.5.1) with your dataset.

![SILK workflow](image)

**Figure 45: SILK workflow based on [33]**.

The rules for comparing two entities can consider not only the two entities themselves but also additional data items found in the graph around each of those entities. For example, a rule may compare the rdfs:label of one entity with the foaf:name of another. The paths from the compared entities to these additional data items are specified as RDF paths. Different transformations can be also be performed on the compared data items. For example, if they are strings (such as labels or names) then they may be both transformed into lower case to prevent the use of lower and upper case leading to a mismatch. The linkage rules also define the comparators used to compute the similarity of the two data items. When comparing two strings, an exact match may be required. Alternatively, similarly may be computed as a Levenshtein edit distance. This indicates the number of single character changes that would need to be made in order to turn one string into the other. This provides a way of matching data items that may contain typos. Similarity metrics can be used for other data types such as dates. Finally, aggregations can be computed from data items associated with the entity. For example, two potentially matching albums could be compared in terms of the number of tracks that they contain. If the number of tracks is equal then this could be further evidence that these two entities refer to the same album.

The SILK Workbench is a web application built on top of SILK that can be used to create projects and manage the creation of links between two RDF datasets. The SILK Workbench has a graphical editor that can be used to create linkage rules. Support is also provided for the automatic learning of linkage rules. Figure 3.46 shows a snapshot of the SILK Workbench. A new project has been created
with the name “MyLinkedMusic”. Two datasets have been added, labelled as DBpedia and MyMusicBrainz. The sections below this are concerned with the specification of linkage rules and the location and format of the output.

Figure 46: Overview of a Silk Workbench project

Figure 47 shows how the graphical editor can be used to specify linkage rules. In this example the foaf:name of the MusicBrainz entity and the rdfs:label of the DBpedia entity are both transformed to lower case and then compared in terms of their Levenshtein edit distance.

Figure 47: Adding a linkage rule in the SILK workbench.

The linkage rules can then be used to generate a set of links as shown in Figure 48. Each of these links between a MusicBrainz and DBpedia identifier has a confidence score. The larger the Levenshtein edit distance between the foaf:name and rdfs:label, the lower the confidence. All of the examples listed have a confidence score of 100% indicating a zero edit distance between the two literals. Confidence can be accumulated from a number of sources. When comparing music groups, this could include other data such as their membership and formation date.
Figure 48: Generating links with the SILK workbench.
The SILK Workbench also provides an interface for examining automatically learned rules. These suggested rules can then be added to the set of linkage rules or rejected.

Figure 49: Rule learning with the SILK workbench.
Chapter 4

Interaction with Linked Data

4.1 Introduction

In the previous chapter we described how linked data could be made available, for example, via a data dump or SPARQL endpoint. The emphasis was on providing data in a form readable by machines such as RDF/XML or Turtle. The results of a SPARQL query could be provided in a format such as JSON that a software developer could then use to develop an application. In this chapter, rather than focusing on how data can be made available to applications in machine-readable form, we will look at how linked data can be presented for use by a human audience.

In this chapter we will once again use music as a motivating example but focus on modules that enable people can interact with and explore music-related data. We consider how RDF data could be visualised and also how statistical and machine learning techniques can be used to extract interesting patterns from data.

Applying visualization techniques to RDF data provides much more engaging ways of communicating data. As illustrated in figure 1, a user may initiate the process by entering a search query. The semantics of the resources returned from the search can then be used to construct a coherent presentation of the information. Text descriptions of RDF resources can be embedded with links to other resources and also other media such as images. Sets of resources, such as the single or album releases of a band, can be aggregated and presented in different forms such a map (showing for example where they were released) and bar chart (showing for example the number of individual releases of each work).
Visualizations have a number of advantages in how they communicate data. Visualizations make it possible to tell a story with the data, giving it some meaning and interpretation. Visualizations also make it easier for people to spot patterns in the data such as changes over time or between different locations. Visualizations can also reveal differences between datasets that may not be apparent from simple descriptive statistics such as the mean and variation of a set of values. An interesting example is Anscombe’s quartet of datasets (see figure 2). The four datasets have a number of statistical properties in common such as the mean and variance of the x and y variables and the correlation and regression values between x and y. However differences between the datasets are clearly apparent when visualized.

Figure 2: Anscombe’s quartet of datasets having similar statistical properties but appearing very different when plotted [2].

4.2 Linked Data Visualization

4.3 LD Visualization Techniques

Linked Data visualization techniques aim to provide graphical representations of some information of interest within a dataset. Visualizations need to be selected that match the type of data, for example whether it be numerical data or location information. Visualizations also need to be selected that match the task that the user is trying to perform, bringing to the foreground the types of data and patterns in the data that they wish to work with.

Figure 3 illustrates the way in which raw RDF data needs to be transformed to produce visualizations. First, the data of interest is extracted from the dataset. Performing a SPARQL query can do this. Second, the data needs to be transformed in order that it can be displayed with the intended visualization methods. A simple example would be to extract a numerical value from a string so that it could then be visualized on a bar chart. Third, the data needs to be mapped to the constructs of the visualization. For example, the numerical value could be
mapped to the y-axis of a bar chart. The view resulting from this process may not be a static image. It may provide ways in which the user can interact with the data, by zooming or clicking to trigger further visualizations.

Figure 3: Linked Data visualization process (partially based on [3])

An example of the Linked Data visualization process is shown in figure 4. A SPARQL query is used to extract the data of interest from the MusicBrainz dataset, in this case the number of Beatles releases per country. In the second step the string value representing the country is transformed into a country code that can be used on the visualization. In the third step, this data is passed to a heatmap visualization. The country code is used to identify an area on the map. The number of releases is mapped to the warmth of the colour of that region. The resulting heatmap is shown in figure 5.

Figure 4. Example of the Linked Data visualization process.
4.4 Challenges for Linked Data visualization

The domain of Linked Data delivers a number of challenges for Linked Data. First, there is the challenge of scalability. The data of interest, for example returned from a SPARQL query of a dataset could be vast. Visualization techniques need to be used that can scale to large amounts of data. The visualizations tools also need to be powerful and efficient enough to render the information with an acceptable timescale.

One useful way of addressing scale is to provide visualizations that enable user interaction. All data of potential interest then does not need to be provided within a single static view. The user has control over the visualization, allowing navigation through the data. User interaction functionality may also provide support for user editing of the data or annotation of the visualization itself. When visualizing the data, the user may spot errors or omissions that could be fixed interactively through the visualization. The user may also wish to highlight or make comments about some region of the data, essentially adding metadata to the dataset.

Linked data visualizations and the software mechanisms used to construct them should ideally be reusable. Developing tools to produce visualizations such as such as maps and timelines involves a lot of effort. It is therefore more efficient to produce generic tools that can be reused with many datasets. The emergence of standards for representing types of data (such as time and location information) facilitates the use of visualization tools. Ideally, the resulting visualizations should also be reusable and sharable using standard formats.

4.5 Challenges for Open Linked Data visualization

When we consider Linked Open Data rather than just Linked Data, further challenges need to be addressed. First, the data of interest may be partitioned across different repositories. Assembling the data of interest will therefore require access to multiple datasets. Second, the assembled heterogeneous data may model the concepts in different ways. Alternative formats may also be used for values.
For example in different repositories date information may be variously represented as DD-MM-YYYY, MM-DD-YYYY or just YYYY. Third, working with a dataset assembled from multiple repositories increases the likelihood of missing data. Visualizations will need to be able to handle the level of missing data and perhaps also indicate to the user data that cannot be represented in the selected visualization.

### 4.6 Classification of visualization techniques

Visualization techniques can be classified according to the type of analytical task that the user is attempting to perform on the data. Visualization techniques such as pie charts are appropriate for comparing the attributes or values of different variables within the dataset. If the user wants to analyse relationships and hierarchies then graphs and other related techniques can be used. The analysis of data in time or space can be supported with timelines and maps. A scatter plot can be used to analyse three-dimensional data. Higher dimensional data can be visualised using techniques such as radar charts. The following subsections will describe in more detail a range of example visualization techniques and how they can be used.

#### Figure 6: Visualization techniques appropriate for different data analysis tasks.

#### 4.6.1 Comparison of attributes/values

The most appropriate visualization for comparing attributes or values will depend on the nature of the data and the task. To compare absolute values (such as total number of sales) associated with a list of items (such as different albums) then a bar chart would be appropriate. If only relative rather than absolute values were of interest then a pie chart could be used. For bar charts and pie charts, the items associated with value (e.g. albums) do not necessarily have any predefined order or position in the chart. If the items do have a pre-define order (for example the release date of the albums) a line chart can be used to show the trend. Finally, frequency distributions for an ordered variable can be visualised using a histogram. This could be used for example to plot the frequency of tracks of varying lengths.
4.6.2 Analysis of relationships and hierarchies

Relationships between nodes can be visualised using a standard graph notation in which relationships are represented as lines. Graphs data can also be visualized using an arc diagram in which the nodes are organised linearly. Relationships between nodes are represented as half circles connecting the two nodes. When using an adjacency matrix, the nodes of the graph and placed on both the x-axis and y-axis. Relationships between the nodes are represented as entries in the grid.
Some visualizations are specifically designed for hierarchical graph data. The indented tree is a familiar formalism commonly used for visualising hierarchies and navigating file directories. The node-link tree is a tree visualization in which the root node is placed in the centre. This provides a visual cue as to the population level of different sections of the hierarchy.
Figure 9: Visualizing hierarchies using an indented tree (right) and node-link tree (right) [4].

A number of space filling visualization techniques have been designed specifically to give an indication of the population level of different parts of the hierarchy. Treemaps visualise nodes within a hierarchy as a set of rectangles. Containment can be used to represent hierarchical relationships between nodes. The size of the rectangle is generally used to represent the number of individuals of a node (i.e. class) within the dataset. Colour can be used to represent some feature of the nodes (i.e. classes) such as the discrete set of superclasses to which they belong.

The icicle visualization can be used to show a node hierarchy and gives a clear indication of depth at different parts of the hierarchy. The sunburst essentially folds the icicle visualization into a circle. Similarly, the rose diagram uses the size of sectors to indicate the population of parts of the hierarchy. Finally, a circle-packing visualization uses containment to represent the hierarchy and size.
of the circle to represent containment.

Figure 10: Space filling visualization of a hierarchy using treemaps (left) and icicles (right) [4].
4.6.3 Analysis of temporal or geographical events  
Timeline visualizations can be used in combination with both discrete data where, for example, individual events are marked as dots on the timeline. Timelines can also be used to represent continuous data. For example, changes in the frequency of different types of event over time could be represented using colour to indicate event type and thickness of the band to indicate the frequency of those event types at that
point in time.

Figure 12: Visualizing discrete [1] (left) and continuous [5] (right) data over time.

Data can also be associated with different types of map visualization. This could involve plotting coordinate points on the map. If the granularity of interest is areas (such as countries) rather than specific points in space, then a choropleth maps can be used. Colour can indicate some feature of the area such as the number of specific data points associated with it. The heat map of Beatles releases shown in figure 5 is an example of a choropleth map. If it is not necessary to indicate the borders between areas (such as country borders) then a Dorling cartogram can be used in which the centre of each circle falls within its associated area and both colour and size of the circle are used to visualize additional data.
4.6.4 Analysis of multidimensional data  Some visualizations can be used to represent data having 3 or more dimensions. Three-dimensional data can be represented using a scatter plot. As well as the x-axis and y-axis of the chart, the size of each dot placed on the scatter plot is used to represent a third dimension. Radar charts or parallel coordinates can be used to represent higher dimension data. In a radar (or star) chart, each multi-dimensional point is represented as a shape whose border connects each axis. The axes of a radar chart are represented as spokes of a wheel. In a parallel coordinates visualization, the axes correspond to vertical lines. A multi-dimensional point is show as a line connecting each axis.
Figure 14: Visualizing multidimensional data using a scatter plot (left) and radar or star chart (right) [4].
4.6.5 Other visualization techniques  

Text-based visualizations use word size to represent frequency. In a standard tag cloud, words or phrases indicate tags or annotations that have been associated with resources. The larger the word, the more commonly it has been used to tag the resources. A variation on the standard tag cloud is the phrase net visualization. This is often used to visualise a document or larger text corpus. Size reflects frequency of the word and also lines connect words that are in close proximity in the text.

Figure 15: Visualizing multidimensional data using parallel coordinates [4].
4.7 Applications of Linked Data visualization techniques

Visualizations can be applied to Linked Data in order to satisfy a number of aims. First, particularly given the potential scale of the Linked Data of interest, visualizations can be used to provide an overview of the data to guide further analysis. Visualization can be used to identify and analyse relevant resources, classes or properties in the dataset.

Visualizations can be used to reveal a great deal about the vocabulary or taxonomy used in the dataset to define resources. For example, the various methods of visualising a hierarchy, shown in section 4.6.2, would quickly reveal attributes related to the depth and breadth of the hierarchy and the frequency of use of different concepts.

Visualizations can also be used to reveal different types of desired and undesired patterns in the data. A graph visualization could reveal missing links between nodes or uncover new paths between resources. Visualizations may also be used to uncover hidden patterns, errors or outliers in the data (along the lines illustrated in figure 2).

4.8 Summary of Linked Data visualization tool requirements

As described in section 4.4 Linked Data visualizations need to offer data navigation and exploration capabilities. Particularly, given the scale of the data, it is unlikely that a static visualization will provide an adequate view on the data for all purposes. This will involve providing user interaction capabilities such as being able to query data of interest, filtering values and folding or expanding parts of the visualization. Visualizations should also exploit the data structures that are inherent in Linked Data such as ontology or taxonomy hierarchies. Ideally, it should be possible for the user to publish or share their visualizations using standard presentation formats for easy distribution. This ability to share should also apply to the extracted data, that is, the particular viewpoint on the data established in the selection and user manipulation of the visualization tool.

4.9 Linked Data visualization tool types

Linked Data visualization tools can be organised into four categories. These range from text-based Linked Data browsers to toolkits with extensive functionality for data transformation and graphical visualization.

1) Linked Data browsers with text-based representation

A text-based LD browser dereferences URIs to retrieve a resource description that is then presented to the user. Most LD browsers will present not only text descriptions but also other media such as images associated with the resource. Text based browsers will generally include hypertext links to connected resources. These may be part of the immediate description (i.e. triples that link from this to another resource) or backlinks (i.e. triples that link from other resources to
this resource). See section 3.5.2 of chapter 3 for guidance on presenting resource descriptions.

2) Linked Data and RDF browsers with visualization options

Some Linked Data and RDF browsers make more extensive use of media associated with the resource and organise this media into more coherent or engaging presentations of the data. These browsers also offer greater use interaction. As well as links to connected resources they provide ways of querying and filtering the data. These types of browser can therefore be used to analyse as well as traverse the data.

3) Visualization toolkits

Visualization toolkits bring together a range of visualization techniques. They generally incorporate methods for transforming the raw data in order that it can be rendered by the visualization. Some visualization toolkits are specifically designed to consume Linked Data.

4) SPARQL visualization

Finally, SPARQL visualizations are tools that dynamically transform the output of SPARQL queries to produce visualizations. These tools can be used to support analysis of the dataset that is accessed by the SPARQL endpoint.

As summarised in the figure below, a number of current tools fall into these four categories. In the next section we will introduce Sig.ma and Sindice as examples of Linked Data browsers and then the Information Workbench will be introduced as an example of a visualization toolkit that can also visualize the results of SPARQL queries.

Figure 17: Types of Linked Data visualization tool.

4.10 Linked Data visualization examples

4.10.1 Sig.ma  Sig.ma [9] is a text-based Linked Data browser. Interaction is usually initiated by a text search. Sig.ma returns all triples associated with the search terms. The returned triples are grouped according to predicate. Sig.ma lists all found values for each predicate. The sources of each triple are displayed in the right hand panel. Each source has a number that is used to reference each
triple in the main body of the page.

Figure 18: The Sig.ma Linked Data browser.
For a particular predicate, such as title or label, Sig.ma may display the same information in multiple languages. Property values that are URIs can be followed in order to view the RDF data of that connected resource.

Figure 19: The Sig.ma Linked Data browser showing multiple values for a predicate.

4.10.2 Sindice Interaction with Sindice [10] also begins with a keyword search. A set or results that match the query are displayed. These can be filtered by document type, for example filtering results to only RDF documents. For each document, the user can inspect the RDF triples that it contains. The user can either inspect a cached set of triples or retrieve triples live from the resource.
Chapter 4. Interaction with Linked Data

Figure 20: The Sindice Linked Data browser.
This list of cached or live triples is displayed as a subject/predicate/object table. Sindice also offers other viewing options such as a graph of the triples.

Figure 21: Using Sindice to view the live or cached triples contained in a document.

4.10.3 Information Workbench  The Information Workbench [11] is a platform targeted at the whole lifecycle of linked data application development including integrating, managing, analysing and exploring linked data. Here we focus on the visualization capabilities of the Information Workbench. Generally, visualizations are constructed using data returned from a SPARQL query. A number of different visualization techniques can be applied to the data including bar charts, pie charts, Google maps and timelines. The visualizations allow for user interaction including browsing and exploring the data. A demo system visualizing MusicBrianz using the Information Workbench is available at [12].

As with Sig.ma and Sindice, user interaction may begin via text search. In the figure below “The Beatles” has been used as a search term. The user may select full text search. In this case, different types of resources will appear in the results set. For example the resource representing the album “With The Beatles” will be returned as well as a resource representing the band itself. The search can also be limited to particular types of resources, such as music artists. More structured searches can be specified, for example, matching the query only to artists from a specified country. Structured search queries are re-represented as SPARQL queries issued against the dataset.
Figure 22: Keyword and structured search using the Information Workbench.

Having searched for the Beatles and selected the band from the results page, the user is directed to an information page about the Beatles. The top of the page is composed of mash-ups with web services. The panel to the top right show tweets mentioning the Beatles. Below this is a Google map showing the UK as their country of origin. Bottom right is a mash-up with last.fm and YouTube data.

Figure 23: Top of the Beatles page showing mash-ups with web services.

Further down the page we see actual visualizations of the data, each making use of particular SPARQL queries against the dataset. The visualizations show: a table of the track numbers and playing times for Beatles albums (top right), the locations of Beatles releases (top left), a timeline of Beatles release (bottom right) and the number of releases for different albums (bottom right).

Figure 24: Visualizations of Beatles data using a table, map, timeline and bar chart.

The visualizations provided by the Information Workbench also enable user interaction. To the left of the figure below we see a tag cloud of music artists. The size of the text represents number of releases. The text label of each artist is a hypertext link that directs the user to data associated with the resource representing that artist. From there, the user can continue to navigate across the dataset.
Figure 25: Linking from a tag cloud to an information page about this artist.

In the Information Workbench all visualizations are implemented as widgets. The region of the dataset of interest is specified using a SPARQL query. The SPARQL query below requests the top ten Beatles releases based on their duration. The easiest way to visualize a result set is displaying it in a table.

```
SELECT ?release
WHERE {
    <http://dbpedia.org/resource/The_Beatles>
    ?release rdf:type release .
    GROUP BY ?release
    ORDER BY DESC(?avg)
    LIMIT 10
}
```

Figure 26: Returning a result set for a SPARQL query.

For many visualizations some level of configuration may need to be specified as well as the SPARQL query. In the example below, we can see that the type of visualization is specified on the first line. This is followed by the SPARQL query. Finally, any configuration settings are provided. If the returned data is to be presented on a bar chart then the author of the visualization will need to specify in the configuration settings the variables returned from the SPARQL query that correspond to the x-axis and y-axis. In the example below, release labels are placed on the x-axis and number of releases is placed on the y-axis. Other features of the visualization may be configured such as the colour and height.
Figure 27: Using a widget to specify a bar chart.

Using the same SPARQL query but with different widgets and configuration setting it is possible to create other visualizations of the same data such as a line chart or pie chart.

![Figure 27: Using a widget to specify a bar chart.](image)

Figure 28: Line chart and pie chart visualizations of the same SPARQL query result.

The system can also suggest appropriate widgets for visualization depending on the returned data. In the example below, the results return the playing times for different Beatles albums (labelled 2). The widget auto suggestion link (labelled 1), provides a list of visualization types on the left. On the right (labelled 3) is the selected bar chart visualization. The auto-suggest facility provides a good way to experiment with different ways of visualizing the data.

![Figure 28: Line chart and pie chart visualizations of the same SPARQL query result.](image)
4.11 Other Linked Data visualization tools

There are other tools available for the visualization of Linked Data. LOD live [13] provides a graph visualization of Linked Data resources. Clicking on the nodes can expand the graph structure. LOD live can be used for live access to SPARQL endpoints. LOD visualization [14] can produce visual hierarchies using treemaps and trees from live access to a SPARQL endpoint.

Figure 30: LOD live [13] and LOD visualization [14] tools.

4.12 Visualizing the Linking Open Data cloud

In chapter 3 (section 3.8.1) we looked at the Linking Open Data cloud diagram that represents connections between the Linked Open Data datasets. This is constructed by hand but there are also tools that can visualize connections between datasets.
Gephi is a platform for visualizing networks, graphs and hierarchies [16]. Gephi can be used to visualize the Linking Open Data cloud. As in the hand-crafted representation, dbpedia.org is the largest node in the network densely connected to other datasets. Colour as well as size is used to represent properties of the dataset. Link length is also used to encode information about the data structure.

Protoviz [17] can also be used to automatically visualize the Linking Open Data cloud. The colour of the node reflects the CKAN rating for the dataset (see section 3.8.1 of chapter 3 for a description of CKAN). The intensity of the colour reflects the number of ratings. The proximity of nodes reflects the level of interconnection between the datasets. Outlying nodes in the graph could indicate broken links to other datasets or a genuine lack of semantic relatedness to other datasets [18]. Clicking on a node takes the user to the CKAN page for that dataset.
4.13 Linked Data reporting and Google’s Structured Data Dashboard

Visualization techniques can also be used in the creation of reports that provide descriptive statistics for a dataset. Often visualizations are displayed in a dashboard that enables user interaction. Several tools exist that can be used for the construction of dashboards including Google Webmaster tools [19], Information Workbench [11] and eCloudManager [20]. We saw earlier some of the visualization capabilities of Information Workbench. The eCloudManager is a specific solution for data centres and cloud management. In the rest of this section we focus on Google Webmaster tools and how they can be used to provide webmasters with information about structured data embedded in websites that is recognised by the Google search engine.

Google Webmaster tools can be used to provide general data about a website, in terms of its traffic and how it is indexed by the Google search engine. As a part of this, Google Webmaster tools provides dashboards on the structured data within a website. The Structured Data Dashboard has three levels. A site-level view aggregates the structured data across all pages according to the classes defined in the vocabulary. An item-type-level view provides separate details for each type of resource. A page-level view shows attributes of every type of resource in a given web page.
Figure 34: A site-level view showing the number of resources of different types that have been detected. The chart shows how the amount of structured data is evolving over time [21].

Figure 35: A page-level view showing the metadata of the imaginary product featured on that page of the website. The detected metadata defines the resource type, image, name and description [21].

4.3 Linked Data Search

4.14 Semantic search process

The figure below provides a way of thinking about semantic search and the potential roles for visualization within that. A user query will be specified possibly as keywords or in natural language. This query can be matched to the underlying graph of data, in order to rank and retrieve results to be returned to the user. Visualization can play two roles. First, visualization can be used to present the search results. For example, search results related to locations may be visualized on a map. Second, visualization can be used to present the query. The user may be able to refine the query by interacting directly with the visualization. Faceted search (discussed more later) would be one example of this. In this case the description of the resources returned in the search can be used to construct a range of filters based on their properties such as time, people and locations. Selecting from the facets limits the part of the data graph of interest.

The user can alter the query by manipulating the visualization. This allows them to modify the underlying SPARQL query with needing to use SPARQL directly.

Figure 36: The role of visualization in search (based on [22]).
4.15 Semantic search

One possible method of input to semantic search is a pseudo natural language query. Entity extraction techniques can then be applied to the text query (a number of entity extraction tools were presented in chapter 3, section 3.9.3).

![User query](image)

"songs written by members of the beatles"

**Entity extraction:** song, written by, member (of), (the) beatles

*Figure 37: Extraction of entities from a pseudo-natural language query.*

By a process termed query expansion, the entities can be mapped to resources within our dataset. For an entity such as "song" this may involve expanding to a number of candidate synonyms and then attempting to map to these to resources in the dataset.

**Query expansion:**

- Candidates:
  - track
  - melody
  - tune

**Entity mapping:** mo:Track

*Figure 38: Mapping the entity “song” to the class Track in the music ontology.*

Similarly, the entity “written by” may be mapped to the entity composer in the music ontology.

**Query expansion:**

- Candidates:
  - composer
  - creator

**Entity mapping:** mo:composer

*Figure 39: Mapping the “written by” entity to the composer property.*

In some cases there may be a direct mapping between an entity and resource in the dataset. For example, the entity “member (of)” may be mapped to the member_of property in the music ontology. This is the inverse of the member property in the music ontology that is used to define artists as members of groups.
Figure 40: Mapping the “member (of)” entity.

A process of contextual analysis is used to decide between candidate mappings. If we start to piece together the parts of the query into the same subgraph we see that the range of the mo:member property is the class mo:MusicGroup. According to this subgraph we would expect the “the Beatles” to be a music group. We would therefore select this expansion of the entity over musical works, posters and books that also have “the Beatles” as their label.

Figure 41: Piecing together a subgraph for the query.

Once the entity mappings have been established, the pseudo natural language query can then be expressed as a SPARQL query (shown visually below) that retrieves tracks (variable ?x) composed by someone (variable ?y) who is a member of The Beatles. (See chapter 2 for an introduction to SPARQL.)

Figure 42: The SPARQL query (shown visually) for tracks by members of The Beatles.

4.16 Semantic search versus SPARQL query

As we can see from the above example, semantic search aims at understanding the meaning of the entities specified in the query. This can involve expanding the entity into a number of candidate synonyms and then matching those to the dataset. Contextual analysis can also be used to decide between alternative meanings by comparing against the subgraph that is being produced by mapping the entities. In some cases (though not seen in the example above) reasoning may be applied in order to derive answers that are not explicitly contained in the data, but can be derived from the data.

Comparing semantic search against querying a dataset using SPARQL we see a number of differences. In semantic search, entities extracted from the search string can be expanded and mapped to resources in the dataset. In SPARQL, generally direct reference will be made to the resources. Semantic search may also allow fuzzy matching in which a weighting or certainty is applied to a mapping between a search term and a resource. Both SPARQL and semantic search work with graph patterns. SPARQL queries are graph patterns applied against the
dataset. In semantic search, the sub-graph built up to represent the query can also be used to analyse context. Finally, semantic search can apply reasoning to identify new paths in the data and derive links between resources not explicit in the dataset.

<table>
<thead>
<tr>
<th>Component</th>
<th>Semantic search</th>
<th>SPARQL query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword or NL / concept matching</td>
<td>Performs entity extraction and matching to formal concepts</td>
<td>Not supported</td>
</tr>
<tr>
<td>Fuzzy concepts/relation/ logics</td>
<td>Allows the application of fuzzy qualifiers as query constraints</td>
<td>Not supported</td>
</tr>
<tr>
<td>Graph patterns</td>
<td>Uses the context and other semantic information to locate interesting sub-graphs</td>
<td>Applies pattern matching</td>
</tr>
<tr>
<td>Path discovery</td>
<td>Finds new interesting links that may lead to additional information</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

**Figure 43:** A comparison of semantic search and SPARQL queries.

### 4.17 Semantic search in Google

Many of the features of semantic search are increasingly found in web search. For a number of queries, Google, as well as providing a page ranked list of links, uses the Google Knowledge Graph to provide direct answers to questions. As we see below, a search for “paul mccartney albums” results in a horizontal panel of albums. On the right we see the disambiguation pane (described in section 3.8.2 of chapter 3 in the context of Rich Snippets) giving additional information about the primary entities extracted from the query and mapped to the Google Knowledge Graph.

**Figure 44:** Semantic search in Google.

### 4.18 Semantic search in DuckDuckGo

Similar semantic features are found in other web search engines such as DuckDuckGo [23] that combines pseudo natural language expressions and semantics
to assist the user in focussing down their search. In the example below a natural language query has produced a list of answers rather than the more conventional list of documents matching the search query.

**Figure 45: DuckDuckGo producing a list of answers.**

DuckDuckGo can also perform disambiguation to offer alternative matches for a search query. In the example below, alternative meanings of the search term jaguar have been offered, grouped into classes such as companies and music. The user can select one of these classes to drill down to their intended meaning.

**Figure 46: Query disambiguation in DuckDuckGo.**

### 4.19 Faceted Search

The Information Workbench, which we saw earlier, is one tool that can provide faceted search over a dataset. Facets are derived from the properties used to describe the resources returned from the search query. From within each facet, the user can select the values. In the figure below the location facet (using the foaf:based_near property) allows the user to drill down to artists (represented as images) from a particular county. The values of the property are sorted according to frequency, therefore giving a higher rank to values that identify the largest number of resources.
Your browser does not support the video tag.

**Movie 5: An introduction to faceted search and the challenges associated with supporting faceted search.**

If the facets are largely independent (i.e. do not demark the same subsets of resources) then a small set of facets can be used to filter quickly a large dataset to a small number of items of interest.

![Faceted search using the Information Workbench.](image)

An interesting challenge when supporting faceted search is determining which of the potentially large number of facets to prioritize in the interface. This can be a particular problem with heterogeneous data, typical of Linked Open Data, where different properties (and therefore facets) apply to different parts of the dataset. Addressing this challenge involves not only prioritizing facets that have good coverage of the resources of interest but also have values that discriminate between them. Facets having values that split the resources into subsets of similar size would then be optimal for filtering.

As the most appropriate facets depend on the properties and values of the resources of interest, they can be expected to change if keyword search is used to select an initial set of resources for faceted browsing.

Another challenge is the real time computation of previews [24]. With a large dataset it can be computationally too expensive to calculate the frequencies for all values of all facets. Counts then need to be predicted from a sample giving the user an indication of what they can expect.

### 4.20 FacetedDBLP

A well-known example of faceted search is FacetedDBLP [25], an interface for browsing DBLP [26], a bibliography of computer science publications. Faceted browsing is generally initiated by a keyword search to identify a region of interest. The returned results (i.e. publication records) can then be filtered using facets to restrict the results in terms of publication year, publication types, venue and authors. The publication records queried via FacetedDBLP are stored in a MySQL database. An RDB2RDF server is used to provide a SPARQL interface to the dataset (see section 3.9.2 of chapter 3 for a description of how to map a relational database to RDF).
4.22 Searching for semantic data

In the examples we have seen so far, search is primarily about finding content. Semantics assist in the search for content by, for example, disambiguating search terms. However, some search engines are directed at finding semantic data. These tools can be used to search for ontologies, vocabularies or particular RDF documents. These tools are therefore aimed more at data specialists rather than the end user.

Figure 48: FacetedDBLP.

Figure 51. Watson semantic web search engine [28].

Vocabularies can be searched using the LOV portal (this was mentioned as a
source of reusable vocabularies in section 3.4 of chapter 3). A keyword search can be used to retrieve a set of vocabularies that can then be filtered using a number of provided facets. The results returned also have a confidence score giving an explicit indication of relevance of that vocabulary to the query. Similar to Watson, the snippets or previews also provide an indication of how the search terms match the vocabulary.

![LOV portal](image)

**Figure 52: Using the LOV portal to search for vocabularies.**

### 4.4 Methods for Linked Data Analysis

#### 4.23 Data analysis methods

In the first part of this chapter we looked at how visualization techniques can be used to reveal patterns in the data. In this section we consider how statistical and machine learning techniques can be used to identify patterns in data. Statistical and machine learning techniques are complementary rather than an alternative to visualization. When analysing data statistically it is good practice to first visualize data to get an idea for what statistical patterns may be expected and therefore which statistical methods to apply. Similarly, visualization is commonly used to explore and describe any pattern detected from statistical analysis and machine learning.

A preliminary step in data analysis is data aggregation. This may be used to merge or summarise the data. This creates the view over the data required for more advanced analysis. This is described in section 4.23.1. Once the appropriate view over the data has been constructed then statistical techniques can be applied, finding for example correlations between the properties. Statistical analysis of linked data is introduced in section 4.23.2. Finally, machine learning can be applied to data in order to learn new groupings or clusters in the data not explicitly defined in the dataset. Machine learning is introduced in section 4.23.3.
4.23.1 Aggregation and filtering of Linked Data   Much of the data aggregation and filtering that might be required to construct the desired view over the data can be carried out using SPARQL. Data can be aggregated using SPARQL functions such as COUNT, SUM and AVG. COUNT returns that number of times the expression has a bound value when the query is run. The functions SUM and AVG return the sum and average bound value. The GROUP BY operator is used in SPARQL to divide the solutions into the groups for which an aggregate value should be calculated. Chapter 2 describes a SPARQL query for returning the playing time of an album by calculating the SUM of its track durations.

SPARQL also has operators that can be used for filtering the results to be returned. FILTER restricts results to those that match a specified triple pattern. HAVING operates in the same way as FILTER but on sets of solutions returned by the GROUP BY operator. See chapter 2 for examples of SPARQL queries using the FILTER and HAVING operators.

<table>
<thead>
<tr>
<th>Features</th>
<th>SPARQL capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregation</td>
<td>Combining aggregate functions (COUNT, SUM, AVG, ...) and GROUP BY operator</td>
</tr>
<tr>
<td>Filtering</td>
<td>Combining projection, FILTER and HAVING operators</td>
</tr>
</tbody>
</table>

Figure 55: Aggregation and filtering using SPARQL.

4.23.2 Statistical analysis of Linked Data   More complex forms of statistical analysis go beyond what can currently be supported by SPARQL. As we have seen above, SPARQL can be used to calculate an average or sum but could not be used to perform other statistical techniques such as regression or analysis of variance. Some approaches [30, 31] can be used to apply statistical techniques directly to data retrieved from a SPARQL endpoint. Without these techniques, the data to be analysed would need to be downloaded in a tabular format and then opened using a statistical package.

R [32] is a free computing package that can be used to carry out a range of statistical techniques including linear and non-linear modelling, time series and analysis of variance. It can also be used to perform machine learning tasks such as clustering and classification. R has a graphical user interface and can also be used to generate visualizations of the data.
Figure 56: The R statistical computing package.

The R for SPARQL package can be used to retrieve data from a SPARQL endpoint over HTTP. A SELECT query returns a result set as what is referred to as a data frame. Visualizations, using some of the techniques described in section 4.6, such as a choropleth map, can also be generated from a data frame.

4.23.3 Machine learning on Linked Data  Machine learning techniques can be used to discover hidden patterns with the dataset. A number of different machine learning techniques can be applied to linked data for different purposes.

Clustering is a technique used to organise a set of items into groups or clusters based on their attributes. In the case of Linked Data, clustering could be used to organise a set of resources on their properties and values. For example, resources representing music albums could be organised. This might reveal certain clusters, based on properties such as duration, artists and year of release. It may be possible to associate names with some of the identified clusters, for example the music albums in a cluster may form a particular genre not explicitly represented in the dataset. This may lead the data analyst to explicitly specify this class in the dataset and assign the albums in the cluster to this class.

Association rule learning is a data mining technique often used to discover relations between variables. Association rules express some regularity or predictability. In the case of Linked Data, association rule learning may discover a pattern. For example if the dataset contained information on albums and which of these are “liked” or owned by a number of people, a rule may be identified that can predict a preference for one album from other albums. This could potentially be used to assert additional relations between the albums included in the rule.

Decision tree learning is a machine learning technique that can be used to define and then predict the classification of a set of items. The decision tree produced by the learning process can predict the classification of an item in answer to a series of questions that lead from the root node to a leaf of the decision tree. In a Linked Data context, decision tree learning could be used to describe and then predict the class membership of a set of instances from their properties and values. For example a subclass of albums representing albums of a particular genre could be predicted from properties such as artist and recording label. This could be used to propose class membership for a set of albums in the dataset that have this information missing.

WEKA [33] is a data mining framework that can be used to apply machine learning techniques to a dataset represented in a tabular format.

The application of machine learning techniques to Linked Data raises a number of challenges [34]. Linked Data is heterogeneous. Different URIs from different datasets may refer to the same resource. Also similar properties but with different constraints may be drawn into the same dataset from different sources. There can also be a high level of redundancy among strongly related parts of the dataset of different origin. This noise and duplication can create performance problems for machine learning algorithms. Another problem for machine learning is the lack of negative examples. For example, decision tree learning can be used to predict membership among disjoint classes. However, in datasets, even if classes are disjoint that is rarely specified. Also the property owl:differentFrom expressing
two resources are not the same can assist the application of machine learning but these negative forms of statement are rarely used.

Machine learning can have a number of applications to Linked Data. First, machine learning can be used to rank nodes according to their relevance to a query. This can be used to prioritize results in a user interface. Second, machine learning can be used for link prediction, proposing new edges between nodes in the RDF graph. Third, entity resolution can be supported, identifying URIs that potentially refer to the same real World object rom similarities in their properties and values. Fourth, techniques such as clustering can be used to propose taxonomies classifying a set of instances. This can be particularly useful when the taxonomy available to classify a set of instances is weak or absent in the dataset.
Chapter 5

Building Linked Data Applications

5.1 Introduction

In this chapter we describe how a Linked Data application is built. This draws on what we have covered in the previous chapters. In chapters 2 and 3 we looked at different methods for consuming Linked Data. Chapter 2 focussed on how to use SPARQL queries to extract data from an RDF dataset. In chapter 3 we looked at other ways in which Linked Data can be made available in order to be consumed by an application. As well as providing a SPARQL endpoint, RDF data can be accessed by dereferencing HTTP URIs, parsing RDFa or reading an RDF dump. In chapter 4 we described ways in which Linked Data can be output for human consumption and the broad range of visualisation tools and techniques that can be used. In this chapter we describe how to bring all of these together in a Linked Data application. Once again we make use of the music application introduced in chapter 1 as our motivating scenario (see figure 1). This chapter looks at the overall composition of the Linked Data architecture and some of the frameworks that can be used when building a Linked Data application.
5.2 Characterization of Linked Data Applications

5.3 Categories of Linked Data applications

Any Linked Data application can be expected to have three parts: Linked Data manager, Linked Data consumer and (Web) User interface. First, the Linked Data Consumer is in charge of retrieving Linked Data from data sources. In cases where the retrieved data is not in the RDF format, wrappers can be used to translate the data into Linked Data. Systems that only consume Linked Data are usually called mashups. Second, the Linked Data consumer is responsible for manipulating the consumed Linked Data in order to produce new Linked Data. Third, the User interface provides a way of interacting with the application. This will often, but not necessarily, be a Web interface. The application may include a user interface supporting, for example, visualization of the data and also an API for programmable interaction with the system.

Linked Data applications themselves can be classified into three main types [1, 2]. First, Linked Data browsers consume Linked Data and present them in a way that allows the users to navigate them. Examples of Linked data browsers such as Sig.ma and Sindice were introduced in Section 10 of Chapter 4. A second category is Linked Data search engines. Unlike conventional search engines that are primarily seen as a means for locating human-readable content, a semantic search engine is used to search for ontologies, vocabularies and RDF documents. Semantic search engines such as Swoogle and Watson were described in section 22 of chapter 4. The third category is Domain-specific Linked Data applications. The Linked Data music application discussed throughout the chapters is one example of a domain-specific application. These applications are built to address a particular range of problems within a specified domain. The vast majority of Linked Data applications fall into this third category. Later in the chapter we will see some examples from different domains.

Linked Data applications can also be categorised on dimensions that describe various technical aspects as to how Linked Data is represented and used (see Figure 2). Linked Data applications can use Semantic Web technologies in a way that is extrinsic or intrinsic. If Semantic Web technologies are extrinsically used, then Linked Data is consumed and processed using APIs. Traditional technologies, such as Relational Database Management Systems (RDBMS) can be used for internal storage and processing. An application may also make intrinsic use of Semantic Web technologies, for example, storing the internal state of the application in a triplestore rather than using RDBMS. A single application may also combine components that make intrinsic and extrinsic use of Semantic Web technologies.

A Linked Data application can also be classified as to whether it consumes Linked Data, produces Linked Data or both. As described earlier, a Linked Data application that only consumes Linked Data can be more appropriately described as a mashup. Applications can also vary in terms of their semantic richness. A relatively shallow representation of semantics may be used, incorporating for example simple taxonomies. For shallow semantics, the RDF and RDFS vocab-
ularies would probably suffice, as they enable the expression of class hierarchies, class membership and properties. Strong semantic richness, expressing more complex relationships between resources would require a representational formalism such as one of the variants of OWL (Web Ontology Language).

Linked Data applications can also be classified as to whether they are isolated from, or integrated with, external vocabularies. A Linked Data application that used its own vocabulary, distinct from other available datasets would be described as being isolated. As described in Chapter 3, Linked Data should wherever possible reuse existing vocabularies or express relationships (for example using owl:sameAs) to other published vocabularies. An application that extensively reuses and interlinks vocabularies would be described as an integrated application. Even an isolated application could become integrated if the vocabulary used within the application is published, and this vocabulary is used to interlink this dataset with others.

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Levels</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantic technology depth</td>
<td>Extrinsic</td>
<td>Use of semantics on the surface of the application.</td>
</tr>
<tr>
<td></td>
<td>Intrinsic</td>
<td>Conventional technologies (e.g., RDBMS) are complemented or replaced with SW equivalents.</td>
</tr>
<tr>
<td>Information flow direction</td>
<td>Consuming</td>
<td>LD is retrieved from the source or via a wrapper.</td>
</tr>
<tr>
<td></td>
<td>Producing</td>
<td>Publishes LD (in RDF-based formats).</td>
</tr>
<tr>
<td>Semantic richness</td>
<td>Shallow</td>
<td>Simple taxonomies, use of RDF or RDFS.</td>
</tr>
<tr>
<td></td>
<td>Strong</td>
<td>High level representation formalisms (OWL variants)</td>
</tr>
<tr>
<td>Semantic integration</td>
<td>Isolated</td>
<td>Creation of own vocabularies</td>
</tr>
<tr>
<td></td>
<td>Integrated</td>
<td>Reuse of information at schema or instance level</td>
</tr>
</tbody>
</table>

Figure 2: Categorisation of Semantic Web Applications [2].

5.3 Examples of Linked Data applications

The following sections outline examples of Linked Data applications in a range of domains.

5.3.1 Governmental Open Data

The website data.gov.uk provides a data catalog of UK government information. Over 9000 datasets are available on the site covering themes such as transport, government spending, health, crime and the economy. All of the data is openly available but only a small proportion of it is available via Linked Data technologies. The majority of the datasets are published in tabular format such as CSV. However, a number of Linked Data applications have been built on top the published datasets [3].
Figure 3: The data.gov.uk catalog of UK governmental information [4].

The data.gov portal is the equivalent website in the US. This contains a much larger number of datasets. However, currently none of the datasets are published following Linked Data standards. Some Web and mobile applications have been developed on top of these datasets.

Figure 4: The data.gov catalog of US governmental information [5].

5.3.2 BBC Dynamic Semantic Publishing  
The BBC Dynamic Semantic Publishing (DSP) [6] architecture aims at automating the aggregation or publishing of interrelated content within the BBC portal. This initiative started with sports content before moving to other areas of the BBC. This functionality allows the user of BBC content to navigate from an article to semantically related content. For example, an article about a football match might mention the teams, key players, the managers and the ground at which the match was played. From this article the reader could follow links to further articles related to any of those entities, and then navigate further from there. The reader could therefore initially find out about the game, move onto a biographical article about one of the managers and then start reading about a previous football team that they managed.
The links between the articles, rather than being manually specified by a journalist, are generated based on semantic annotations associated with each article. The Graffiti tool is used to add these semantic annotations, associating the articles with Linked Data concepts such as people and locations (see Figure 5). An OWLIM triplestore is used to store and reason over the RDF data representing the articles and associated data.

As a triplestore is used to drive the data presented on the website, this application can be classified as intrinsic in terms of its use of Semantic Web technologies. However, its use of Semantic Web technologies is not particularly visible to the user, who interacts with it as a conventional website. This can be contrasted with Linked Data browsers that directly expose the RDF data to the user.

5.3.3 Research Space  
ResearchSpace [7] is an environment for conducting cultural historical research. It provides RDF datasets and tools for investigating cultural objects such as paintings. Associated concepts such as artists and locations are semantically represented. A number of tools are provided allowing users both to access the data and also contribute new data in the form of RDF annotations. Figure 6 shows the Image Annotation tool. This can be used to describe different features of an artwork such as the subject and its location.

Figure 6: Image annotation.
Your browser does not support the video tag.

Movie 1: ResearchSpace as an example Linked Data application.

The architecture of ResearchSpace is shown below. An SQL database is used to store documents and media. A triplestore is used to store annotations and associated vocabularies. The Nuxeo content management platform delivers the user interface and has a series of plugins to support annotation and communication with the triplestore.

![ResearchSpace Architecture](image)

**Figure 7: ResearchSpace architecture [8].**

ResearchSpace also offers a faceted search interface to the cultural resources. Facets are driven by the semantic description of the artefacts according to the CIDOC CRM ontology [9]. This allows filtering of resources according to a number of associated concepts such as object type (e.g. sketch, painting) location and creator. See Section 19 of Chapter 4 for more examples of faceted search over RDF data.

![ResearchSpace Faceted Search](image)

**Figure 8: The ResearchSpace CRM Search System [10].**

ResearchSpace can be classified as a hybrid tool that intrinsically and extrinsically combines Semantic Web technologies. For example it uses a combination of a triplestore and relational database to store content. It also both consumes and produces Linked Data in combining existing data about museum objects with additional annotations. Semantic descriptions exploit the structure of the CIDOC CRM ontology to express the different features of the described objects.
5.3.4 **Open Pharmacology Space**  
Open Pharmacology Space is a Linked Data application that aims to provide a semantic research environment for pharmacology, comparable to the support offered by ResearchSpace for cultural research. Open Pharmacology Space integrates different data sources and provides an API to access the aggregated data. Three tools that have been built on top of the Open Pharmacology Space are Open PHACTS Explorer, ChemBioNavigator and PharmaTrek. Open PHACTS Explorer provides essentially a Linked Data browser of the RDF data. ChemBioNavigator is a tool for the visualisation of the chemical and biological space of a molecule group. PharmaTrek is a visualisation tool specifically designed for use with the ChEMBL biochemical database.

![Figure 9: Open PHACTS Explorer, ChemBioNavigator and PharmaTrek][11]

The architecture of Open Pharmacology Space is shown below. A number of data sources are consumed in a triplestore referred to as an RDF data cache. Some of these sources have to be harvested and transformed by the application. The semantic data workflow engine drives the logic of the application. Data produced by Open Pharmacology Space can be accessed via an API or a SPARQL endpoint. This data is made available according to a unified Open Pharmacology Space data model.

![Figure 10: Open Pharmacology Space architecture][12]

The Open Pharmacology Space makes both intrinsic and extrinsic use of Semantic Web technologies. The storage of data in RDF format and the use of semantics to represent workflow are intrinsic aspects of the application. Open Pharmacology Space is also both a consumer and producer of RDF data. It consumes vocabularies of varied richness and produces data according to its own unified data model.

Your browser does not support the video tag.

*Movie 2: Screencast of the Linked Data API (LDA).*
5.3.5 eCloudManager  eCloudManager is a data center management application. A data centre typically comprises a large number of hardware components from a number of manufacturers. Each of these components will have a reporting system, custom-made by their manufacturer. Together these components provide metadata information on all of the hardware running in the data centre, its location and status. The physical hardware itself will generally be used to run a number of virtual machines that may be migrated across different hardware platforms. The hardware and configuration of virtual machines will deliver a number of software applications each with their own licensing arrangements and separate data stores. A data center therefore has a number of tools that can all provide partial information about the overall state of a data center.

![Diagram of a typical data center](image)

Figure 11: A typical data center comprising hardware, virtual machines and software applications [13].

The aim of eCloudManager is to integrate different hardware and software components into a single semantic view. This views brings together the hardware components of various manufacturers, the virtualisation layer delivered by the hardware and the supported range of software applications. The eCloudManager also provides a business view indicating which departments are responsible for different services or hardware, and which customers may be affected by the failure of those components.

The overall integrated view provided by eCloudManager can be filtered depending on the user’s interest. The user could create a view on the system dedicated to the storage infrastructure comprising hardware components from multiple manufacturers. Similar views can also be created on the virtualisation, application or project levels.
5.3 Architecture of Linked Data Applications

5.4 Architecture of Linked Data applications

In the previous section we saw a number of example Linked Data applications. Each application not only had a different purpose but also made different architectural decisions in terms of how the data was accessed, processed and stored. In this section we present generic architectural patterns of Linked Data applications and different design decisions associated with these patterns.

The term software architecture describes the components of a software system and the relationships between those elements. For a web-based system the components could include software modules, databases and web servers. Some parts of the architecture may be legacy components brought forward from previous systems. The relationships between the components of a system indicate which components communicate during operation of the system and the mechanisms by which that communication takes place. As well as specifying the structure of the system, the software architecture also stipulates a set of design practices to be followed in order to create and maintain the architecture.

5.5 Multitier architecture

An important architectural pattern used in system development is the multitier architecture. A multitier architecture separates functionality into a number of layers from low-level data storage through to user interaction components. This architecture is commonly used for many kinds of web application. As many Linked Data applications are also web applications, they tend to conform to this architectural approach.
An important advantage of the tiered architecture is that it logically separates the functionality of the system into a series of layers and specifies the communication between those layers. This separation makes it far easier to replace a layer of the architecture or reuse a layer of an existing architecture in a new application. For example, an application may have a layer or tier dedicated to data storage. This functionality may be provided by a particular database or triplestore. The storage layer could be reused in an alternative application or replaced by another storage layer providing the same functionality and communication with other layers.

The most commonly used multitier architecture is the three-tier architecture. First, a presentation tier provides a user interface that can accept user input and render results in a human-readable form. Second, a logic tier implements the business logic of the application. This takes the available data and analyses and transforms it to meet the needs of the user. Third, a data tier stores the underlying data in a form independent from the business logic applied to it in the application. Figure 13 illustrates our music example as a three-tier architecture.

The presentation tier handles user queries and the returned outputs including textual results and visualisations. The logic tier transforms user queries into SPARQL queries and aggregates the RDF results. The data tier is responsible for storage and in this case uses a triplestore. One important aspect to note in the case of Linked Data applications is that the dividing line between the data tier and the logic tier may not be so clear-cut. If a relational database is used as a storage layer, then all processing of the data beyond the returning of results to a database query is done in the logic layer. However, triplestores are capable of performing various types of reasoning and therefore in some cases a significant part of the business logic can be carried within the triplestore. For reasons of performance it is advantageous to perform reasoning on a lower level as possible.

![Figure 13: Music example of the three-tier architecture.](image)

Figure 14 shows the architecture of our music application with a particular emphasis on the components within the data tier. The presentation layer produces the kinds of visualisations we saw in Chapter 4. The logic layer processes data for presentation. The data tier as well as implementing some of the logic does much more work than just data storage. Data stored within the data tier may be consumed from a number of sources. These may be consumed from SPARQL endpoints or RDF dumps. Wrappers may be required to covert the data into an appropriate format. As described in chapter 3 R2RML may be used to transform data from a relational database into RDF.

Also, as covered in Chapter 3, the retrieved data may use different vocabularies on the schema level and may also on the instance level have different individuals
referring to the same thing. As described in Chapter 3, languages such as SKOS can be used to express relationships across vocabularies and the owl:sameAs property can be used to relate resources. Chapter 3 also describes how tools such as the SILK framework, could be used to identify and express relationships across datasets.

Some data cleansing may also be required, for example, to identify ambiguities between names of the resources within the datasets and to fix these ambiguities. Therefore, vocabulary mapping, interlinking and cleansing need to be carried out to produce a consistent dataset. As well as being used by the logic layer, a direct facility to republish the data may be provided.

Figure 14: General architecture of Linked Data applications.

5.6 Architectural patterns

The data consumed and integrated within the application may be accessed in a number of ways. Three main architectural patterns can be identified. First, there is the **crawling pattern**, in which data is loaded in advance. The data may also need to be transformed as described above. The data is managed in one triplestore so that it can be accessed efficiently. The disadvantage of this pattern is that the data might not be up to date. Second, there is the **On-The-Fly Dereferencing Pattern**. Here, URIs are dereferenced at the moment that the application requires the data. This pattern retrieves up to date data but performance is affected when the application must dereference many URIs. Third, there is a **(Federated) Query Pattern** in which complex queries are submitted to a fixed set of data sources. This approach enables applications to work with current data directly retrieved from the sources. However, finding optimal query execution plans over a large number of sources is a complex problem. This third pattern in specific situations can offer a way of accessing up-to-date data with adequate response times.

5.7 Data layer

In the data tier new Linked Data may be consumed from a SPARQL endpoint in RDF. As discussed above, if the data is in another form such as CSV (Comma Separated Values) then a wrapper would be used to translate the data to RDF.
Linked Data applications may implement a Mediator-Wrapper Architecture to access heterogeneous sources in which wrappers are built around each data source in order to provide an unified view of the retrieved data. The Mediator-Wrapper Architecture could be used with any of the three architectural patterns (i.e. crawling pattern, on-the-fly dereferencing pattern, federated query pattern). The most appropriate patterns will depend on a number of factors such as the number of sources that need top be accessed, how up-to-date the data is required to be and the speed of response required by the data layer.

A number of tools are available that can be used when implementing a data access component. Linked Data Crawlers are web crawlers designed to harvest RDF data. Linked Data client libraries support the access and traversal of Linked Data. SPARQL Client Libraries provide an API for accessing SPARQL endpoints. Federated SPARQL Engines provide a single access point for multiple heterogeneous data sources. Finally, Search Engine APIs such as Sindice (see chapter 4, section 10.2) support semantic searches that return RDF documents.

<table>
<thead>
<tr>
<th>Mechanisms</th>
<th>Tools (Examples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked Data Crawlers</td>
<td>LDspider <a href="https://code.google.com/p/ldspider/">https://code.google.com/p/ldspider/</a></td>
</tr>
<tr>
<td>Linked Data Client Libraries</td>
<td>Semantic Web Client Library <a href="http://wdl03.informatik.uni-weimar.de/ldg/semwebclient.html">http://wdl03.informatik.uni-weimar.de/ldg/semwebclient.html</a></td>
</tr>
<tr>
<td></td>
<td>The Tabulator <a href="http://www.w3.org/2005/ajr/tab">http://www.w3.org/2005/ajr/tab</a></td>
</tr>
<tr>
<td></td>
<td>Moriairy <a href="https://code.google.com/p/moriairy/">https://code.google.com/p/moriairy/</a></td>
</tr>
<tr>
<td>Federated SPARQL Engines</td>
<td>ANAPSID <a href="https://github.com/anapsid/anapsid">https://github.com/anapsid/anapsid</a></td>
</tr>
<tr>
<td></td>
<td>SPLENDOR <a href="https://code.google.com/p/rdffederator/">https://code.google.com/p/rdffederator/</a></td>
</tr>
<tr>
<td>Search Engine APIs</td>
<td>Sindice <a href="http://sindice.com/developers/spi">http://sindice.com/developers/spi</a></td>
</tr>
</tbody>
</table>

Figure 15: Linked Data access mechanisms.

The integrated dataset can then be kept in a local triplestore. A triplestore is required unless data is retrieved on-the-fly, and discarded just after building a response to the user request. A number of commercial or free RDF triplestores are available including OWLIM [14], Jena TDB [15], Cumulus [16], AllegroGraph [17], Virtuoso Universal Server [18] and RDF3x [19]. As described in Chapter 3, SPARQL endpoints and RDF dumps can be used to make RDF data available. As we will see later, data can also be made available via APIs or using functionality provided by your chosen application framework.

5.8 Logic and presentation layers

Once the integrated data is available on-the-fly or in a triplestore, it can be used and accessed by the logic and presentation layers. As mentioned above, some of the logic may be implemented in the data layer by reasoning over the triplestore. Other forms of processing that cannot be implemented on the data later are carried out in the logic layer. As described in section 23 of Chapter 4, this may involve the application of statistical or machine learning processes to make inferences from the data. Other forms of business logic such as workflow are also implemented within this layer. Finally, the presentation layer displays the information to the user in various formats, including text, diagrams or other types of visualization techniques. A range of tools and formalisms for the display
of information were outlined in Chapter 4.

5.4 Using Web APIs

5.16 HTTP communication

If a Linked Data application is built on the web then it may use Web APIs to either provide data or consume data from other sources. The HTTP protocol is the fundamental technology on which Web APIs are built. The serving of documents on the Web (for example the serving of an HTML page to a web browser) is carried out using the HTTP protocol.

HTTP communication is based on an interaction that involves a series of requests and responses. A client sends a request to a server. The server sends back a response to the client.

![Request-Response interaction](image)

**Figure 34: Request-Response interaction.**

5.16.1 HTTP Request  
Each HTTP request contains a method, URI, Header and optionally a body. The *method* indicates the type of the request that the server should perform. The most familiar types of HTTP request are *GET* and *POST*. A GET request means the client wants to retrieve content. The URI sent with the GET request is the resource from which the content should be retrieved. A POST request is used to send data. The accompanying URI indicates where the data should be sent. Web forms generally use POST requests to send data to the server.

The other types of HTTP request are used more broadly in web-based client-server communication but not necessarily important when using a web-browser to retrieve and send content. A *PUT* request to used to store data at the specified URI. A *DELETE* request is used to delete the specified URI.

Other types of HTTP request include HEAD, TRACE, CONNECT, OPTIONS and PATCH. For example, a *HEAD* request is used to retrieve header information. A *TRACE* request is used to allow the client to see what the server is receiving. This is generally used for diagnostics.

As well as the method and URI a HTTP request contains *header* information that gives additional detail about the HTTP request. A *body* is required with POST and PUT methods and expresses the data that is to be sent to, or stored at the specified URI. For example, if you submit a web form, then the data entered into the form can be represented in the body of the HTTP request.
5.16.2 HTTP Response  A response to a HTTP request contains a numerical response code, a header and optionally a body. The response code gives overall status information on how the request is being handled. The response code is a three digit number beginning with a 1, 2, 3, 4 or 5. Response codes beginning with 1 are provisional responses indicating that the request has been received and is being acted upon.

Requests starting with 2 indicate that the request has been successfully received, understood and accepted by the server. Codes beginning with 3 indicate that further action needs to be taken by the client that issued the initial request. Codes beginning with 4 indicate that the request is erroneous and cannot be met by the server. The most commonly seen response code beginning with 4 is the 404 response code. The 404 response code informs the client that the request was erroneous because it asked for a resource that does not exist on the server. Finally, codes beginning with 5 indicate that there is an error, but in this case the problem is with the server and it is unable to fulfil the valid request.

5.16.3 HTTP Request Response pattern  We can now put the request and response together to illustrate the HTTP request-response pattern, in which a client requests and then receives web page. The client issues a GET request with the URI for the Wikipedia page about The Beatles. This tells the server that the client wants to retrieve information from the provided URI. The final response from the server has a response code 200 (indicating that the request has succeeded) and returns the HTML page about The Beatles.

5.16.4 HTTP content negotiation  When requesting data in a particular media from DBpedia rather than Wikipedia, the communication pattern between client and server is more complex. In this case the URI refers to the concept of The Beatles rather than any particular document describing The Beatles. As shown below, the request uses the method GET and declares that a response is required in HTML format. The URI refers to the concept of The Beatles, which is not a resource in HTML format. The server responds with a code of 303 and another URI. This tells the client to instead make a request for HTML from this alternative URI. The client then makes a second request (not shown) for HTML using the new URI and receives a HTML page about The Beatles with a response code of 200.

Figure 35: The HTTP request-response pattern.
Chapter 5. Building Linked Data Applications

Figure 36: Content negotiation when requesting HTML using a Linked Data URI.
In the above HTTP conversation the client requested information in a certain format using a particular URI. The server responded with an alternative URI. The client then made a second request with the new URI. The server responded with the requested information. This conversation between client and server to determine the correct resource is called content negotiation and is often abbreviated to conneg.

The reason this conversation to determine the appropriate content is required is that different types of content can potentially be returned associated with the same resource. If information is being accessed via a web browser, then HTML is likely to be the preferred format. If the client is a Linked Data application consuming data, then the requested information about The Beatles will be preferred in an RDF format such as Turtle (see chapter 1 for more information about the Turtle format). In the figure below we have a GET request using the same URI as in the HTML example above. However, this time the client requests text/turtle rather than text/html. The server responds with the status code 303 (i.e. see other) and a URI where the information can be accessed in Turtle format.

Figure 37: Requesting data in Turtle format
The client then issues a second request and this time retrieves the data in Turtle format along with a status code of 200. This approach is routinely used to publish Linked Data, in which a series of URIs name concepts used in the Linked Data application, such as The Beatles and Paul McCartney. Requests for data to a Linked Data URI are directed to another URI depending on the data format requested. Multiple RDF formats of the same data may be made available such as RDF/XML and Turtle.

Figure 38: Retrieving data in Turtle format.
5.17 Web APIs

The HTTP conversations we have seen above provide the foundation for Web APIs. Web APIs are particularly important when a Linked Data application needs access to data that is being dynamically created. If a Linked Data application is using data about music artists and releases from the 1960s then few updates to the stored data will be required. If dynamically changing data is being used such as current weather conditions or traffic levels, then web APIs can be used to provide access to new data. A Web API can provide a range of functionalities, giving access to views on that data and also transforming the data in ways of use to other applications.

Over the past few years there has been a huge growth in the number of Web APIs, though most of these are not Linked Data Web APIs. The most common form of API is the REST (Representational State Transfer) API. REST is an architectural style that uses the HTTP protocol for communication between client and server. The Programmable Web is a general directory of Web APIs. This allows providers to register their API and other application developers to search for available APIs. The vast majority of Web APIs registered with the Programmable Web use the REST model.

Figure 39. Growth in WEP APIs [28].

5.18 Richardson Maturity model for Web Services

The Richardson Maturity Model provides a way of thinking about the main elements that make up a REST architecture. The model is divided into a number of layers, each layer being a precondition of the one above. All of these layers can be seen as a necessary requirement of a REST architecture. Starting from the lowest level, we have resources and their URIs. For example, as we saw above, The Beatles resource is identified by the URI http://dbpedia.org/resource/The_Beatles. The second level is HTTP verbs. These are essentially the HTTP methods (such as GET and POST) we saw earlier. These define actions to be carried out such as sending or retrieving data. The third level is known by the acronym HATEOAS (Hypertext As The Engine Of Application State). This describes the Hypermedia controls, in other words, the higher-level functions provided by the Web API such as inspecting and modifying the music releases associated with a music artist.
On the lower Resource level, the Web API just makes available URIs that identify resources. As we saw in the previous section, a Linked Data URI may direct the client to alternative representations of that resource in, for example, Turtle or RDF/XML. On the second level, the HTTP verbs are methods that can act on those resources. Different methods can be used such as GET, POST and DELETE. The METHOD option is used by a client to get information about the types of request currently available. The server responds requests with an appropriate code such as 200 (OK), 303 (see other) or 404 (not found). The methods and their associated response codes give us a standardized form of communication in terms of the HTTP protocol. This therefore defines the different types of request that a client can make and the ways in which a server can respond to those requests.

HTTP verbs can be characterised as to whether they are safe and whether they are idempotent. A verb is characterised as safe if it cannot change the resource addressed on the server. For example, GET is safe because it merely retrieves information from the resource. It does not attempt to modify the resource. A HTTP verb is characterised as idempotent if the effect of sending one request is identical to sending multiple identical requests. The method DELETE is idempotent. Sending a single request to DELETE a resource will remove it form the server. Sending the DELETE request multiple times results in the same state. The resource stays deleted no matter how many times the request is sent. The only difference when sending multiple DELETE requests is that once the resource has been deleted the server will respond with the code 404 (not found) rather than 200 (OK) as the resource is no longer available for deletion.

<table>
<thead>
<tr>
<th>HTTP Verb</th>
<th>Effect</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>retrieve the representation of a resource identified with a URI</td>
<td>safe, idempotent</td>
</tr>
<tr>
<td>PUT</td>
<td>create or overwrite a resource identified by a client-generated URI</td>
<td>idempotent</td>
</tr>
<tr>
<td>POST</td>
<td>create a resource identified by a server-generated URI</td>
<td></td>
</tr>
<tr>
<td>DELETE</td>
<td>delete a resource (or its representation) identified with a URI</td>
<td>idempotent</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>request for information about the available communication options</td>
<td>safe, idempotent</td>
</tr>
</tbody>
</table>

On the third level, HATEAOs describes how we use resources to drive the application through a series of dynamic states. For example, a client may send an order for a music album to a Web API. In this case the request will need to identify the album to be purchased, such as Revolver by The Beatles. The Web API creates a new resource to identify the order and sends a response to the client. The response indicates the resource identifying the order and the options
available to the client. In this case, the server indicates that the order is awaiting payment and the price to be paid. The client could then respond by sending payment details such as a credit card number of the resource created for the order. If payment was accepted, then the Web API may trigger a physical shipment of the album or send the customer details of where a digital copy of the music can be accessed.

Figure 42: Transitions through states in a music ordering process.
Chapter 6

Text and image sources, contributors, and licenses

6.1 Text

- Course:How to use Linked Data/Introduction and Application Scenarios/Introduction Source: https://en.wikitolearn.org/Course%3AHow_to_use_Linked_Data/Introduction_and_Application_Scenarios/Introduction?oldid=4610 Contributors: WikiToBot and Move page script


- Course:How to use Linked Data/Introduction and Application Scenarios/Introduction to Linked Data Source: https://en.wikitolearn.org/Course%3AHow_to_use_Linked_Data/Introduction_and_Application_Scenarios/Introduction_to_Linked_Data?oldid=10359 Contributors: Toma.luca95, WikiToBot, M.bona, Move page script and Anonymous: 1

- Course:How to use Linked Data/Querying Linked Data/Introduction Source: https://en.wikitolearn.org/Course%3AHow_to_use_Linked_Data/Querying_Linked_Data/Introduction?oldid=4634 Contributors: WikiToBot and Move page script

- Course:How to use Linked Data/Querying Linked Data/Introduction to SPARQL Source: https://en.wikitolearn.org/Course%3AHow_to_use_Linked_Data/Querying_Linked_Data/Introduction_to_SPARQL?oldid=10302 Contributors: WikiToBot, M.bona, Kugnant, Move page script and Anonymous: 1

- Course:How to use Linked Data/Querying Linked Data/Updating Linked Data with SPARQL 1.1 Source: https://en.wikitolearn.org/Course%3AHow_to_use_Linked_Data/Querying_Linked_Data/Updating_Linked_Data_with_SPARQL_1.1?oldid=4638 Contributors: WikiToBot, M.bona, Move page script and Anonymous: 1

- Course:How to use Linked Data/Providing Linked Data/Introduction Source: https://en.wikitolearn.org/Course%3AHow_to_use_Linked_Data/Providing_Linked_Data/Introduction?oldid=4622 Contributors: WikiToBot and Move page script

- Course:How to use Linked Data/Providing Linked Data/The Linked Data Lifecycle Source: https://en.wikitolearn.org/Course%3AHow_to_use_Linked_Data/Providing_Linked_Data/The_Linked_Data_Lifecycle?oldid=4630 Contributors: WikiToBot, Move page script and Anonymous: 1

- Course:How to use Linked Data/Providing Linked Data/Creation of Linked Data Source: https://en.wikitolearn.org/Course%3AHow_to_use_Linked_Data/Providing_Linked_Data/Creation_of_Linked_Data?oldid=4618 Contributors: WikiToBot, Move page script and Anonymous: 1
6.2 Images


6.3 Content license

- [Project:Copyright Creative Commons Attribution Share Alike 3.0 & GNU FDL]
- Creative Commons Attribution-Share Alike 3.0