sparqling-genomics

https://www.sparqling-genomics.org

version 0.99.11, June 19, 2020

Contents

Chapter 1

Getting started

SPARQLing genomics is a combination of tools and practices to create a knowledge graph to make *discovering*, *connecting*, and *collaborating* easy.

1.1 Prerequisites

The programs provided by this project are designed to build a knowledge graph. However, a knowledge graph store (better known as an RDF store) is not included because various great RDF stores already exist, including Virtuoso, 4store and BlazeGraph. We recommend using one of the mentioned RDF stores with the programs from this project.

Before we can use the programs provided by this project, we need to build them. The build system needs GNU Autoconf, GNU Automake, GNU Make and pkg-config. Additionally, for building the documentation, a working LaTeX distribution is required including the `pdflatex` program. Because LaTeX distributions are rather large, this is dependency is optional, at the cost of not being able to (re)generate the documentation.

Each component in the repository has its own dependencies. Table 1.1 provides an overview for each tool. A • indicates that the program (row) depends on the program or library (column). Care was taken to pick dependencies that are widely available on GNU/Linux systems.

	C compiler	raptor2	libxml2	HTSLib	zlib	GNU Guile	GnuTLS	LaTeX
`vcf2rdf`	•	•		•			•	
`bam2rdf`	•	•		•			•	
`table2rdf`	•	•			•		•	
`json2rdf`	•	•			•		•	
`xml2rdf`	•	•	•		•		•	
`folder2rdf`						•		
`sg-web`						•	•	
`sg-web-test`						•	•	
`sg-auth-manager`						•	•	
Documentation								•

Table 1.1: External tools required to build and run the programs of this project.

The manual provides example commands to import RDF using cURL.

1.2 Installing the prerequisites

1.2.1 Debian

Debian includes all tools, so use this command to install the build dependencies:

```
apt-get install autoconf automake gcc make pkg-config zlib1g-dev  \
                guile-2.2 guile-2.2-dev libraptor2-dev libhts-dev \
                texlive curl libxml2-dev gnutls-dev
```

This command has been tested on Debian 10. If you're using a different version of Debian, some package names may differ.

1.2.2 CentOS

CentOS 7 and 8 do not include `htslib`. Follow the instructions on the `htslib` website[1] to build `htslib` from source.

All other dependencies can be installed using the following command:

```
yum install autoconf automake gcc make pkgconfig guile guile-devel \
            raptor2-devel texlive curl libxml2-devel gnutls-devel
```

1.2.3 GNU Guix

For GNU Guix, use the `environment.scm` file to set up the development environment:

```
guix environment -l environment.scm
```

1.2.4 MacOS

The necessary dependencies to build `sparqling-genomics` can be installed using homebrew:

```
brew install autoconf automake gcc make pkg-config guile \
             htslib curl raptor libxml2 zlib gnutls
```

Due to a missing LaTeX distribution on MacOS, the documentation cannot be build.

1.3 Obtaining the source code

The source code can be downloaded at the Releases[2] page. Make sure to download the `sparqling-genomics-0.99.11.tar.gz` file.

Or, directly download the tarball using the command-line:

```
```

[1] https://www.htslib.org/

[2] `https://github.com/UMCUGenetics/sparqling-genomics/releases`

```
curl -LO https://github.com/UMCUGenetics/sparqling-genomics/releases/\
download/0.99.11/sparqling-genomics-0.99.11.tar.gz
```

After obtaining the tarball, it can be unpacked using the `tar` command:

```
tar zxvf sparqling-genomics-0.99.11.tar.gz
```

1.4 Installation instructions

After installing the required tools (see section 1.1 'Prerequisites'), and obtaining the source code (see section 1.3 'Obtaining the source code'), building involves running the following commands:

```
cd sparqling-genomics-0.99.11
autoreconf -vif # Only needed if the "./configure" step does not work.
./configure
make
make install
```

To run the `make install` command, super user privileges may be required. This step can be ignored, but will keep the tools in the project's directory. So in that case, invoking `vcf2rdf` must be done using `tools/vcf2rdf/vcf2rdf` when inside the project's root directory, instead of "just" `vcf2rdf`.

Alternatively, specify a `--prefix` to the `configure` script to install the tools to a user-writeable location.

Individual components can be built by replacing `make` with the more specific `make -C <component-directory>`. So, to *only* build `vcf2rdf`, the following command could be used:

```
make -C tools/vcf2rdf
```

1.5 Using a pre-built Docker image

A pre-built Docker container can be obtained from the release page. It can be imported into docker using the following commands:

```
curl -LO https://github.com/UMCUGenetics/sparqling-genomics/releases/\
download/0.99.11/sparqling-genomics-0.99.11-docker.tar.gz
docker load < sparqling-genomics-0.99.11-docker.tar.gz
```

The container includes both SPARQLing genomics and Virtuoso (open source edition).

Chapter 2

The knowledge graph

In this manual we define a *knowledge graph* as a collection of facts stated in a coherent way so that inferences can be drawn from them. We implement a knowledge graph using the Resource Description Framework (Lassila, 1999), hereafter referred to as RDF. The knowledge graph is the main value obtained from this project.

The programs from chapter 4 'Command-line programs' read data in a domain-specific format, and translate it into *facts* in the form `subject` → `predicate` → `object`, which is the form of an RDF triplet.

Once all desired data is described as RDF triplets, we can use the SPARQL Protocol and RDF Query Language ("SPARQL 1.1 Overview", 2013), better known as simply "SPARQL", to extract knowledge from the facts.

2.1 A layered inference system

Stating facts as RDF is a necessary first step to a more powerful inference system. To understand the knowledge graph and its intended use, we can think of the knowledge graph as having multiple layers. Figure 2.1 displays a small example of two layers. The figure we shows knowledge from two separate sources (gene locations and genomic variants) from which we can derive new knowledge in an inference layer.

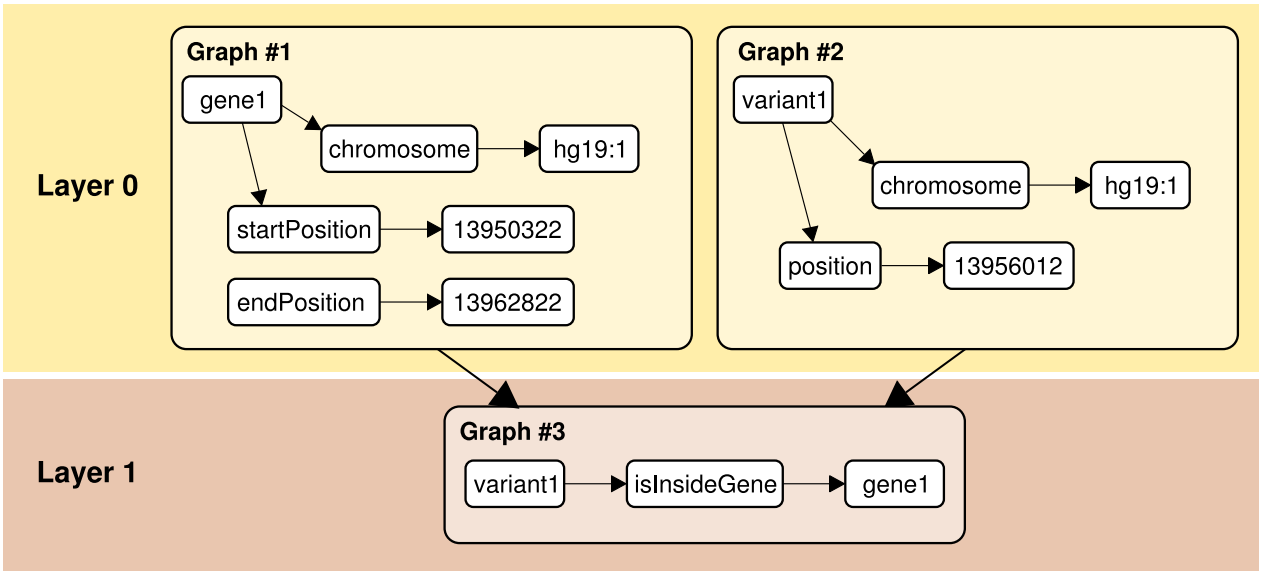

Figure 2.1: *An illustration of knowledge layers.*

Separating layers in graphs makes describing the flow of information easier, because the knowledge from a layer 1 graph can oftentimes be summarized as a query that described the relationship between the layer 0

graph(s) and the newly created layer 1 graph.

Programs can operate on multiple layers of knowledge. A program operates on the first layer (layer 0) when it translates a non-RDF format into RDF. These programs (re)state observations. In the second layer (layer 1) and up, we find programs that operate on facts from layer 0 and generate inferences.

From a computational perspective, these inferences allow programs to take shortcuts, and therefore answer questions (called *querying*) faster. The performance of querying the graph can therefore be tuned by cleverly stating facts. For example, by using the inference drawn in figure 2.1, a query asking for "all variants in a gene" no longer needs to compute whether a variant is inside any of the genes. The query planner can also narrow the search space by only considering the variants in the layer 1 graph.

From a data access perspective, these inferences allow fine-grained access to knowledge. For example, access to a layer 1 graph can be given, but not to its underlying layer 0 graph(s). Properties can be removed (like patient identifiers), or made less precise (only tell that there is a variant in a particular gene, rather than the variant details).

The knowledge graph contains two types of nodes; uniquely identifiable names having a symbolic value (1), and literal values like numbers and text (2). Nodes with a symbolic value, or simply *symbols* can be used to add context to. For example, consider the following statement: `<H>` → `<electronegativity` → `2.20`. We can extend our knowledge about `<H>` by also stating: `<H>` → `<numberOfElectrons>` → `1`. The identifier `<H>` by itself has no meaning. But that there is one "thing" for which both properties `<electronegativity>` and `<numberOfElectrons>` have a certain value provides insight into a deeper structure.

We could go on and find more entities for which the properties `<electronegativity>` and `<numberOfElectrons>` are bound to specific values. We can then identify the group of entities by a single *type* to make it easier to talk about. In the example, the type `<ChemicalElement>` may be suitable. So we would add the triplet: `<H>` → `rdf:type` → `<ChemicalElement>`.

2.2 Patterns for layer 0

The programs that are part of SPARQLing genomics use a few patterns to come up with identifiers. For starters, the MD5 algorithm is used to come up with identifiers for the files that form the basis for layer 0. We use a special URI prefix for these "origins":

`<origin://d41d8cd98f00b204e9800998ecf8427e>` → `rdf:type` → `sg:Origin`

By using a hashing algorithm we can safely assume that the identifiers will be the same when the input data was the same, regardless of who, when and in what order data was processed to (re)build the knowledge graph.

This patterns is implemented by all programs described in chapter 4 'Command-line programs'.

2.2.1 Public URIs and ontologies

We have been using two types of symbols so far: internal (1) and publicly accessible (2). We use the former for data-specific identifiers (like the MD5 sum for an origin) and the latter for increasing interoperability between distinct knowledge graphs.

Such publicly accessible symbols are distributed in the form of a *vocabulary* or *ontology*. In this manual the words "vocabulary" and "ontology" are used interchangeably. The use of symbols from various ontologies is the subject of chapter 3 'Use of ontologies'.

Chapter 3

Use of ontologies

In chapter 2 'The knowledge graph' we described publicly accessible symbols as the building blocks for ontologies. In this chapter, we explain which ontologies we use, and how we use them.

The following lookup table provides the name to abbreviate an ontology, and the full ontology URI.

Abbreviation	Ontology URI
`dcterms`	http://purl.org/dc/terms/
`dc`	http://purl.org/dc/elements
`faldo`	http://biohackathon.org/resource/faldo#
`pato`	http://purl.obolibrary.org/obo/
`prov`	http://www.w3.org/ns/prov#
`sg`	https://www.sparqling-genomics.org/0.99.11/

Table 3.1: Lookup table for ontology URIs and their abbreviations.

In addition to these ontologies, we use the following internal prefixes in the remainder of the document.

Abbreviation	Ontology URI
`bam2rdf`	`sg://0.99.11/bam2rdf/`
`table2rdf`	`sg://0.99.11/table2rdf/`
`json2rdf`	`sg://0.99.11/json2rdf/`
`vcf2rdf`	`sg://0.99.11/vcf2rdf/`
`xml2rdf`	`sg://0.99.11/xml2rdf/`

Table 3.2: Internal abbreviations used in the manual.

3.1 Describing genomic positions with FALDO

When describing the position of a nucleotide relative to its reference genome, we use the Feature Annotation Location Description Ontology (FALDO) (Bolleman et al., 2016). Table 3.3 provides an overview of the properties we use.

Term	Used as	Usage
`faldo:position`	Predicate	Used by 'vcf2rdf' to describe the basepair position within a chromosome or contig.
`faldo:reference`	Predicate	Used by 'vcf2rdf' to describe the chromosome or contig to which the `faldo:position` is relative to.

Table 3.3: Terms used from FALDO.

3.2 Dublin Core Terms

In our published datasets and the 'sg-web' interface, we use the Dublin Core Terms ontology ("DCMI Metadata Terms", 2012) to define organizations, collections, datasets, and samples. Table 3.4 provides an overview of the properties we use.

Term	Used as	Description
`dcterms:Agent`	Object	Used by sg-web to describe the user or organization that produced a collection or dataset.
`dctype:Collection`	Object	Reserved for a portal page in the web interface as a filter mechanism to browse `dctype:Datasets`.
`dctype:Dataset`	Object	Reserved for a portal page in the web interface to describe data in a graph.
`dcterms:date`	Object	Used by 'sg-web' to describe the time a query was run.
`dcterms:isPartOf`	Predicate	Reserved for a portal page in the web interface to express that a `dctype:Dataset` is linked to a `dctype:Collection`.
`dcterms:title`	Predicate	Used by 'sg-web' to name a `dctype:Collection` or a `dctype:Dataset`.
`dcterms:publisher`	Predicate	Used by 'sg-web' to identify the organization that published a collection or dataset.
`dcterms:description`	Predicate	Used by 'sg-web' to provide a textual description of a `dctype:Collection` or a `dctype:Dataset`.

Table 3.4: Terms used from the Dublic Core Terms ontology.

3.3 Provenance tracking with PROV-O

To maintain an understanding of "who did what?" when it comes to importing, modifying, removing, and querying data in the knowledge graph, we use parts of the PROV ontology (Lebo, McGuinness, & Sahoo, 2013).

Term	Used as	Usage
`prov:startedAtTime`	Predicate	Used by 'sg-web' to describe the date and time a query was run.
`prov:endedAtTime`	Predicate	Used by 'sg-web' to describe the date and time a query finished.

Table 3.5: Terms used from PROV-O.

3.4 Phenotype And Trait Ontology

When describing phenotype properties, we use the Phenotype And Trait Ontology (PATO) (Mungall et al., 2020). Table 3.6 provides an overview of the properties we use.

Term	Used as	Usage
`obo:PATO_0000384`	Object	Used in the published datasets to describe a donor as male.
`obo:PATO_0000383`	Object	Used in the published datasets to describe a donor as female.
`obo:PATO_0001894`	Predicate	Used in the published datasets to describe the phenotypic sex.

Table 3.6: Terms used from PATO.

3.5 Custom terms

Sometimes we miss the right term to describe a statement. In such cases we decide on a new term that is then part of the *SPARQLing-genomics ontology*. The use of these terms is subject to change in upcoming versions of `sparqling-genomics`. Table 3.7 summarizes the terms that are waiting to be replaced by an external ontology.

Term	Used as	Usage
`sg:Origin`	Object	Used by the command-line tools (see chapter 4 'Command-line programs') to point to the file or resource from which information was derived.
`sg:Sample`	Object	Used by '`vcf2rdf`' and '`bam2rdf`' to describe a sample.

Table 3.7: Custom terms used by SPARQLing-genomics.

Chapter 4

Command-line programs

SPARQLing genomics provides programs to create an extensive knowledge graph from various domain-specific data formats. The programs described in this chapter provide the "layer 0" for the knowledge graph, and the tools to discover the data in this layer. All tools described in the remainder of this chapter can be invoked with the `--help` argument to get a complete overview of options for that particular tool.

4.1 Preparing variant call data with '`vcf2rdf`'

Variant callers extract variation from sequenced data. These programs often output the variants they found in the *Variant Call Format* (VCF) or its binary equivalent (BCF). The '`vcf2rdf`' program extracts knowledge from a VCF or BCF file and writes it as RDF.

4.1.1 Knowledge extracted by '`vcf2rdf`'

The program treats the VCF as its own ontology. It uses the VCF header as a guide. All fields described in the header of the VCF file will be translated into triples. In addition to the knowledge from the VCF file, '`vcf2rdf`' provides the following triples:

Subject	Predicate	Object	Description
`origin://`*MD5*	`rdf:type`	`sg:Origin`	This defines a uniquely identifiable reference to the originating file.
`origin://`*MD5*	`sg:filename`	*filename*	This triple states the originating filename.
`origin://`*MD5*	`sg:md5`	*MD5 sum*	This triple states the MD5 sum of the content of the original file.
`sample://`*name*	`rdf:type`	`sg:Sample`	This states that there is a sample with *sample name*.
`sample://`*name*	`sg:foundIn`	`origin://`*MD5*	This triple states that a sample can be found in a file identified by the `sg:Origin` with a specific identifier.
`origin://`*MD5*	`sg:convertedBy`	`sg:vcf2rdf-0.99.11`	This triple states that the file was converted with '`vcf2rdf`'.

Table 4.1: The additional triple patterns provided by '`vcf2rdf`'.

The following snippet is an example of the extra data in Turtle-format:

```
@prefix sg: <https://sparqling-genomics.org/0.99.11/> .
@prefix orig: <origin://> .

orig:e9e38f2e4279eda346918ba69fd86c5f
    a sg:Origin ;
    sg:convertedBy <https://sparqling-genomics.org/0.99.11/vcf2rdf-0.99.11> ;
    sg:filename "tests/sample.vcf"^^xsd:string ;
    sg:md5 "e9e38f2e4279eda346918ba69fd86c5f"^^xsd:string .

<origin://e9e38f2e4279eda346918ba69fd86c5f@S0>
    a sg:Sample ;
    rdfs:label "SAMLPEA"^^xsd:string ;
    sg:foundIn orig:e9e38f2e4279eda346918ba69fd86c5f .
```

4.1.2 Handling multi-sample VCF files

From version `0.99.11` onwards, '`vcf2rdf`' only writes variants that have at least one alternative allele specified. It will reserve the numeric ID to preserve compatibility with previous and future versions.

The numeric IDs are calculated left-to-right, top-to-bottom. So for a file containing two samples, the first variant for the first sample will receive the numeric ID 0, and the first variant for the second sample will receive numeric ID 1. The second variant for the first sample receives numeric ID 2, and so on.

4.1.3 Example usage

The following command invocation will produce RDF in the `ntriples` format:

```
vcf2rdf -i /path/to/my/variants.vcf > /path/to/my/variants.n3
```

To get a complete overview of options for this program, use:

```
vcf2rdf --help
```

4.1.4 Run-time properties

Depending on the serialization format, the program typically uses from two megabytes (in `ntriples` mode), to multiple times the size of the input VCF (in `turtle` mode).

The `ntriples` mode can output triples as soon as they are formed, while the `turtle` mode waits until all triples are known, so that it can output them efficiently, producing compact output at the cost of using more memory.

We recommend using the `ntriples` format for large input files, and `turtle` for small input files. The following example illustrates how to use `turtle` mode.

```
vcf2rdf -i /path/to/my/variants.vcf -O turtle > /path/to/my/variants.ttl
```

4.2 Preparing sequence alignment maps with '`bam2rdf`'

When DNA sequencing reads are aligned to a predetermined *reference genome*, it's often formatted in the *sequence alignment map* (SAM) format, its equivalent *binary alignment map* (BAM) format, or its equivalent

CRAM format. The ‘bam2rdf’ program can read data in all three formats.

4.2.1 Knowledge extracted by ‘bam2rdf’

The current version of ‘bam2rdf’ merely extracts information from the alignment map header. In addition to the knowledge from the file, it also produces the following metadata:

Subject	Predicate	Object	Description
origin://*MD5*	rdf:type	sg:Origin	This defines a uniquely identifiable reference to the originating file.
origin://*MD5*	sg:filename	*filename*	This triple states the originating filename.
origin://*MD5*	rdf:type	One of: bam2rdf:HeaderItem, bam2rdf:ReferenceSequence, bam2rdf:ReadGroup, bam2rdf:Program, bam2rdf:Comment.	The *objects* correspond to the various types of header lines that can occur in the SAM format.
origin://*MD5*	bam2rdf:*type*/*key*	Literal value.	Each header field consists of a key/value pair. The key is used as predicate.
origin://*MD5*	sg:convertedBy	sg:bam2rdf-0.99.11	This triple states that the file was converted with ‘bam2rdf’.

Table 4.2: The additional triple patterns provided by ‘bam2rdf’.

4.2.2 Example usage

The following command invocation will produce RDF in the ntriples format:

```
bam2rdf -i /path/to/my/sequencing_data.bam > /path/to/my/sequencing_data.n3
```

To get a complete overview of options for this program, use:

```
bam2rdf --help
```

4.3 Preparing tabular data with ‘table2rdf’

Data that can be represented as a table, like comma-separated values (CSV) or BED files, can be imported using ‘table2rdf’. The column headers are used as predicates, and each row gets a unique row ID. Non-alphanumeric characters in the header line are replaced by underscores, and all characters are replaced by their lowercase equivalent to make a consistent scheme for predicates.

When the file does not contain a header line, one can be specified using the --header-line argument. When using this command-line argument, the delimiter must be a semicolon (;).

The program can also read files compressed with ‘gzip’.

4.3.1 Transforming objects

Unfortunately, '`table2rdf`' knows nothing about ontologies. So when the input table has a column "Chromosome", by default '`table2rdf`' will treat these cells as literal values (as a `string`). A *transformer* can be used to express a column as an *individual* in RDF. An example might explain this best.

Take the following input file:

```
$ cat test.tsv
Chromosome      Position
chr1    1500000
chrMT   11000
```

Running '`table2rdf`' with its default settings will produce:

```
$ table2rdf -i test.tsv -O turtle
...
<origin://...@0>
    sg:originatedFrom <http://sparqling-genomics/...> ;
    col:chromosome "chr1"^^xsd:string ;
    col:position 1500000 ;
    a :Row .

<origin://...@1>
    sg:originatedFrom <http://sparqling-genomics/...> ;
    col:chromosome "chrMT"^^xsd:string ;
    col:position 11000 ;
    a :Row .
...
```

When we know that the data in a column refers to items in an ontology, like chromosomes defined in <http://rdf.biosemantics.org/data/genomeassemblies/hg19>, '`table2rdf`' can be told to use that ontology to describe that column.

To do so, we use the `--transform-object` option, or `-t` for short:

```
$ table2rdf -O turtle \
    -i test.tsv       \
    -t Chromosome=http://rdf.biosemantics.org/data/genomeassemblies/hg19#
...
@prefix p00000:  <http://rdf.biosemantics.org/data/genomeassemblies/hg19#> .
...
<http://sparqling-genomics/table2rdf/Row/...-R0000000000>
    sg:originatedFrom <http://sparqling-genomics/...> ;
    col:chromosome p00000:chr1 ;
    col:position 1500000 ;
    a :Row .

<http://sparqling-genomics/table2rdf/Row/...-R0000000001>
    sg:originatedFrom <http://sparqling-genomics/...> ;
    col:chromosome p00000:chrMT ;
    col:position 11000 ;
    a :Row .
```

```
...
```

After the transformation, the output produced by `table2rdf` uses URIs pointing to the ontology instead of literal values for chromosomes.

4.3.2 Transforming predicates

Like transforming a cell in a table to a URI instead of a literal value, we can also specify the value for the column name. By default, the column names are transformed using the `table2rdf:Column/` prefix (e.g. `chromosome` becomes `sg://0.99.11/table2rdf/Column/chromosome`). By using the `--transform-predicate` option, or `-T` for short, a different transformation can be made:

```
$ table2rdf -O turtle \
    -i test.tsv       \
    -t Chromosome=http://rdf.biosemantics.org/data/genomeassemblies/hg19# \
    -T Chromosome=http://biohackathon.org/resource/faldo#reference
...
@prefix p00000:  <http://rdf.biosemantics.org/data/genomeassemblies/hg19#> .
@prefix p00001:  <http://biohackathon.org/resource/faldo#> .

<origin://...@0>
    sg:originatedFrom <origin://...> ;
    p00001:reference p00000:chr1 ;
    col:position 1500000 ;
    a :Row .

<origin://...@1>
    sg:originatedFrom <origin://...> ;
    p00001:reference p00000:chrMT ;
    col:position 11000 ;
    a :Row .
...
```

4.3.3 Delimiters

Tabular data consists of rows and columns. A field is a specific place in a table, having a column-coordinate, and a row-coordinate. To distinguish fields from one another we use a delimiter. Which delimiter to use (a tab, a comma, or a semicolon, etc.) is up to the dataset. The delimiter can be chosen using the `--delimiter` option, or `-d` for short.

Sometimes a single field can consist of multiple "subfields". To distinguish subfields, we use a secondary delimiter. In RDF, we can split those subfields by using the same predicate as we would use for the entire field. Using the `--secondary-delimiter` option, we can invoke this behavior.

The following example demonstrates the usage of `--delimiter` and `--secondary-delimiter`. Take the following input file:

```
$ cat multi.tsv
Chromosome Position Filter
1 10000 A;B;C;D
```

Without using the secondary delimiter, we get:

```
$ table2rdf -i multi.tsv -O turtle
...
<origin://...@0>
    sg:originatedFrom <http://sparqling-genomics/...> ;
    col:chromosome 1 ;
    col:filter "A;B;C;D"^^xsd:string ;
    col:position 10000 ;
    a :Row .
```

Using the secondary delimiter, we get:

```
$ table2rdf -i multi.tsv --secondary-delimiter ";" -O turtle
...
<origin://...@0>
    sg:originatedFrom <http://sparqling-genomics/...> ;
    col:chromosome 1 ;
    col:filter "A"^^xsd:string, "B"^^xsd:string, "C"^^xsd:string,
               "D"^^xsd:string ;
    col:position 10000 ;
    a :Row .
```

Notice how the `col:filter` predicate now describes a connection to four objects instead of one.

4.3.4 Knowledge extracted by ‘`table2rdf`’

The ‘`table2rdf`’ program extracts all fields in the table. In addition to the knowledge from the table file, ‘`table2rdf`’ stores the following metadata:

Subject	Predicate	Object	Description
`origin://`*MD5*	`rdf:type`	`sg:Origin`	This defines a uniquely identifiable reference to the originating file.
`origin://`*MD5*	`:filename`	*filename*	This triple states the originating filename.
`origin://`*MD5*	`:convertedBy`	`sg:table2rdf-0.99.11`	This triple states that the file was converted with ‘`table2rdf`’.

Table 4.3: The additional triple patterns provided by ‘`table2rdf`’.

The following snippet is an example of the extra data in Turtle-format:

```
<origin://1jka8923i4>
    :convertedBy :table2rdf ;
    :filename "grch37.bed"^^xsd:string ;
    a :Origin .

sample:grch37
    :foundIn <origin://1jka8923i4> ;
    a :Sample .
```

4.3.5 Example usage

The following command invocation will produce RDF in the `ntriples` format:

```
table2rdf -i /path/to/my/table.tsv > /path/to/my/table.n3
```

To get a complete overview of options for this program, use:

```
table2rdf --help
```

4.4 Converting MySQL data to RDF with '`table2rdf`'

Relational databases store data in tables. With '`table2rdf`' we can oftentimes convert the data in a single go to RDF triples. The following example extracts the `regions` table from a MySQL server in a database called `example`.

```
mysql --host=127.0.0.1 -e "SELECT * FROM example.regions" \
      --batch | table2rdf --stdin > regions.n3
```

The '`mysql`' command outputs the table in tab-delimited form when using the `--batch` argument, which is the default input type for '`table2rdf`'. To accept input from a UNIX pipe '`table2rdf`' must be invoked with the `--stdin` argument.

4.5 Preparing XML data with '`xml2rdf`'

Data encoded in the EXtensible Markup Language (XML) can be converted to RDF in a naive way by '`xml2rdf`'. A child element refers to its parent with the `sg:isPartOf` predicate.

Subject	Predicate	Object	Description
`origin://`*MD5*	`rdf:type`	`sg:Origin`	This defines a uniquely identifiable reference to the originating file.
`origin://`*MD5*	`sg:filename`	*filename*	This triple states the originating filename.
`origin://`*MD5*	`sg:convertedBy`	`sg:xml2rdf-0.99.11`	This triple states that the file was converted with '`xml2rdf`'.

Table 4.4: The triplet patterns used by '`xml2rdf`'.

4.5.1 Example usage

The following command invocation will produce RDF in the `ntriples` format:

```
xml2rdf -i /path/to/my/data.xml > /path/to/my/data.n3
```

To get a complete overview of options for this program, use:

```
xml2rdf --help
```

4.6 Preparing JSON data with 'json2rdf'

The JavaScript Object Notation (JSON) has become a popular format for encoding information from and to web APIs. With 'json2rdf', a layer 0 representation in RDF can be created.

4.6.1 Knowledge extracted by 'json2rdf'

Each key-value pair is translated into `Unique ID` → `key` → `value`. When the value of a pair contains a structure with more key-value pairs, the triplet's value refers to the `unique-id` assigned to that structure. In addition to the key-value pairs, 'json2rdf' stores the following metadata:

Subject	Predicate	Object	Description
origin://*MD5*	rdf:type	sg:Origin	This defines a uniquely identifiable reference to the originating file.
origin://*MD5*	sg:filename	*filename*	This triple states the originating filename.
origin://*MD5*	sg:convertedBy	sg:json2rdf-0.99.11	This triple states that the file was converted with 'json2rdf'.

Table 4.5: The triplet patterns used by 'xml2rdf'.

4.6.2 Example usage

The following command invocation will produce RDF in the `ntriples` format:

```
json2rdf -i /path/to/my/data.json > /path/to/my/data.n3
```

To get a complete overview of options for this program, use:

```
json2rdf --help
```

4.7 Extracting knowledge from folders with `folder2rdf`

The 'folder2rdf' program finds files in a directory to extract knowledge from. It attempts to convert files with extensions `.vcf`, `.vcf.gz`, `.bcf`, and `.bcf.gz` using 'vcf2rdf', and files with extensions `.sam`, `.bam`, and `.cram` using 'bam2rdf'.

4.7.1 Example usage

```
folder2rdf --input-directory=/vcf-data   \
           --output-directory=/rdf-data  \
           --project-name Example        \
           --recursive                   \
           --compress                    \
           --threads=4
```

... where `/vcf-data` is a directory containing VCF files, and `/rdf-data` is the directory to store the converted files.

4.7.2 Knowledge extracted by 'folder2rdf'

In addition to the knowledge extracted by 'vcf2rdf', this program extracts the following data:

Subject	Predicate	Object	Description
`sg:Project/`*`project-name`*	`rdf:type`	`sg:Project`	This defines the identifier for the project.
`sg:User/`*`username`*	`rdf:type`	`sg:User`	This defines the identifier for the file owner (username).
`origin://`*`MD5`*	`rdf:type`	`sg:Origin`	This defines a uniquely identifiable reference to the originating file.

Table 4.6: The additional triple patterns produced by '`folder2rdf`'.

4.8 Importing data with '`curl`'

To load RDF data into a triple store (our database), we can use '`curl`'.

The triple stores typically store data in *graphs*. One triple store can host multiple graphs, so we must tell the triple store which graph we would like to add the data to.

4.8.1 Example usage

```
curl -X POST                                                    \
     -H "Content-Type: text/turtle"                             \
     -T /path/to/variants.ttl                                   \
     -G <endpoint URL>                                          \
     --digest -u <username>:<password>                          \
     --data-urlencode graph=http://example/graph
```

Virtuoso example

The following example inserts the file `vcf2rdf-variants.ttl` into a graph called `http://example/graph` in a Virtuoso endpoint at `http://127.0.0.1:8890` with the username `dba` and password `qwerty`.

```
curl -X POST                                                    \
     -H "Content-Type: text/turtle"                             \
     -T vcf2rdf-variants.ttl                                    \
     -G http://127.0.0.1:8890/sparql-graph-crud-auth            \
     --digest -u dba:qwerty                                     \
     --data-urlencode graph=http://example/graph
```

4store example

Similar to the Virtuoso example, for '`4store`' the command looks like this:

```
curl -X POST                                                    \
     -H "Content-Type: text/turtle"                             \
     -T vcf2rdf-variants.ttl                                    \
     -G http://127.0.0.1:8080/data/http://example/graph
```

Notice that '`4store`' does not provide an authentication mechanism.

Sending `gzip`-compressed data

When the RDF file is compressed with '`gzip`', extra HTTP headers must be added to the '`curl`' command:

```
curl -X POST                                                    \
     -H "Content-Type: text/turtle"                             \
     -H "Transfer-Encoding: chunked"                            \
     -H "Content-Encoding: gzip"                                \
     ...
```

Chapter 5

Web interface

The command-line programs described in chapter 4 'Command-line programs' provide data for layer `0`. With the web interface described in this chapter, this data can be further structured, analyzed, and shared. In short, with the web interface you can:

- Manage data and data access across RDF stores;
- Write, execute, and share SPARQL queries;
- Explore the inner-structure of datasets.

5.1 Setting up the web interface

Before the web interface can be started, a few parameters have to be configured. This is done through an XML file. The following example displays all options, except for the authentication part, which is discussed separately in section 5.1.8 'User management and authentication'.

```
<?xml version="1.0" encoding="utf-8"?>
<web-interface>
  <fork>0</fork>
  <bind-address>127.0.0.1</bind-address>
  <port>8080</port>
  <developer-mode>0</developer-mode>
  <backtrace-on-error>0</backstrace-on-error>
  <upload-root>/data</upload-root>
  <system-connection>
    <uri>http://localhost:8890/sparql-auth</uri>
    <backend>virtuoso</backend>
    <username>dba</username>
    <password>dba</password>
  </system-connection>
  <authentication>
    <!-- Either LDAP settings, or local-user authentication -->
  </authentication>
</web-interface>
```

5.1.1 To fork or not to fork

The `fork` property can be either `0` to keep the '`sg-web`' process in the foreground of your shell, or `1` to run the '`sg-web`' process as a daemon.

5.1.2 Bind address and port

To enable running multiple web services on single machine, ‘sg-web’ can be configured to bind on an arbitrary address and an arbitrary port. Both IPv4 and IPv6 addresses are supported.

5.1.3 Developer mode

By default, changes to code in the ‘www/pages/’ directory are applied when restarting ‘sg-web’. When turning on ‘developer-mode’, pages are reloaded when accessed, which makes interactive development of web pages easier.

5.1.4 Logging and backtraces

The ‘sg-web’ program keeps two logs: one for error reporting, and one for access and non-critical messages. The log files can be specified as command-line options (-e for error reporting, -d for non-critical messages).

Backtraces help the developer to get an idea about how something might’ve gone wrong. It displays the location nearby the location in the source code that throwed an error. Backstraces can get long and verbose. Therefore we don’t display them by default. To enable reporting backtraces in the error log, set the ‘backtrace-on-error’ option to 1.

5.1.5 Upload directory

Users can upload RDF files to import it into an RDF store. The files must be stored on the filesystem. With upload-root, the location of the upload directory can be changed. It can also be set using the environment variable SG_WEB_UPLOAD_ROOT. If none of these are specified, the value of the environment variable TMPDIR will be used, or when that is not set, ‘/tmp’ will be used.

5.1.6 System connection

The web interface stores its own information as RDF. Therefore it needs a connection to an RDF store where it can write to the graphs described in table 5.1.

Graph	Reason
http://sparqling-genomics.org/sg-web/state	In this graph, queries and projects are stored.
http://sparqling-genomics.org/sg-web/cache	This graph is used to speed up the web interface by pre-running various SPARQL queries.

Table 5.1: Graphs that need to be writable for the web interface.

System connection example

To configure the *system connection*, two parameters need to be specified: uri, and backend. Additionally, when the RDF store requires authentication for writing to it, a username and a password can be provided.

The following example shows how to configure the *system connection*:

```
<?xml version="1.0" encoding="utf-8"?>
<web-interface>
  ...
  <system-connection>
    <uri>http://localhost:8890/sparql-auth</uri>
    <backend>virtuoso</backend>
    <username>dba</username>
```

```
    <password>dba</password>
  </system-connection>
</web-interface>
```

5.1.7 Beacon support

Beacon is an interface to create a "global search engine for genetic mutations" ("Beacon Network", 2018). It achieves this by defining a standard that institutions must implement so that one search engine can query the implementations from each institution to find a specific genetic mutation.

The web interface can function as a Beacon in the Beacon network ("Beacon Network", 2018). The Beacon API uses a separate connection, similar to the `system-connection`, so that access can be controlled at the user level.

The following example shows how to configure *Beacon* including the *Beacon connection*:

```
<?xml version="1.0" encoding="utf-8"?>
<web-interface>
  ...
  <beacon>
    <enabled>1</enabled>
    <organization>
      <id>SG</id>
      <name>SPARQLing-genomics Beacon service</name>
      <description>
        This Beacon service provides variant information for data hosted by
        this instance of the RDF store.
      </description>
      <address>Not provided</address>
      <welcome-url>https://www.sparqling-genomics.org</welcome-url>
      <contact-url>mailto:beacon@sparqling-genomics.org</contact-url>
      <logo-url>https://www.sparqling-genomics.org/static/images/logo.png</logo-url>
      <info>Not provided</info>
    </organization>
    <connection>
      <uri>http://localhost:8000</uri>
      <backend>virtuoso</backend>
      <username>beacon</username>
      <password>changeme</password>
    </connection>
  </beacon>
</web-interface>
```

The implementation assumes the following conditions are met:

- Variant call data was imported using `vcf2rdf`;
- The reference genome can be identified by the chromosome's NCBI identifier.

5.1.8 User management and authentication

The '`sg-web`' assumes every user of the system has a username and a password to authenticate oneself. There are two ways to configure authentication. For isolated deployments or environments, preconfigured accounts can be specified. For organizational deployments, the web interface can be configured to use an LDAP server that supports version 3 of the LDAP protocol.

Regardless of the authentication mechanism, there are a few reserved users that carry out a specific task within `sg-web`. The table below describes the reserved users.

Username	Task
`beacon`	When the BEACON API (see section 5.1.7 'Beacon support') is enabled, and this user is added to a project, the data that is accessible by the project will be exposed to the BEACON network.
`portal`	This username has been reserved for an implementation of a data portal.

Table 5.2: Reserved users for `sg-web`.

Local users configuration

The simplest form of authentication is the "local-user configuration". Configuring it involves providing a username and the SHA256 sum of a password for each account. The following example shows how to configure "local-user authentication":

```
<?xml version="1.0" encoding="utf-8"?>
<web-interface>
  ...
  <authentication>
    <user>
      <username>user</username>
      <!-- The password field must contain the SHA256 sum of the
           plaintext password -->
      <password>9f86d08...0f00a08</password>
    </user>
    <user>
      <username>user2</username>
      <password>152f347...0b7a26a</password>
    </user>
  </authentication>
</web-interface>
```

LDAP authentication example

To configure LDAP, four parameters must be specified: the URI to the LDAP service (1), optionally, an extra "common name" (2), optionally the "organizational unit" (3), and the "domain" (4). The username is used as a "common name".

Additionally, an alternative SSL certificate bundle can be configured with the parameters `ssl-certificate-directory` and `ssl-certificate-file`.

The following example shows how to configure LDAP authentication:

```
<?xml version="1.0" encoding="utf-8"?>
<web-interface>
  ...
  <authentication>
    <ldap>
      <uri>ldaps://example.local</uri>
      <common-name>AdditionalCN</common-name>
```

```
      <organizational-unit>People</organizational-unit>
      <domain>department.organization.tld</domain>
      <ssl-certificate-directory>/etc/ssl/certs</ssl-certificate-directory>
      <ssl-ca-certificate-file>
        /etc/ssl/certs/ca-certificates.crt
      </ssl-ca-certificate-file>
    </ldap>
  </authentication>
</web-interface>
```

5.1.9 Running the web interface

The web interface can be started using the `sg-web` command:

```
sg-web --configuration-file=file.xml
```

... where `file.xml` is a configuration file as discussed in section 5.1 'Setting up the web interface'.

5.2 Managing multi-node setups with '`sg-auth-manager`'

When the knowledge graph grows beyond the capabilities of a single machine, one can consider scaling out to another. The provided solution to manage multiple RDF stores while maintaining a single-point-of-entry for users involves running helper program called '`sg-auth-manager`' alongside a secondary RDF store. This program communicates with '`sg-web`' to manage permissions, and handle data import and export. Figure 5.1 provides a schematic overview of the deployment model.

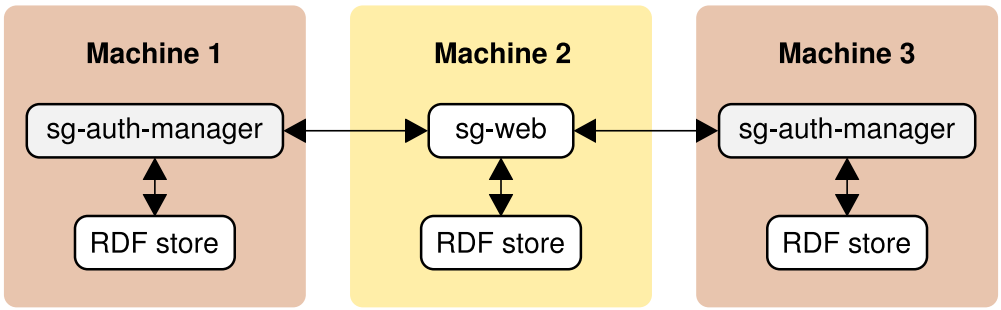

Figure 5.1: *Illustrating a three-node setup.*

The '`sg-auth-manager`' program verifies the validity of the user's session token with the '`sg-web`' instance before acting as a proxy to the managed RDF store.

5.2.1 Importing data into secondary stores

RDF importing is handled by an API call to the instance managed by '`sg-auth-manager`'.

When '`sg-web`' receives a request to import data into an '`sg-auth-manager`'-managed RDF store, it passes the data directly to the '`sg-auth-manager`', which in turn handles the importing of data into that store.

5.2.2 How `sg-web` handles downtime of a '`sg-auth-manager`'

The '`sg-auth-manager`' registers itself with the pre-configured '`sg-web`' instance upon start. In turn, the '`sg-web`' instance polls the availability of the '`sg-auth-manager`' instance every 10 seconds. When the '`sg-auth-manager`' instance is unavailable for 30 consecutive seconds, '`sg-web`' removes the connection, and the '`sg-auth-manager`' instance must re-register itself.

5.2.3 Federated querying of protected endpoints

The `sg-auth-manager` provides a mechanism to execute federated queries using protected endpoints by allowing the session token (see section 5.4.4) as part of the URI in the `SERVICE` specification.

5.3 Using the web interface

Once the web interface is up and running, a logged-in user will land on the *Dashboard*.

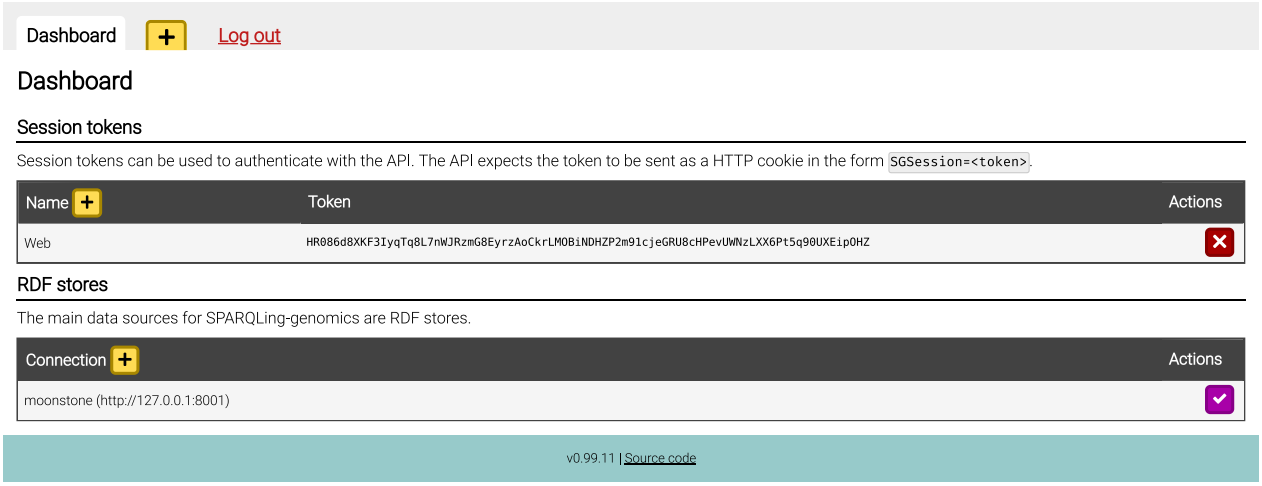

Figure 5.2: On the *Dashboard*, tokens and connections can be configured.

The dashboard provides access to two resources: *session tokens* and *RDF stores*. Session tokens will be discussed in section 5.4.4 'Token-based authentication'. This leaves us with *RDF stores* for now.

5.3.1 Configuring connections

The web interface supports two types of connections; *user-specific connections* and *system-wide connections*. Users can add the former using the + button. Such a connection will only be visible to the user who added it. On the contrary, *system-wide connections* are visible to all users.

System-wide connections are therefore not configured by a user, but by a program called '`sg-auth-manager`'. This program works as a middle-man between a private RDF store, and a running instance of '`sg-web`'. It uses the user management from '`sg-web`' described in section 5.1.8 'User management and authentication' to allow users to access graphs assigned to one of their projects.

An instance of '`sg-auth-manager`' registers itself with '`sg-web`' as a system-wide connection: making itself available to all users of the '`sg-web`' instance.

5.3.2 Projects

The main way to access the knowledge graph, and to collaborate with other users is through a *project*. Projects combine *users*, *access to data* and *queries*. The structure of a project attempts to capture general phases of analyzing data: *collect* → *structure* → *query* → *report* → *automate*.

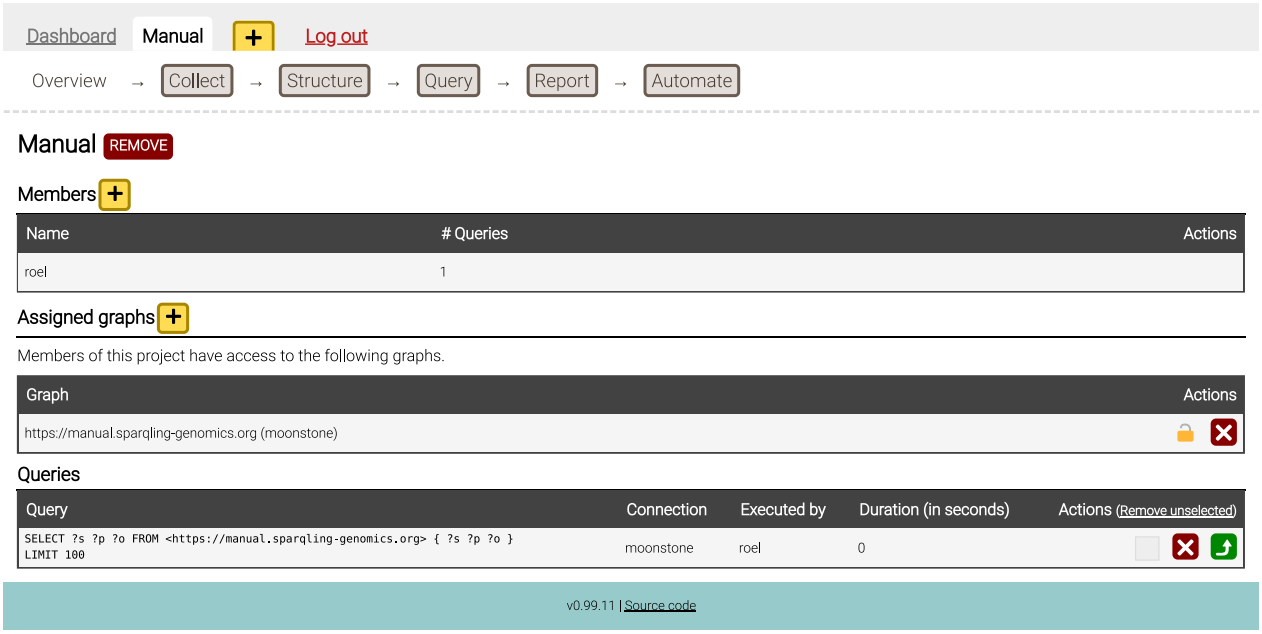

Figure 5.3: The project overview page.

Collecting data

Before data can be analyzed, it must be collected and stored. Graphs are the primary place to store data. Graphs are identified by a uniform resource identifier (URI). So before data can be imported, a project must assign a graph.

After assigning a graph to the project on the *Overview* page, the command to load a file can be generated in three steps.

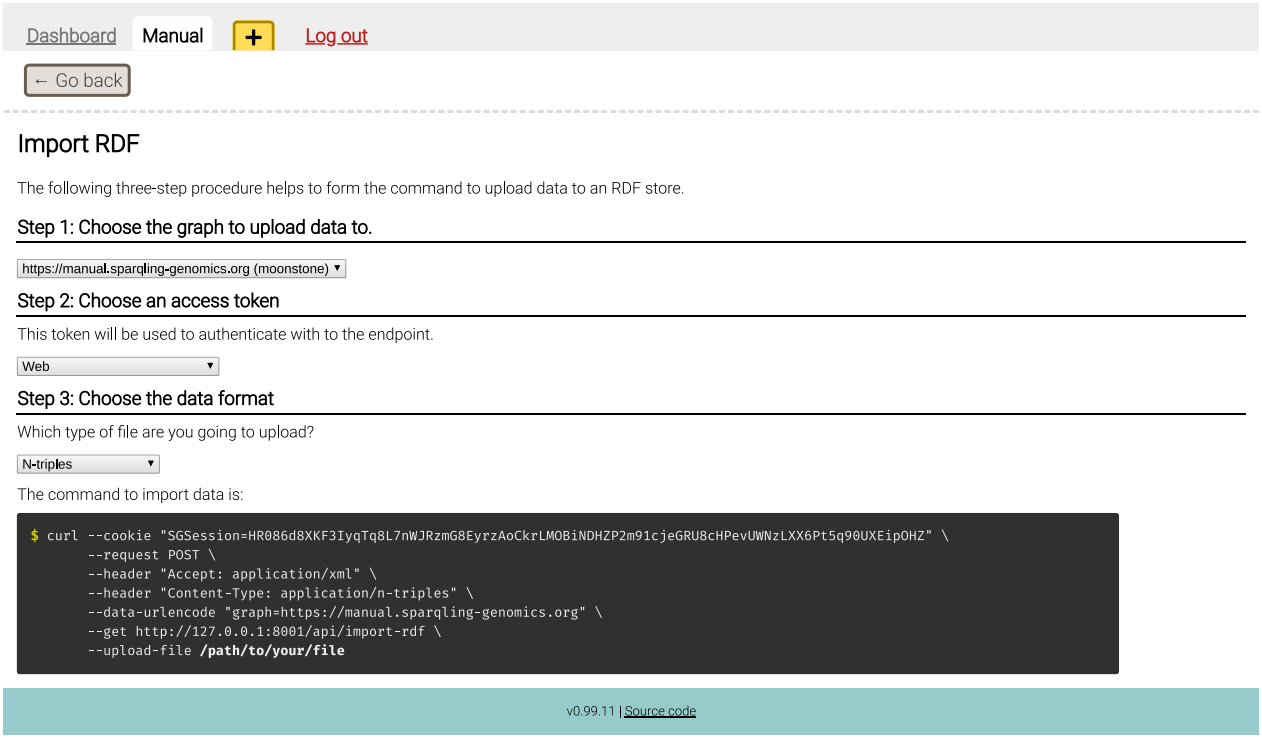

Figure 5.4: Importing RDF in three steps

5.3.3 Executing queries

After configuring at least one endpoint, it can be chosen on the *query* page to execute a query against it.

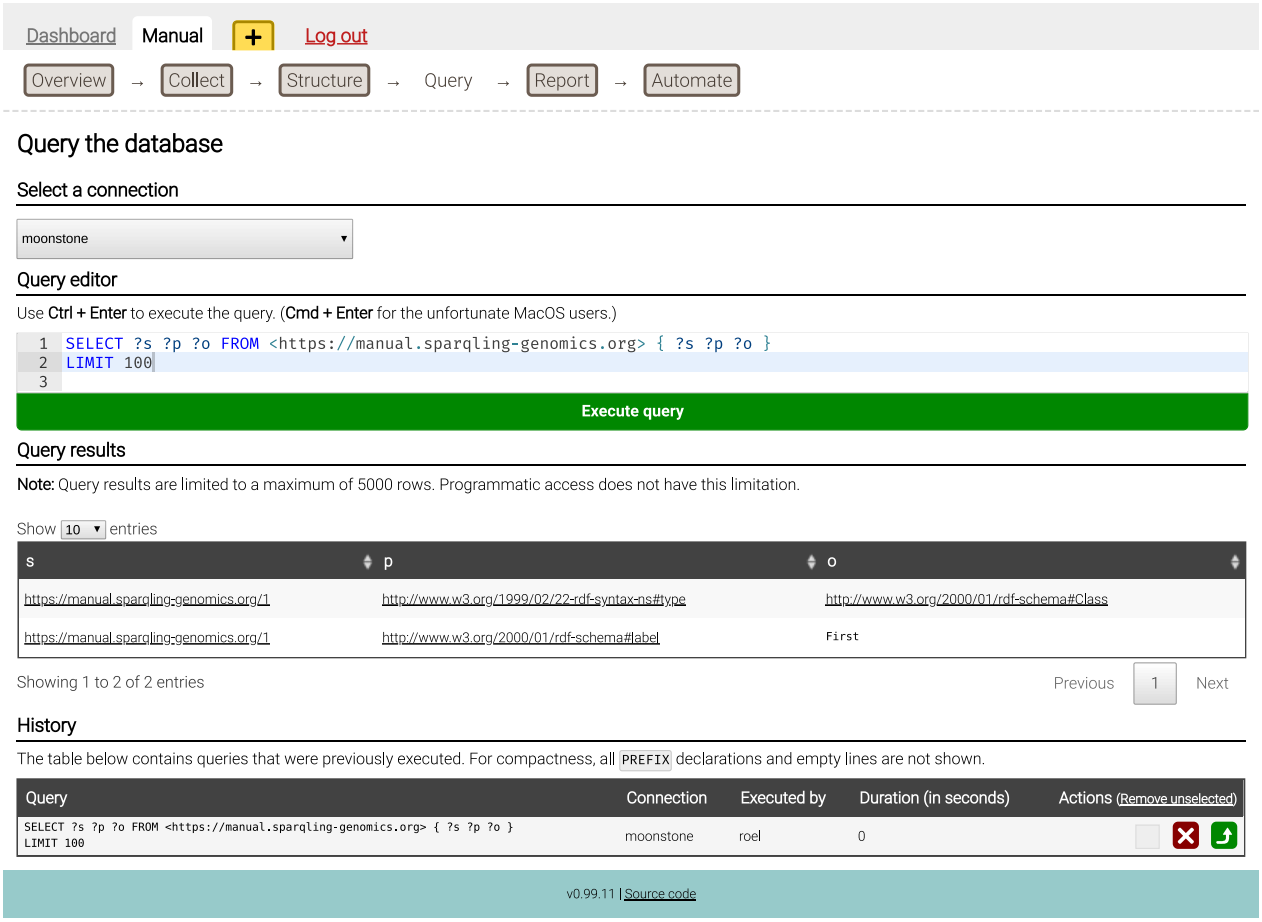

Figure 5.5: The *query* page enables users to execute a query against a SPARQL endpoint. The connections configured at the *connections* page can be chosen from the drop-down menu.

5.3.4 Query history

When prototyping SPARQL queries, better known as “SPARQLing around”, it’s good to know that all queries that yielded a result are stored in the *query history*. The history is shown on the *query* page below the query editor. Each *project* has its own query history.

5.3.5 Explore graphs with the Exploratory

Another utility aimed at SPARQLing around faster is the *exploratory*.

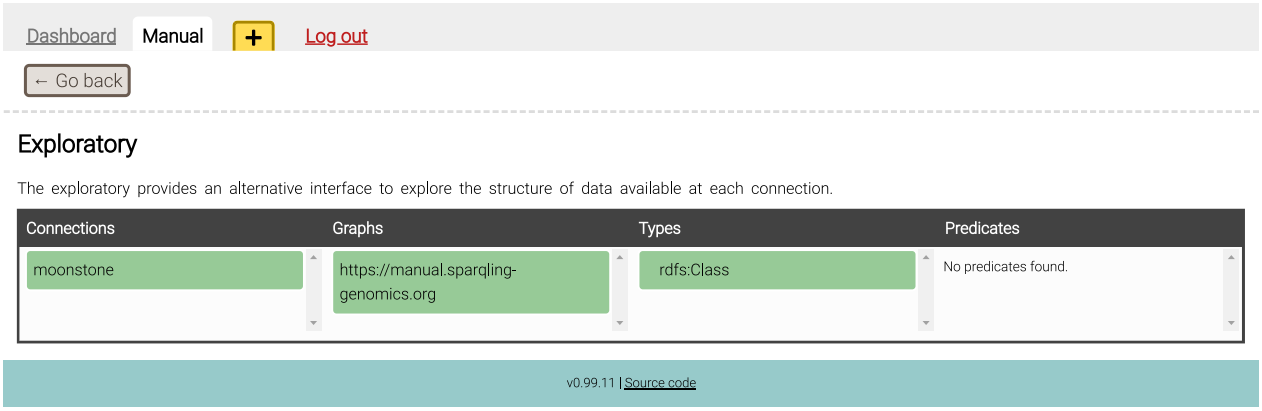

Figure 5.6: The *exploratory* page enables users to learn about the structure of the triplets in a graph.

The exploratory uses a common pattern in RDF to help writing queries. Its interface provides a four-step selection process to find *predicates* associated with an `rdf:type`. The programs described in chapter 4 'Command-line programs' automatically add the `rdf:type` annotations.

5.3.6 Connections and graphs

The first step in finding predicates involves choosing a connection (see section 5.3.1 'Configuring connections'). The second step involves choosing a graph. If the connection does not support the use of graphs, the journey ends here.

5.3.7 Types

The third step looks for triplets that match the pattern *subject* → `rdf:type` → *type*. All matches for *type* are displayed. For data imported with `vcf2rdf` (see section 4.1 'Preparing variant call data with '`vcf2rdf`''), this will display (among other types) the `VariantCall` type.

5.3.8 Predicates

Staying with the `VariantCall` example; All data properties extracted from a VCF file can be found under this type. A predicate displayed in this column occurs in *at least* one triplet. It not necessarily occurs in *every* triplet. Especially when using `INFO` and `FORMAT` fields in a VCF file, we recommend using them in a query inside an `OPTIONAL` clause.

5.4 Programming interface

Other than a user interface, the web interface provides a programming interface using the HyperText Transport Protocol (HTTP). The interface supports XML, JSON, and S-expressions. Table 5.3 summarizes the supported formats.

Content-Type	Example response
`application/json`	`[{ "message": "This is a JSON response." }]`
`application/xml`	`<message>This is a XML response.</message>`
`application/s-expression`	`(message . "This is a S-expression response.")`

Table 5.3: Implemented content types for the API. The `Content-Type` can be used in the `Accept` HTTP header.

5.4.1 Formatting `POST` requests

The `Accept` parameter influences the response format of the API, and the `Content-Type` parameter can be used to indicate the format of the request.

In addition to the documented `Content-Type` values, the type `application/x-www-form-urlencoded` is also allowed, which expects the following format:

```
parameter1=value1&parameter2=value2&...
```

The `Content-Type` header line does not have to be equal to the `Accept` header line, so for example, parameters can be sent in XML and the response can be formatted in JSON or the other way around.

5.4.2 Conventions when using XML

When using XML, there are a few conventions to follow. To communicate a list or array of records, the API relies on using pre-defined elements. The API expects parameters to be wrapped in `<parameters>...</parameters>` elements. So, to log in using an XML request, the following message will be accepted:

```
POST /api/login HTTP/1.1
Host: ...
Content-Type: application/xml
Content-Length: 80
Connection: close

<parameters>
  <username>...</username>
  <password>...</password>
</parameters>
```

Subsequently, when `Accepting` XML the results are wrapped in `<results>...</results>`, and data structures built from multiple key-value pairs are wrapped in `<result>...<result>`. The following example illustrates receiving a project record:

```
GET /api/projects HTTP/1.1
Host: ...
```

```
Accept: application/xml
Cookie: ...
Connection: close

<results>
  <result>
    <projectId>http://sparqling-genomics.org/0.99.10/Project/...</projectId>
    <creator>http://sparqling-genomics.org/0.99.10/Agent/...</creator>
    ...
  </result>
</results>
```

Note: The actual response strips the whitespace. It was added to this example for improved readability.

5.4.3 Authenticating API requests with `/api/login`

Before being able to interact with the API, a session token must be obtained. This can be done by sending a `POST`-request to `/api/login`, with the following parameters:

Parameter	Example	Required?
`username`	`jdoe`	Yes
`password`	`secret`	Yes

The following cURL command would log the user `jdoe` in:

```
curl --cookie-jar - http://localhost/api/login \
     --data "username=jdoe&password=secret"
```

The response contains a cookie with the session token. Use this cookie in subsequent requests. When authentication fails, the service will respond with HTTP status-code `401`.

5.4.4 Token-based authentication

When automating data importing and running queries there comes the point at which you have a choice: Do I put in my login credentials in a script, a separate file, or anywhere else on the filesystem? This bad security practice can be overcome by using *tokens*.

A token is a randomly generated string that can be used to authenticate with, instead of your username and password. Tokens are easy to create, and more importantly, easy to revoke.

Generating a token can be done on the *Dashboard*, or using the API call `/api/new-session-token`. Removing a token can be done on the *Dashboard*, or using the API call `/api/delete-token`.

After generating a token, it can be used as a cookie in API calls. The name of the cookie is always `SGSession`, regardless of the name of the token.

5.4.5 Managing connections

The API can be used to store connections and refer to them by name. The remainder of this section describes the API calls related to connections.

Retrieve connections with `/api/connections`

With a call to `/api/connections`, the pre-configured connections can be viewed. This resource expects a `GET` request and needs no parameters. It returns all connection records associated with the currently logged-in user.

The following cURL command would retrieve all connection records in JSON format:

```
curl http://localhost/api/connections \
     --cookie "$COOKIE"               \
     -H "Accept: application/json"
```

Create a new connection with `/api/add-connection`

A call to `/api/add-connection` will create a new connection. The table below summarizes the parameters that can be used in this call.

Parameter	Example	Required?
`name`	`Example`	Yes
`uri`	`http://my/endpoint`	Yes
`username`	`jdoe`	No
`password`	`secret`	No
`backend`	`4store`	No

The following cURL command would create a connection, using JSON as the format to express the parameters:

```
curl http://localhost/api/add-connection          \
     --cookie "$COOKIE"                           \
     -H "Accept: application/xml"                 \
     -H "Content-Type: application/json"          \
     --data '{ "name":     "Example",             \
               "uri":      "http://my/endpoint", \
               "username": "jdoe",                \
               "password": "secret",              \
               "backend":  "4store" }'
```

Remove a connection with `/api/remove-connection`

A call to `/api/remove-connection` will remove an existing connection. The table below summarizes the parameters that can be used in this call.

Parameter	Example	Required?
`name`	`Example`	Yes

The following cURL command would remove the connection that was created in section 5.4.5 'Create a new connection with `/api/add-connection`':

```
curl http://localhost/api/remove-connection       \
     --cookie "$COOKIE"                           \
     -H "Accept: application/xml"                 \
```

```
     --data "name=Example"
```

5.4.6 Managing projects

The API has various resources to manage projects, as described in section 5.3.2 'Projects'.

Retrieve a list of projects with `/api/projects`

Retrieving a list of projects can be done by sending a `GET` request to `/api/projects`.

The following cURL command would retrieve a list of projects:

```
curl http://localhost/api/projects      \
     --cookie "$COOKIE"                 \
     -H "Accept: application/xml"
```

Create a new project with `/api/add-project`

To create a new project, send a `POST` request to `/api/add-project`. The table below summarizes the parameters that can be used with this call.

Parameter	Example	Required?
`name`	`Example project`	Yes

The following cURL command would add a project:

```
curl http://localhost/api/add-project      \
     --cookie "$COOKIE"                    \
     -H "Accept: application/xml"          \
     --data "name=Example project"
```

Remove a project with `/api/remove-project`

To remove a project, send a `POST` request to `/api/remove-project`. The table below summarizes the parameters that can be used with this call.

Parameter	Example	Required?
`project-uri`	`http://sparqling-genomics.org/Project/640c0...5a6d2`	Yes

Alternatively, instead of the full URI, the project's hash can be used. In that case, the parameters are:

Parameter	Example	Required?
`project-hash`	`640c0...5a6d2`	Yes

The following cURL command would remove the project created in 5.4.6 'Create a new project with `/api/add-project`':

```
curl http://localhost/api/remove-project      \
     --cookie "$COOKIE"                       \
```

```
     -H "Accept: application/xml"             \
     --data "project-uri=http://sparqling-genomics.org/Project/640c0...5a6d2"
```

Assign a graph to a project `/api/assign-graph`

Assigning a graph to a project can be done by sending a `POST` request to `/api/assign-graph`. The required parameters are:

- `project-uri`: This can be obtained from a call to `/api/projects`.
- `connection`: The name of a connection obtained from a call to `/api/connections`.
- `graph-uri`: The graph URI to assign.

The following example based on cURL shows how to perform the request:

```
curl -X POST                                          \
     -H "Accept: application/json"                    \
     -H "Content-Type: application/json"              \
     --cookie "SGSession=..."                         \
     --data '{ "project-uri": "http://the-project-uri",    \
               "connection": "wikidata",              \
               "graph-uri": "http://the-new-graph-name" }' \
     http://localhost/api/assign-graph
```

Unassign a graph from a project with `/api/unassign-graph`

Like assigning a graph to a project, it can be unassigned as well using `/api/unassign-graph` using the same parameters as `/api/assign-graph`.

The following example based on cURL shows how to perform the request:

```
curl -X POST                                          \
     -H "Accept: application/json"                    \
     -H "Content-Type: application/json"              \
     --cookie "SGSession=..."                         \
     --data '{ "project-uri": "http://the-project-uri",    \
               "graph-uri": "http://the-new-graph-name" }' \
     http://localhost/api/unassign-graph
```

5.4.7 Running and viewing queries

Retrieve previously run queries with `/api/queries`

When running queries through SPARQLing-genomics's interface, each succesful query is stored in a "query history". This query history can be retrieved using a call to `/api/queries`. This resource expects a `GET` request.

The following cURL command would retrieve the entire query history, formatted as XML:

```
curl http://localhost/api/queries \
     --cookie "$COOKIE"           \
     -H "Accept: application/xml"
```

Toggle query marks

Previously executed queries can be marked as *important*, which means they will survive a call to `/api/queries-remove-unmarked`.

The following example based on cURL shows how to mark a query as *important*:

```
curl -X POST                                                \
     -H "Accept: application/json"                          \
     -H "Content-Type: application/json"                    \
     --cookie "SGSession=..."                               \
     --data '{ "query-id": "...", "state": true }'
     http://localhost/api/query-mark
```

Remove unmarked queries

Removing unmarked queries for a project can be done by sending a `POST` request to `/api/queries-remove-unmarked`. The required parameters are:

- `project-uri`: This can be obtained from a call to `/api/projects`.

The following example based on cURL shows how to perform the request:

```
curl -X POST                                                \
     -H "Accept: application/json"                          \
     -H "Content-Type: application/json"                    \
     --cookie "SGSession=..."                               \
     --data '{ "project-uri": "http://the-project-uri" }'
     http://localhost/api/queries-remove-unmarked
```

Chapter 6

Information retrieval with SPARQL

In section 4.1 'Preparing variant call data with `vcf2rdf`' we discussed how to extract triples from common data formats. In section 4.8 'Importing data with `curl`' we discussed how we could insert those triples into a SPARQL endpoint.

In this section, we will start exploring the inserted data by using a query language called *SPARQL*. Understanding SPARQL will be crucial for the integration in our own programs or scripts — something we will discuss in chapter 8 'Using SPARQL with other programming languages'.

The queries in the remainder of this chapter can be readily copy/pasted into the query editor of the web interface (see chapter 5 'Web interface').

6.1 Local querying

When we request information from a SPARQL endpoint, we are performing a *local query* because we request data from a single place. In our case, that is most likely to be our own SPARQL endpoint.

In contrast to *local querying*, we can also query multiple SPARQL endpoints in one go, to combine the information from multiple locations. Combining information from multiple SPARQL endpoints is called *federated querying*.

Federated querying is discussed in section 6.2 'Federated querying'.

6.1.1 Listing non-empty graphs

Each SPARQL endpoint can host multiple *graphs*. Each graph can contain an independent set of triples. The following query displays the available non-empty graphs in a SPARQL endpoint:

```
SELECT DISTINCT ?graph WHERE { GRAPH ?graph { ?s ?p ?o } }
```

Which may yield the following table:

graph
http://example
http://localuriqaserver/sparql
http://www.openlinksw.com/schemas/virtrdf#
http://www.w3.org/2002/07/owl#
http://www.w3.org/ns/ldp#

Table 6.1: Results of the query to list non-empty graphs.

The graph names usually look like URLs, like we would encounter them on the web. In fact, not only graph names, but any node that has a symbolic meaning, rather than a literal[1] meaning is usually written as a URL. We can go to such a URL with a web browser and might even find more information.

6.1.2 Querying a specific graph

The sooner we can reduce the dataset to query over, the faster the query will return with an answer. One easy way to reduce the size of the dataset is to be specific about which graph to query. This can be achieved using the `FROM` clause in the query.

```
SELECT ?s ?p ?o
FROM <graph-name>
WHERE { ?s ?p ?o }
```

The `graph-name` must be one of the graphs returned by the query from section 6.1.1 'Listing non-empty graphs'.

Without the `FROM` clause, the RDF store will search in all graphs. We can repeat the `FROM` clause to query over multiple graphs in the same RDF store.

```
SELECT ?s ?p ?o
FROM <graph-name>
FROM <another-graph-name>
WHERE { ?s ?p ?o }
```

In section 6.2 'Federated querying' we will look at querying over multiple RDF stores.

6.1.3 Exploring the structure of knowledge in a graph

Inside the `WHERE` clause of a SPARQL query we specify a graph pattern. When the information in a graph is structured, there are only few predicates in comparison to the number of subjects and the number of objects.

```
SELECT COUNT(DISTINCT ?s) AS ?subjects
       COUNT(DISTINCT ?p) AS ?predicates
       COUNT(DISTINCT ?o) AS ?objects
FROM <http://example>
WHERE { ?s ?p ?o }
```

On a typical graph with data originating from `vcf2rdf`, this may yield the following table:

[1] Examples of literals are numbers and strings. Symbols are nodes that don't have a literal value.

subjects	predicates	objects
3011691	229	4000809

Table 6.2: Results of the query to count the number of subjects, predicates, and objects in a graph.

Therefore, one useful method of finding out which patterns exist in a graph is to look for predicates:

```
SELECT DISTINCT ?predicate
FROM <http://example>
WHERE {
  ?subject ?predicate ?object .
}
```

Which may yield the following table:

predicate
http://biohackathon.org/resource/faldo#position
http://biohackathon.org/resource/faldo#reference
http://sparqling-genomics/vcf2rdf/filename
http://sparqling-genomics/vcf2rdf/foundIn
http://sparqling-genomics/vcf2rdf/sample
http://sparqling-genomics/vcf2rdf/VariantCall/ALT
http://sparqling-genomics/vcf2rdf/VariantCall/FILTER
...

Table 6.3: Results of the query to list predicates.

6.1.4 Listing samples and their originating files

Using the knowledge we gained from exploring the predicates in a graph, we can construct more insightful queries, like finding the names of the samples and their originating filenames from the output of `vcf2rdf`:

```
PREFIX vcf2rdf: <http://sparqling-genomics/vcf2rdf/>

SELECT DISTINCT STRAFTER(STR(?sample), "Sample/") AS ?sample ?filename
FROM <graph-name>
WHERE {
  ?variant  vcf2rdf:sample    ?sample .
  ?sample   vcf2rdf:foundIn   ?origin .
  ?origin   vcf2rdf:filename  ?filename .
}
```

Which may yield the following table:

sample	filename
REF0047	/data/examples/TUMOR_REF0047.annotated.vcf.gz
TUMOR0047	/data/examples/TUMOR_REF0047.annotated.vcf.gz
...	...

Table 6.4: Results of the query to list samples and their originating filenames.

Notice how most predicates for `vcf2rdf` in table 6.3 start with `http://sparqling-genomics/vcf2rdf/`. In the above query, we used this to shorten the query. We started the query by writing a `PREFIX` rule for `http://sparqling-genomics/vcf2rdf/`, which we called `vcf2rdf:`. This means that whenever we write `vcf2rdf:FOO`, the SPARQL endpoint interprets it as if we would write `<http://sparqling-genomics/vcf2rdf/FOO>`.

We will use more prefixes in the upcoming queries. We can look up prefixes for common ontologies using http://prefix.cc.

6.1.5 Listing samples, originated files, and number of variants

Building on the previous query, and by exploring the predicates of a `vcf2rdf:VariantCall`, we can construct the following query to include the number of variants for each sample, in each file.

```
PREFIX vcf2rdf: <http://sparqling-genomics/vcf2rdf/>
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT STRAFTER(STR(?sample), "Sample/") AS ?sample
       ?filename
       COUNT(DISTINCT ?variant) AS ?numberOfVariants

FROM <graph-name>
WHERE
{
  ?variant  rdf:type                vcf2rdf:VariantCall ;
            vcf2rdf:sample          ?sample ;
            vcf2rdf:originatedFrom  ?origin .

  ?origin   vcf2rdf:filename        ?filename .
}
```

Which may yield the following table:

sample	filename	numberOfVariants
REF0047	/data/examples/TUMOR_REF0047.annotated.vcf.gz	1505712
TUMOR0047	/data/examples/TUMOR_REF0047.annotated.vcf.gz	1505712
...	...	...

Table 6.5: Results of the query to list samples, their originated filenames, and the number of variant calls for each sample in a file.

By using `COUNT`, we can get the `DISTINCT` number of matching patterns for a variant call for a sample, originating from a distinct file.

6.1.6 Retrieving all variants

When retrieving potentially large amounts of data, the `LIMIT` clause may come in handy to prototype a query until we are sure enough that the query answers the actual question we would like to answer.

In the next example query, we will retrieve the sample name, chromosome, position, and the corresponding VCF `FILTER` field(s) from the database.

```
PREFIX vcf2rdf: <http://sparqling-genomics/vcf2rdf/>
PREFIX vc:      <http://sparqling-genomics/vcf2rdf/VariantCall/>
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX faldo:   <http://biohackathon.org/resource/faldo#>

SELECT DISTINCT ?variant ?sample ?chromosome ?position ?filter
FROM <graph-name>
WHERE
{
  ?variant  rdf:type               vcf2rdf:VariantCall ;
            vcf2rdf:sample         ?sample ;
            faldo:reference        ?chromosome ;
            faldo:position         ?position ;
            vc:FILTER              ?filter .
}
LIMIT 100
```

By limiting the result set to the first 100 rows, the query will return with an answer rather quickly. Had we not set a limit to the number of rows, the query could have returned possibly a few million rows, which would obviously take longer to process.

6.1.7 Retrieving variants with a specific mutation

Any property can be used to subset the results. For example, we can look for occurrences of a `C` to `T` mutation in the positional range 202950000 to 202960000 on chromosome 2, according to the *GRCh37 (hg19)* reference genome with the following query:

```
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
PREFIX faldo: <http://biohackathon.org/resource/faldo#>
PREFIX hg19:  <http://rdf.biosemantics.org/data/genomeassemblies/hg19#>
PREFIX v:     <http://sparqling-genomics/vcf2rdf/>
PREFIX vc:    <http://sparqling-genomics/vcf2rdf/VariantCall/>
PREFIX seq:   <http://sparqling-genomics/vcf2rdf/Sequence/>

SELECT COUNT(DISTINCT ?variant) AS ?occurrences ?sample
FROM <http://example>
WHERE {
  ?variant  rdf:type         v:VariantCall .
  ?variant  rdf:type         ?genotype .
  ?variant  v:sample         ?sample .
  ?variant  vc:REF           seq:C .
  ?variant  vc:ALT           seq:T .
  ?variant  faldo:reference  hg19:chr2 .
  ?variant  faldo:position   ?position .
```

```
  FILTER (?position >= 202950000)
  FILTER (?position <= 202960000)

  # Exclude variants that actually do not deviate from hg19.
  FILTER (?genotype != v:HomozygousReferenceGenotype)
}
LIMIT 2
```

Which may yield the following table:

occurrences	sample
5	http://sparqling-genomics/vcf2rdf/Sample/REF0047
5	http://sparqling-genomics/vcf2rdf/Sample/TUMOR0047

Table 6.6: Query results of the above query.

6.1.8 Comparing two datasets on specific properties

Suppose we run variant calling on the same sample with slightly different analysis programs. We expect a large overlap in variants between the datasets, and would like to view the few variants that differ in each dataset.

We imported each dataset in a separate graph (`http://comparison/aaa` and `http://comparison/bbb`).

The properties we are going to compare are the predicates `faldo:reference`, `faldo:position`, `vc:REF`, and `vc:ALT`.

The query below displays how each variant in `http://comparison/aaa` can be compared to a matching variant in `http://comparison/bbb`. Only those variants in `http://comparison/aaa` that **do not** have an equivalent variant in `http://comparison/bbb` will be returned by the SPARQL endpoint.

```
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX faldo:   <http://biohackathon.org/resource/faldo#>
PREFIX vcf2rdf: <http://sparqling-genomics/vcf2rdf/>
PREFIX vc:      <http://sparqling-genomics/vcf2rdf/VariantCall/>

SELECT DISTINCT
  STRAFTER(STR(?chromosome), "hg19#") AS ?chromosome
  ?position
  STRAFTER(STR(?reference), "Sequence/") AS ?reference
  STRAFTER(STR(?alternative), "Sequence/") AS ?alternative
  STRAFTER(STR(?filter), "vcf2rdf/") AS ?filter
WHERE
{
  GRAPH <http://comparison/aaa>
  {
    ?aaa_variant  rdf:type         vcf2rdf:VariantCall ;
                  vc:REF           ?reference ;
                  vc:ALT           ?alternative ;
                  vc:FILTER        ?filter ;
                  faldo:reference  ?chromosome ;
                  faldo:position   ?position .
```

```
  }

  MINUS
  {
    GRAPH <http://comparison/bbb>
    {
      ?variant  rdf:type         vcf2rdf:VariantCall ;
                vc:REF           ?reference ;
                vc:ALT           ?alternative ;
                faldo:reference  ?chromosome ;
                faldo:position   ?position .
    }
  }
}
```

So the `MINUS` construct in SPARQL can be used to filter overlapping information between multiple graphs.

This query demonstrates how a fine-grained "diff" can be constructed between two datasets.

6.2 Federated querying

Now that we've seen local queries, there's only one more construct we need to know to combine this with remote SPARQL endpoints: the `SERVICE` construct.

For the next example, we will use the public SPARQL endpoint hosted by EBI.

6.2.1 Get an overview of Biomodels (from ENSEMBL)

```
PREFIX sbmlrdf: <http://identifiers.org/biomodels.vocabulary#>
PREFIX sbmldb:  <http://identifiers.org/biomodels.db/>

SELECT ?speciesId ?name {
  SERVICE <http://www.ebi.ac.uk/rdf/services/sparql/> {
    sbmldb:BIOMD0000000001 sbmlrdf:species ?speciesId .
    ?speciesId sbmlrdf:name ?name
  }
}
```

Which may yield the following table:

speciesId	name
http://identifiers.org/biomodels.db/BIOMD0000000001#_000003	BasalACh2
http://identifiers.org/biomodels.db/BIOMD0000000001#_000004	IntermediateACh
http://identifiers.org/biomodels.db/BIOMD0000000001#_000005	ActiveACh
http://identifiers.org/biomodels.db/BIOMD0000000001#_000006	Active
http://identifiers.org/biomodels.db/BIOMD0000000001#_000007	BasalACh
...	...

Table 6.7: Query results of the above query.

6.3 Tips and tricks for writing portable queries

While SPARQL has a formal standard specification, due to the different implementations of RDF stores, a query may sometimes produce an error on one endpoint, and a perfectly fine answer on another.

In this chapter we discuss ways to write "portable" queries, so that the queries can be run equally on each type of endpoint.

6.4 Provide names for aggregated columns

When using aggregated results in a column, for example by using the `COUNT` or `SUM` functions, always provide a name for the column. Let's take a look at the following example:

```
PREFIX bd: <http://www.bigdata.com/rdf#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wikibase: <http://wikiba.se/ontology#>

SELECT DISTINCT ?cause COUNT(?cause)
WHERE {
  ?human  wdt:P31    wd:Q5     ;        # Instance of human
          wdt:P509   ?cid      .        # Cause of death
  ?cid    wdt:P279*  wd:Q12078 .        # Type of cancer

  SERVICE wikibase:label
  {
    bd:serviceParam wikibase:language "[AUTO_LANGUAGE],nl" .
    ?cid rdfs:label ?cause .
  }
}
GROUP BY ?cause
```

This query displays number of occurrences, and the causes of death for humans known to Wikipedia, limited to cancer. The two columns are specified in the following line:

```
SELECT DISTINCT ?cause COUNT(?cause)
```

The first column will be named "cause", but what about the second? Some endpoints will automatically assign a unique name to the column, but others do not, and respond with an error.

To avoid this, always provide a name for such a column by using the `AS` keyword. The following line displays its usage:

```
SELECT DISTINCT ?cause (COUNT(?cause) AS ?occurrences)
```

In addition to using the `AS` keyword, also wrap the statement in parentheses, so that the SPARQL interpreter can determine which name should be assigned to which column.

Our final query looks like this:

```
PREFIX bd: <http://www.bigdata.com/rdf#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wikibase: <http://wikiba.se/ontology#>

SELECT DISTINCT ?cause (COUNT(?cause) AS ?occurrences)
WHERE {
  ?human  wdt:P31    wd:Q5     ;        # Instance of human
          wdt:P509   ?cid      .        # Cause of death
  ?cid    wdt:P279*  wd:Q12078 .        # Type of cancer

  SERVICE wikibase:label
  {
    bd:serviceParam wikibase:language "[AUTO_LANGUAGE],nl" .
    ?cid rdfs:label ?cause .
  }
}
GROUP BY ?cause
```

Chapter 7

Information management with SPARQL

In chapter 6 'Information retrieval with SPARQL' we discussed how to ask questions to SPARQL endpoints. In this chapter we will look at how we can modify the data in SPARQL endpoints.

Using SPARQL, we can write "layer 1" programs — programs that use RDF, and generate more RDF.

Like the queries from chapter 6 'Information retrieval with SPARQL', the examples can be readily used in the query editor of the web interface (see chapter 5 'Web interface').

7.1 Managing data in graphs

A simple way to subset data is to put triples in separate graphs. When uploading RDF data to an RDF store, we must provide a graph name, so this sort of works by default.

Sometimes we'd like to remove a graph altogether to make space for new datasets. For this purpose we can use the `CLEAR GRAPH` query:

```
CLEAR GRAPH <http://example>
```

After executing this query, all triples in the graph identified by `<http://example>` will be sent to a pieceful place where they cannot be accessed anymore.

The `CLEAR GRAPH` query is equivalent to the more elaborate:

```
DELETE { ?s ?p ?o }
FROM <http://example>
WHERE { ?s ?p ?o }
```

Using the `DELETE` construct, we can be more specific about which triples to remove from a graph by filling in one of the variables.

7.2 Storing inferences in new graphs

Calculating inferences from a large amount of data can take a lot of time. To avoid calculating inferences over and over again, we can store the inferred information as triples. The following example attempts to infer the gender related to a sample by looking at whether there's a mutation on the Y-chromosome.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sg:  <http://sparqling-genomics/>
```

```
PREFIX faldo: <http://biohackathon.org/resource/faldo#>
PREFIX hg19:  <http://rdf.biosemantics.org/data/genomeassemblies/hg19#>

SELECT DISTINCT ?sample
FROM <http://hmf/variant_calling>
WHERE {
  ?sample   rdf:type        sg:Sample .
  ?variant  sg:sample       ?sample ;
            faldo:reference hg19:chrY .
}
```

Each `sample` returned by this query must've originated from a male donor, because it has a Y-chromosome (and also a mutation on the Y-chromosome). Note that we cannot distinguish between females and males without a mutation on the Y-chromosome with this data, so we cannot accurately determine the gender for other samples.

For the samples that definitely originated from a male donor (according to this inference rule), we can add a triplet in the form:

```
   <sample-URI>  sg:gender sg:Male .
```

To do so, we use the `INSERT` construct:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sg:  <http://sparqling-genomics/>
PREFIX faldo: <http://biohackathon.org/resource/faldo#>
PREFIX hg19:  <http://rdf.biosemantics.org/data/genomeassemblies/hg19#>

INSERT {
  GRAPH <http://meta> {
    ?sample   sg:gender        sg:Male .
  }
}
WHERE {
  GRAPH <http://hmf/variant_calling> {
    ?sample   rdf:type         sg:Sample .
    ?variant  sg:sample        ?sample ;
              faldo:reference  hg19:chrY .
  }
}
```

After which we can query for samples that definitely originated from a male donor using the following query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sg:  <http://sparqling-genomics/>
PREFIX faldo: <http://biohackathon.org/resource/faldo#>
PREFIX hg19:  <http://rdf.biosemantics.org/data/genomeassemblies/hg19#>

SELECT (COUNT (DISTINCT ?sample)) AS ?samples
FROM <http://hmf/variant_calling>
FROM <http://meta>
```

```
WHERE {
  ?sample  rdf:type   sg:Sample ;
           sg:gender  sg:Male .
}
```

The meaning of inferences is oftentimes limited in scope. For example, inferring the gender by looking for mutations on the Y-chromosome works as long as the sequence mapping program did not accidentally map a read to the reference Y-chromosome because the X and Y chromosomes share homologous regions (El-Mogharbel & Graves, 2008).

We therefore recommend keeping inferences (layer 1) in separate graphs than observed data (layer 0) because it allows users to choose which inferences are safe to apply in a particular case.

7.3 Foreign information gathering and SPARQL

The inference example in section 7.2 'Storing inferences in new graphs' was able to create information without needing additional data that isn't described as triples.

Additional insights may require a combination of RDF triples and foreign data. In such cases, a general-purpose programming language and SPARQL can form a symbiosis. To display such a symbiosis, the following example uses the output of `vcf2rdf` to find out which samples belong to which user, by looking at the originating filenames.

Furthermore, the example uses `guile-sparql` to interact with the SPARQL endpoint and GNU Guile as general-purpose programming language.

```
(use-modules (sparql driver)
             (sparql util)
             (sparql lang))

(define %output-directory "/data/output")
(define %query "
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sg: <http://sparqling-genomics/>
PREFIX vcf2rdf: <http://sparqling-genomics/vcf2rdf/>

SELECT ?origin ?filename
WHERE {
  ?sample rdf:type         sg:Sample ; sg:foundIn  ?origin .
  ?origin vcf2rdf:filename ?filename .
}")

(define (gather-ownership-info)
  (let* (;; Gather the origins and filenames from the SPARQL endpoint.
         (origins (query-results->list (sparql-query %query)))

         ;; We are going to store triples in this file.
         (ownership-file (string-append %output-directory "/ownership.n3"))

         ;; Define ontology prefixes.
         (rdf        (prefix "http://www.w3.org/1999/02/22-rdf-syntax-ns#"))
         (sg         (prefix "http://sparqling-genomics/"))
         (vcf2rdf    (prefix "http://sparqling-genomics/vcf2rdf/"))
```

```
          (user       (prefix "http://sparqling-genomics/User/"))
          (user-class "<http://sparqling-genomics/User>")
          (owner-pred "<http://sparqling-genomics/owner>"))

  ;; Generate triples for each entry.
  (call-with-output-file ownership-file
    (lambda (port)
      (for-each (lambda (entry)
                  ;; Extract the username of a file.
                  (let ((owner-name (passwd:name
                                      (getpwuid
                                        (stat:uid (stat (cadr entry)))))))
                  ;; Write RDF triples to the file.
                  (format port "~a ~a ~a .~%"
                               (user owner-name) (rdf "type") user-class)
                  (format port "<~a> ~a ~a .~%"
                               (car entry) owner-pred user-class)))
                (cdr origins))))))

(gather-ownership-info)
```

For a small amount of files, we could directly execute an `INSERT` query on the SPARQL endpoint, however, for a large amount of files we may want to use the RDF store's data loader for better performance.

This program provides the triples that enables us to find which user contributed which variant call data in the graph:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sg:  <http://sparqling-genomics/>
PREFIX vcf2rdf:  <http://sparqling-genomics/vcf2rdf/>

SELECT DISTINCT ?sample ?filename ?user
FROM <http://hmf/variant_calling>
FROM <http://ownership> # Assuming we the imported data into this graph
WHERE {
  ?sample   rdf:type         sg:Sample ;
            sg:foundIn       ?origin   .

  ?origin   vcf2rdf:filename ?filename ;
            sg:owner         ?user     .
}
```

Chapter 8

Using SPARQL with other programming languages

8.1 Using SPARQL with R

Before we can start, we need to install the `SPARQL` package from CRAN.

```
install.packages("SPARQL")
```

Once the package is installed, we can load it:

```
library("SPARQL")
```

Let's define where to send the query to:

```
endpoint <- "http://localhost:8890/sparql"
```

. . . and the query itself:

```
query <- "PREFIX vcf2rdf: <http://sparqling-genomics/vcf2rdf/>
PREFIX vc:    <http://sparqling-genomics/vcf2rdf/VariantCall/>
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX faldo: <http://biohackathon.org/resource/faldo#>

SELECT DISTINCT ?variant ?sample ?chromosome ?position ?filter
FROM <graph-name>
WHERE
{
  ?variant  rdf:type               vcf2rdf:VariantCall ;
            vcf2rdf:sample         ?sample ;
            faldo:reference        ?chromosome ;
            faldo:position         ?position ;
            vc:FILTER              ?filter .
}
LIMIT 10";
```

To actually execute the query, we can use the `SPARQL` function:

```
query_data <- SPARQL (endpoint, query)
```

If the query execution went fine, we can gather the resulting `dataframe` from the `results` index.

```
query_results <- query_data$results
```

8.1.1 Querying with authentication

When the SPARQL endpoint we try to reach requires authentication before it accepts a query, we can use the `curl_args` parameter of the `SPARQL` function.

In the following example, we use `dba` as username, and `secret-password` as password.

```
endpoint     <- "http://localhost:8890/sparql-auth"
auth_options <- curlOptions(userpwd="dba:secret-password")
query        <- "SELECT DISTINCT ?p WHERE { ?s ?p ?o }"
query_data   <- SPARQL (endpoint, query, curl_args=auth_options)
results      <- query_data$results
```

8.2 Using SPARQL with GNU Guile

For Schemers using GNU Guile, the guile-sparql[1] package provides a SPARQL interface.

The package provides a `driver` module that communicates with the SPARQL endpoint, a `lang` module to construct SPARQL queries using S-expressions, and a `util` module containing convenience functions.

After installation, the modules can be loaded using:

```
(use-modules (sparql driver)
             (sparql lang)
             (sparql util))
```

Using the `sparql-query` function, we can execute a query:

```
(let ((endpoint       "http://localhost:8890/sparql-auth")
      (authentication "dba:secret-password")
      (query          "SELECT DISTINCT ?p WHERE { ?s ?p ?o }"))
  (display-query-results-of
    (sparql-query query
                  #:uri    endpoint
                  #:digest authentication)))
```

[1] https://github.com/roelj/guile-sparql

References

Beacon Network. (2018). Global Alliance for Genomics and Health. Retrieved from https://beacon-network.org

Bolleman, J. T., Mungall, C. J., Strozzi, F., Baran, J., Dumontier, M., Bonnal, R. J. P., … Cock, P. J. A. (2016, Jun 13). Faldo: a semantic standard for describing the location of nucleotide and protein feature annotation. *Journal of Biomedical Semantics*, *7*(1), 39. Retrieved from https://doi.org/10.1186/s13326-016-0067-z doi: 10.1186/s13326-016-0067-z

DCMI Metadata Terms. (2012). Dublin Core Metadata Initiative. Retrieved from http://dublincore.org/documents/2012/06/14/dcmi-terms/

El-Mogharbel, N., & Graves, J. A. (2008). X and y chromosomes: Homologous regions. In *els.* American Cancer Society. Retrieved from https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470015902.a0005793.pub2 doi: 10.1002/9780470015902.a0005793.pub2

Lassila, O. (1999, February). Resource description framework (RDF) model and syntax specification [W3C Recommendation]. (http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/)

Lebo, T., McGuinness, D., & Sahoo, S. (2013, April). PROV-o: The PROV ontology [W3C Recommendation]. (http://www.w3.org/TR/2013/REC-prov-o-20130430/)

Mungall, C., Vasilevsky, N., Matentzoglu, N., Osumi-Sutherland, D., Slater, L., Dahdul, W., … Meier, A. (2020, March). PATO - the Phenotype And Trait Ontology. Zenodo. (https://doi.org/10.5281/zenodo.3726344)

SPARQL 1.1 overview [W3C Recommendation]. (2013, March). (http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/)