



<https://github.com/UMCUGenetics/sparqling-genomics>
v0.99.10, July 9, 2019

Contents

1	Getting started	1
1.1	Prerequisites	1
1.2	Setting up a build environment	2
1.2.1	Debian	2
1.2.2	CentOS	2
1.2.3	GNU Guix	2
1.2.4	MacOS	2
1.3	Obtaining the source code	2
1.4	Installation instructions	3
1.5	Using a pre-built Docker image	3
2	The knowledge graph	4
3	Use of ontologies	6
3.1	Dublin Core Terms	6
3.2	Describing genomic positions with FALDO	6
3.3	Custom terms	7
4	Command-line programs	8
4.1	Preparing variant call data with vcf2rdf	8
4.1.1	Knowledge extracted by vcf2rdf	8
4.1.2	Example usage	9
4.1.3	Run-time properties	9
4.2	Preparing sequence alignment maps with bam2rdf	9
4.2.1	Knowledge extracted by bam2rdf	9
4.3	Preparing tabular data with table2rdf	10
4.3.1	Transforming objects	10
4.3.2	Transforming predicates	11
4.3.3	Delimiters	12
4.3.4	Knowledge extracted by table2rdf	13
4.3.5	Example usage	13
4.4	Converting MySQL data to RDF with table2rdf	14
4.5	Extracting knowledge from folders with folder2rdf	14
4.5.1	Example usage	14
4.5.2	Knowledge extracted by folder2rdf	14
4.6	Importing data with curl	15
4.6.1	Example usage	15
5	Web interface	16
5.1	Configuring the web interface	16
5.1.1	To fork or not to fork	16

5.1.2	Bind address and port	16
5.1.3	System connection	16
5.1.4	Authentication	17
5.2	Running the web interface	18
5.3	Configuring connections	18
5.4	Managing projects	18
5.5	Executing queries	19
5.5.1	Query history	19
5.6	Explore graphs with the Exploratory	19
5.6.1	Connections and graphs	19
5.6.2	Types	20
5.6.3	Predicates	20
6	Information retrieval with SPARQL	21
6.1	Local querying	21
6.1.1	Listing non-empty graphs	21
6.1.2	Querying a specific graph	22
6.1.3	Exploring the structure of knowledge in a graph	22
6.1.4	Listing samples and their originating files	23
6.1.5	Listing samples, originated files, and number of variants	24
6.1.6	Retrieving all variants	25
6.1.7	Retrieving variants with a specific mutation	25
6.1.8	Comparing two datasets on specific properties	26
6.2	Federated querying	27
6.2.1	Get an overview of Biomodels (from ENSEMBL)	27
6.3	Tips and tricks for writing portable queries	28
6.4	Provide names for aggregated columns	28
7	Information management with SPARQL	30
7.1	Managing data in graphs	30
7.2	Storing inferences in new graphs	30
7.3	Foreign information gathering and SPARQL	32
8	Using SPARQL with other programming languages	34
8.1	Using SPARQL with R	34
8.1.1	Querying with authentication	35
8.2	Using SPARQL with GNU Guile	35

Chapter 1

Getting started

1.1 Prerequisites

The programs provided by this project build a knowledge graph. However, a knowledge graph store (better known as an RDF store) is not included.

Through the years various great RDF stores have been developed, 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 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	libgcrypt	raptor2	libxml2	HTSLib	zlib	GNU Guile	GnuTLS	\LaTeX
vcf2rdf	•	•	•		•				
bam2rdf	•	•	•		•				
table2rdf	•	•	•			•			
json2rdf	•	•	•			•			
xml2rdf	•	•	•	•		•			
folder2rdf							•		
sg-web							•	•	
Documentation									•

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

The manual provides example commands to import RDF using [cURL](#).

1.2 Setting up a build environment

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 libgcrypt-dev \
                zlib-dev guile-2.0 guile-2.0-dev libraptor2-dev texlive \
                curl libxml2-dev
```

1.2.2 CentOS

CentOS 7 does not include `htslib`. All other dependencies can be installed using the following command:

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

1.2.3 GNU Guix

If **GNU Guix** is available on your system, an environment that contains all external tools required to build the programs in this project can be obtained running the following command from the project's repository root:

```
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 libgcrypt guile \
            htslib curl raptor libxml2
```

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**¹ page. Make sure to download the `sparqling-genomics-0.99.10.tar.gz` file.

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

```
curl -LO https://github.com/UMCUGenetics/sparqling-genomics/releases/\
download/0.99.10/sparqling-genomics-0.99.10.tar.gz
```

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

¹<https://github.com/UMCUGenetics/sparqling-genomics/releases>

```
tar zxvf sparqling-genomics-0.99.10.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-0.99.10
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, the 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/0.99.10/sparqling-genomics-0.99.10-docker.tar.gz
docker load < sparqling-genomics-0.99.10-docker.tar.gz
```

The container contains 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 the explicitly stated facts. 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* \rightarrow *predicate* \rightarrow *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.

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.

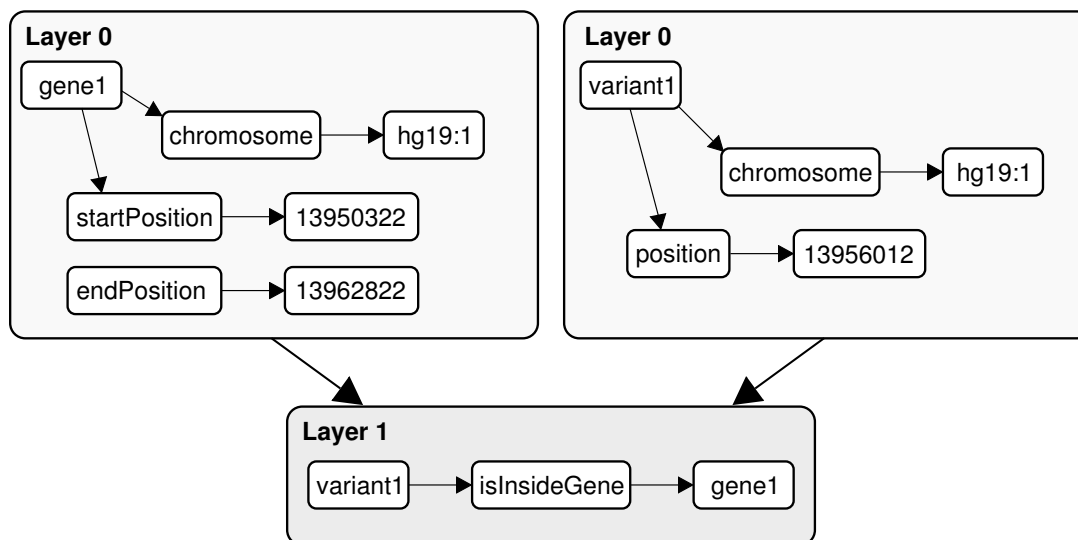


Figure 2.1: Illustrating knowledge layers. In the figure we have knowledge from two separate sources (gene locations and genomic variants) from which we can derive new knowledge in an inference layer.

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 (rounding variant positions to the nearest 1000).

The knowledge graph contains two types of nodes; uniquely identifiable names having a symbolic value (1), and literal values like numbers and text (2). The symbolic values are written as URIs, for which all symbols defined by programs that are part of SPARQLing genomics share a common prefix: `<http://sparqling-genomics/>`. When we describe a node in the remainder of the manual, we shorten the URI with this prefix. For example, to describe the URI `<http://sparqling-genomics/Origin/1ec192jh5>`, we could equally write `:Origin/1ec192jh5`, where the colon means “use the common prefix `<http://sparqling-genomics/>`”.

The programs that are part of SPARQLing genomics use a few patterns to come up with identifiers. For example, to link facts to their original (non-RDF) source, we use the type `:Origin`. When describing the originated file of some facts, we use `:Origin/<SHA256 sum of the file>`, so that we can be sure that when the same identifier occurs, the originating bytes are identical.

In chapter 4 ‘**Command-line programs**’ we define more types, all of them built up from the pattern `:<type name>/<string>`. This can be used as a guideline to interact and extend the knowledge graph.

We attempt to provide the practical tools to build and maintain a flexible knowledge graph, and these tools may change over time. When writing new tools or changing existing ones, please consider the effect on the knowledge graph first.

Chapter 3

Use of ontologies

Throughout *sparqling-genomics* we use ontologies defined by external working groups to increase interoperability with foreign systems. In this chapter, we explain which ontologies we use, and how we use them.

3.1 Dublin Core Terms

In the web interface, we use the Dublin Core Terms ontology ([“DCMI Metadata Terms”, 2012](#)) to define organizations, collections, datasets, and samples. Table 3.1 provides an overview of the properties we use.

Term	Used as	Description
dcterms:Agent	Class	Used by the web interface to describe the organization that produced a collection or dataset.
dctype:Collection	Class	Used by the portal page in the web interface as a filter mechanism to browse dctype:Datasets..
dctype:Dataset	Class	Used by the portal page in the web interface to describe data in a graph.
dcterms:isPartOf	Predicate	Used by the portal page in the web interface to express that a dctype:Dataset is linked to a dctype:Collection.
dcterms:title	Predicate	Used by the web interface to name a dctype:Collection or a dctype:Dataset.
dcterms:publisher	Predicate	Used by the web interface to identify the organization that published a collection or dataset.
dc:description	Predicate	Used by the web interface to provide a textual description of a dctype:Collection or a dctype:Dataset.

Table 3.1: Terms used from the Dublin Core Terms ontology.

3.2 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 3.2 provides an overview of the properties we use.

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

Table 3.2: Terms used from FALDO.

3.3 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 3.3 summarizes the terms that are waiting to be replaced by an external ontology.

Term	Used as	Usage
<code>sg:Origin</code>	Class	Used by the command-line tools (see chapter 4 ‘ <i>Command-line programs</i> ’) to point to the file or resource from which information was derived.
<code>sg:containsDataFor</code>	Class	Used by the portal page in the web interface to express which graph (object) belongs to which dataset (subject).
<code>sg:mappedToGenomeAssembly</code>	Class	Used by the portal page in the web interface to describe the reference genome assembly used for a <code>dctype:Dataset</code> .
<code>sg:Sample</code>	Class	Used by <code>vcf2rdf</code> to describe a sample.

Table 3.3: Terms made up by us.

Chapter 4

Command-line programs

SPARQLing genomics provides programs to create an extensive knowledge graph from genomics-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`

Obtaining variants from sequenced data is a task performed by *variant callers*. These programs often output the variants they found in the *Variant Call Format* (VCF). The `vcf2rdf` program extracts knowledge from a VCF 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
<code>:Origin/identifier</code>	<code>rdf:type</code>	<code>:Origin</code>	This defines a uniquely identifiable reference to the originating file.
<code>:Origin/identifier</code>	<code>:filename</code>	<i>filename</i>	This triple states the originating filename.
<code>:Origin/identifier</code>	<code>:sha256sum</code>	<i>SHA256 sum</i>	This triple states the SHA256 sum of the content of the original file.
<code>:Sample/sample name</code>	<code>rdf:type</code>	<code>:Sample</code>	This states that there is a sample with <i>sample name</i> .
<code>:Sample/sample name</code>	<code>:foundIn</code>	<code>:Origin/identifier</code>	This triple states that a sample can be found in a file identified by the <code>:Origin</code> with a specific identifier.
<code>:Origin/identifier</code>	<code>:convertedBy</code>	<code>:vcf2rdf-0.99.10</code>	This triple states that the file was converted with <code>vcf2rdf</code> .

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

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

```
<http://sparqling-genomics/Origin/14f2b609b>
  :convertedBy :vcf2rdf-0.99.10 ;
  :filename "clone_ref_tumor.vcf.gz"^^xsd:string ;
  :sha256sum "14f2b609b" ;
  a :Origin .

<http://sparqling-genomics/Sample/CLONE_REF>
  :foundIn <http://sparqling-genomics/Origin/14f2b609b3> ;
  a :Sample .

<http://sparqling-genomics/Sample/CLONE_TUMOR>
  :foundIn <http://sparqling-genomics/Origin/14f2b609b3> ;
  a :Sample .
```

4.1.2 Example usage

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

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

```
vcf2rdf --help
```

4.1.3 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 ntriples mode.

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

4.2 Preparing sequence alignment maps with bam2rdf

Aligning reads from a DNA sequencer to a predetermined *reference genome* is a task performed by *read mapper* programs. Oftentimes, the output produced by these programs are in the *sequence alignment map* (SAM) format, or its equivalent *binary alignment map* (BAM) format. The bam2rdf program can read data in either format.

4.2.1 Knowledge extracted by bam2rdf

The current version of bam2rdf merely extracts information from the alignment map header.

Subject	Predicate	Object	Description
<code>:Origin/identifier</code>	<code>rdf:type</code>	<code>:Origin</code>	This defines a uniquely identifiable reference to the originating file.
<code>:Origin/identifier</code>	<code>:filename</code>	<i>filename</i>	This triple states the originating filename.
<code>:bam2rdf/unique identifier</code>	<code>rdf:type</code>	One of: <code>:bam2rdf/HeaderItem</code> , <code>:bam2rdf/ReferenceSequence</code> , <code>:bam2rdf/ReadGroup</code> , <code>:bam2rdf/Program</code> , <code>:bam2rdf/Comment</code> .	The <i>objects</i> correspond to the various types of header lines that can occur in the SAM format.
<code>:bam2rdf/unique identifier</code>	<code>:foundIn</code>	<code>:Origin/identifier</code>	This triple states that a header line can be found in a file identified by the <code>:Origin</code> with a specific identifier.
<code>:bam2rdf/unique identifier</code>	<i>type class/key</i>	Literal value.	Each header field consists of a key/value pair. The key is used as predicate.
<code>:Origin/identifier</code>	<code>:convertedBy</code>	<code>:bam2rdf</code>	This triple states that the file was converted with <code>bam2rdf</code> .

Table 4.2: The additional triple patterns provided by `bam2rdf`.

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
...
<http://sparqling-genomics/table2rdf/Row/...-R0000000000>
  sg:originatedFrom <http://sparqling-genomics/...> ;
  col:chromosome "chr1"^^xsd:string ;
  col:position 1500000 ;
  a :Row .

<http://sparqling-genomics/table2rdf/Row/...-R0000000001>
  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 \
  -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 `http://sparqling-genomics/table2rdf/Column/chromosome`). By using the `--transform-predicate` option, or `-T` for short, a different transformation can be made:

```

$ table2rdf \
  -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#> .

<http://sparqling-genomics/table2rdf/Row/...-R00000000000>
  sg:originatedFrom <http://sparqling-genomics/...> ;
  p00001:reference p00000:chr1 ;
  col:position 1500000 ;
  a :Row .

<http://sparqling-genomics/table2rdf/Row/...-R00000000001>
  sg:originatedFrom <http://sparqling-genomics/...> ;
  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
...
<http://sparqling-genomics/table2rdf/Row/...-R00000000000>
  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 ";"
...
<http://sparqling-genomics/table2rdf/Row/...-R0000000000>
  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
<code>:Origin/identifier</code>	<code>rdf:type</code>	<code>:Origin</code>	This defines a uniquely identifiable reference to the originating file.
<code>:Origin/identifier</code>	<code>:filename</code>	<i>filename</i>	This triple states the originating filename.
<code>:Origin/identifier</code>	<code>:convertedBy</code>	<code>:table2rdf</code>	This triple states that the file was converted with <code>table2rdf</code> .
<code>:Sample/sample name</code>	<code>rdf:type</code>	<code>:Sample</code>	This states that there is a sample with <i>sample name</i> .
<code>:Sample/sample name</code>	<code>:foundIn</code>	<code>:Origin/identifier</code>	This triple states that a sample can be found in a file identified by the <code>:Origin</code> with a specific identifier.

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

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

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

sample:grch37
  :foundIn <http://sparqling-genomics/table2rdf/1jka8923i4> ;
  a :Sample .
```

4.3.5 Example usage


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

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 -O ntriples > 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 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`.

4.5.1 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.

4.5.2 Knowledge extracted by folder2rdf

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

Subject	Predicate	Object	Description
<code>:Project/project-name</code>	<code>rdf:type</code>	<code>:Project</code>	This defines the identifier for the project.
<code>:User/username</code>	<code>rdf:type</code>	<code>:User</code>	This defines the identifier for the file owner (username).
<code>:Origin/identifier</code>	<code>rdf:type</code>	<code>:Origin</code>	This defines a uniquely identifiable reference to the originating file.

Table 4.4: The additional triple patterns produced by `folder2rdf`.

4.6 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.

4.6.1 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

In addition to the command-line programs, the project provides a web interface for prototyping queries, and quick data reporting. With the web interface you can:

- Write and execute SPARQL queries;
- Collaborate within “projects”;
- Browse available datasets;
- Explore the inner-structure of datasets.

5.1 Configuring 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 5.1.4 ‘Authentication’.

```
<?xml version="1.0" encoding="utf-8"?>
<web-interface>
  <fork>0</fork>
  <bind-address>127.0.0.1</bind-address>
  <port>8080</port>
  <authentication>
    <!-- Either LDAP settings, or single-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

Because web services are popular these days, sg-web can be configured to bind on an arbitrary address and an arbitrary port.

5.1.3 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>
```

5.1.4 Authentication

There are two ways to configure authentication. For single-user deployments or environments that lack an LDAP service, a preconfigured username and password can be set. For a multi-user deployment, the web interface can be configured to use an LDAP server.

Single-user configuration

The simplest form of authentication is the “single-user configuration”. Configuring it involves providing a username and the SHA256 sum of a password. The following example shows how to configure “single-user authentication”:

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

LDAP authentication example

To configure LDAP, three parameters must be specified: the URI to the LDAP service (1), the “organizational unit” (2), and the “domain” (3). The username is used as the “common name”.

The following example shows how to configure LDAP authentication:

```
<?xml version="1.0" encoding="utf-8"?>
<web-interface>
  ...
  <authentication>
    <ldap>
      <uri>ldap://example.local</uri>
      <organizational-unit>People</organizational-unit>
      <domain>department.organization.tld</domain>
    </ldap>
  </authentication>
</web-interface>
```

5.2 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 ‘Configuring the web interface’.

5.3 Configuring connections

The first useful step is to configure a connection to a SPARQL endpoint.

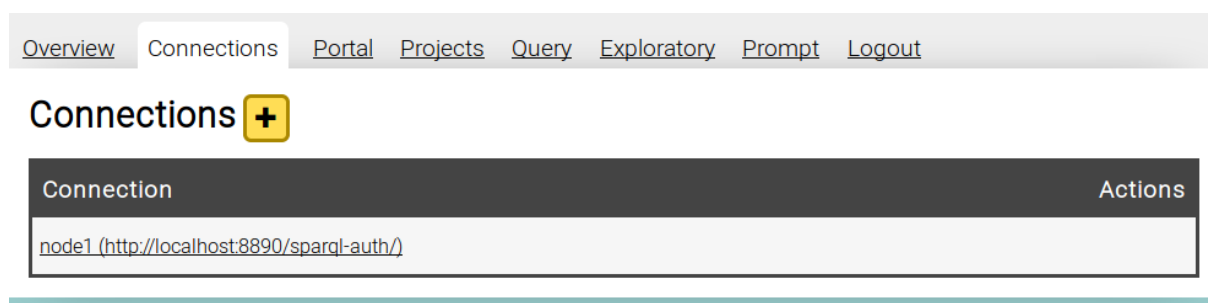


Figure 5.1: The *connections* page enables users to configure accessible SPARQL endpoints. Adding a connection here will provide an option to query it on the *query* page.

5.4 Managing projects

Projects are a loosely-defined way to group queries and to collaborate with other users. Projects provide a way to manage access to *graphs*, and to share previously-executed queries among project members.

Marking a project as “active” indicates that queries executed using the web interface relate to that project. See also section 5.5.1 ‘Query history’.

5.5 Executing queries

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

Overview Connections Portal Projects **Query** Exploratory Prompt Logout

Query the database

Select a connection

node1

Query editor

Use **Ctrl + Enter** to execute the query. (**Cmd + Enter** for the unfortunate MacOS users.)

```
1 SELECT DISTINCT ?graph WHERE { GRAPH ?graph { ?s ?p ?o } }
2 LIMIT 3
```

Query results

Show 10 entries

graph
http://www.openlinksw.com/schemas/virtrdf#
http://www.w3.org/ns/ldp#
https://www.ncbi.nlm.nih.gov/snp

Showing 1 to 3 of 3 entries Previous 1 Next

Figure 5.2: 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.

5.5.1 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, and newly executed queries are added to the current *active* project.

5.6 Explore graphs with the Exploratory

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


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.

5.6.1 Connections and graphs

The first step in finding predicates involves choosing a connection (see section 5.3 ‘*Configuring connections*’). The second step involves choosing a graph. If the connection does not support the use of graphs, the journey ends here.

[Overview](#)
[Connections](#)
[Portal](#)
[Projects](#)
[Query](#)
[Exploratory](#)
[Prompt](#)
[Logout](#)

Exploratory

The exploratory provides an alternative interface to explore the structure of data available at each connection. It is optimized for speed, allowing it to show outdated information. By using the  button, you can request the most recent data. This may be a bit slower than showing the outdated information.

Connections	Graphs	Types	Predicates
node1	http://roel/sg-cache http://sparqling-genomics.org/sg-web/state http://www.openlinksw.com/schemas/virttrdf# http://www.w3.org/ns/ldp# https://gnomad.broadinstitute.org/exomes https://gnomad.broadinstitute.org/genomes https://node1.roelj.com/portal https://www.ncbi.nlm.nih.gov/clinvar https://www.ncbi.nlm.nih.gov/snp	sg-old:Origin vcf2rdf:FilterItem vcf2rdf:HeaderItem vcf2rdf:InfoItem vcf2rdf:VariantCall	rdf:type sg-old:convertedBy sg-old:filename sg-old:sha256sum

A list of connections is stored internally.

To get the graphs, the following query is used:

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

Types are determined using the following query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?type
WHERE {
  GRAPH <https://www.ncbi.nlm.nih.gov/clinvar> {
    ?s rdf:type ?type .
  }
}
```

Predicates are found using the following query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?predicate
WHERE {
  GRAPH <https://www.ncbi.nlm.nih.gov/clinvar> {
    ?s rdf:type <sg-old:Origin> ;
    ?predicate ?o .
  }
}
```

Figure 5.3: The *exploratory* page enables users to learn about the structure of the triplets in a graph.

5.6.2 Types

The third step looks for triplets that match the pattern $subject \rightarrow rdf:type \rightarrow type$. 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.

5.6.3 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.

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 4.6 ‘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://localuriquaserver/sparql
http://www.openlinksw.com/schemas/virttrdf#
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 literal¹ 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 }
```

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

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

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:F00`, the SPARQL endpoint interprets it as if we would write `<http://sparqling-genomics/vcf2rdf/F00>`.

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 sbmlldb: <http://identifiers.org/biomodels.db/>

SELECT ?speciesId ?name {
    SERVICE <http://www.ebi.ac.uk/rdf/services/sparql/> {
        sbmlldb: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](https://github.com/roelj/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

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