sparqling-genomics

https://github.com/UMCUGenetics/sparqling-genomics

v0.99.5, August 14, 2018

Contents

Chapter 1

Getting started

1.1 Prerequisites

In addition to the tools provided by this project, a RDF store is required. In the manual we use Virtuoso, but 4store, BlazeGraph, or AllegroGraph may also be used.

Before we can use the programs provided by this project, we need to build them first.

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 optional.

Each component in the repository has its own dependencies. Table 1.1 provides an overview for each tool.

vcf2rdf	table2rdf	Web interface	Documentation
GNU C compiler	GNU C compiler	GNU Guile	LaTeX distribtion
libgcrypt	libgcrypt	Fibers	
`raptor2`	`raptor2`		
HTSLib			

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
```

The command has been tested on Debian 9.

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
```

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
```

The only missing dependency is a LaTeX distribution. But this is only needed to build this documentation.

Building on MacOS has not been tested. If you've tried it, please let us know, so we can attempt to support it in the future.

1.3 Obtaining the source code

The source code can be downloaded at the Releases[1] page. Make sure to download the `sparqling-genomics-0.99.5.tar.gz` file.

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

```
curl -O https://github.com/UMCUGenetics/sparqling-genomics/releases/\
download/0.99.5/sparqling-genomics-0.99.5.tar.gz
```

After obtaining the tarball, it can be unpacked using the following commands:

```
tar zxvf sparqling-genomics-0.99.5.tar.gz
cd sparqling-genomics-0.99.5
```

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:

```
autoreconf -vif # Only needed if the "./configure" step does not work.
./configure
make
```

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

```
make install
```

To run the `make install` command, super user privileges are possibly required. This step can be ignored, but will keep the tools in the project's directory. So, 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 && make install` with `make -C <component-directory>`. So, to only build `vcf2rdf`, the following command could be used:

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

Chapter 2

The knowledge graph

The tools provided by `sparqling-genomics` are designed to build a common format to express genomic information. Each program reads data in a domain-specific format, and translates it into a common format; the Resource Description Framework (RDF).

Programs can be categorized in layers. A program belongs to the first layer (layer 0) when it translates a non-RDF format into RDF. In the second layer (layer 1), we find programs that read RDF and generate more RDF. Higher-level layers depend on the knowledge added by programs from the previous layer.

In `sparqling-genomics`, the knowledge graph created by the programs is more important than the programs themselves. When designing and implementing new programs, we should consider the added knowledge first.

Furthermore, programs should not depend on programs, but on the knowledge produced by programs. For example, the `vcf2rdf` program always writes genomic positions by using the *FALDO* ontology. An annotation program needs not to know about the existence of `vcf2rdf`, but it needs to know about the *FALDO* ontology. Therefore, the common interface between programs dealing with genomic positions is the *FALDO* ontology. This enables developers of the knowledge graph to understand the bigger picture without needing to understand the details of each program, or each individual data format.

The next challenge is to describe knowledge in an integrative manner. Again, *FALDO* serves a good example: it describes a way of expressing knowledge that multiple programs can use; locations in a genome. Developing effective ontologies means extracting common patterns in how information is described. This is an ever-ongoing process of refinement that changes over time with the knowledge that is most valuable to the researcher.

With `sparqling-genomics`, we attempt to design a knowledge graph and provide the tools to practically implement it. When improving `sparqling-genomics`, please always keep an eye out for the knowledge graph.

2.1 Knowledge expressed by SPARQLing genomics tools

The `vcf2rdf` and `table2rdf` programs express knowledge using a unified pattern. This overlap between the programs defines the *ontology* for SPARQling genomics. We wrote the ontology in the Terse RDF Triple Language (Turtle) format, and it can be found in the release tarball in '`ontologies/sparqling-genomics.ttl`'.

Chapter 3

Command-line programs

The project provides programs to create a complete pipeline including data conversion, data importing and data exploration. The tasks we can perform with the command-line programs are:

- Extract triples from VCF files;
- Extract triples from tabular data files;
- Push data to a SPARQL endpoint.

3.1 Preparing variant call data with `vcf2rdf`

Obtaining variants from sequenced data is a task of so called *variant callers*. These programs often output the variants they found in the *Variant Call Format* (VCF). Before we can use the data described in this format, we need to extract *knowledge* (in the form of triples) from it.

The `vcf2rdf` program does exactly this, by converting a VCF file into an RDF format. In section 3.4 'Importing data with `curl`' we describe how to import the data produced by `vcf2rdf` in the database.

3.1.1 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` stores the following metadata:

Subject	Predicate	Object	Description
:Origin	rdf:type	owl:Class	`:Origin` is used to identify a data origin (which is usually a file).
:Sample	rdf:type	owl:Class	`:Sample` is used to identify a sample name.
:filename	rdf:type	xsd:string	`:filename` contains the path to the file that `:Origin` represents.
:convertedBy	rdf:type	owl:AnnotationProperty	`:convertedBy` is used to identify the program that performed the VCF->RDF conversion.
:foundIn	rdf:type	owl:AnnotationProperty	`:foundIn` relates the `:Origin` to a `:Sample`.

Table 3.1: The additional triple patterns described by `vcf2rdf`.

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

```
<http://rdf.umcutrecht.nl/vcf2rdf/14f2b609b>
    :convertedBy :vcf2rdf ;
    :filename "clone_ref_tumor.vcf.gz"^^xsd:string ;
    a :Origin .

sample:CLONE_REF
    :foundIn <http://rdf.umcutrecht.nl/vcf2rdf/14f2b609b3> ;
    a :Sample .

sample:CLONE_TUMOR
    :foundIn <http://rdf.umcutrecht.nl/vcf2rdf/14f2b609b3> ;
    a :Sample .
```

3.1.2 Example usage

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

3.1.3 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.

3.2 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 (`;`).

3.2.1 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	rdf:type	owl:Class	`:Origin` is used to identify a data origin (which is usually a file).
:Sample	rdf:type	owl:Class	`:Sample` is used to identify a sample name.
:filename	rdf:type	xsd:string	`:filename` contains the path to the file that `:Origin` represents.
:convertedBy	rdf:type	owl:AnnotationProperty	`:convertedBy` is used to identify the program that performed the VCF->RDF conversion.
:foundIn	rdf:type	owl:AnnotationProperty	`:foundIn` relates the `:Origin` to a `:Sample`.

Table 3.2: The additional triple patterns described by `table2rdf`.

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

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

sample:grch37
    :foundIn <http://rdf.umcutrecht.nl/table2rdf/1jka8923i4> ;
    a :Sample .
```

3.2.2 Example usage

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

3.3 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.

3.4 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.

3.4.1 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 4

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;
- Combine multiple SPARQL endpoints;
- Configure “projects” to subset data.

4.1 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 4.1.3 ‘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>
```

4.1.1 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.

4.1.2 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.

4.1.3 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". Make sure the configuration file is only readable by yourself, as the credentials are stored in plain text. 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>
      <password>test</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>
```

4.2 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 4.1 'Configuring the web interface'.

4.3 Configuring connections

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

When providing a username and password for a connection, it will attempt to connect using *digest authentication.*

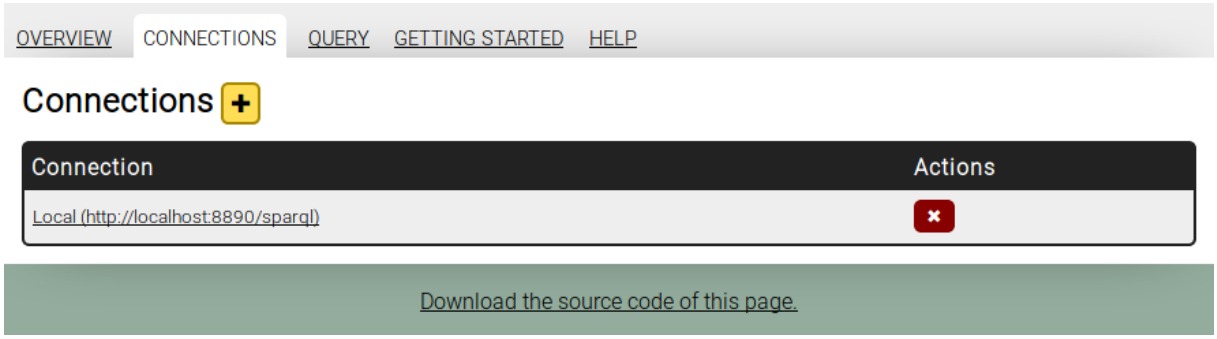

Figure 4.1: The *connections* page enables users to configure accessible SPARQL endpoints. Adding a connection here will provide an option to query it on the *query* page.

4.4 Executing queries

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

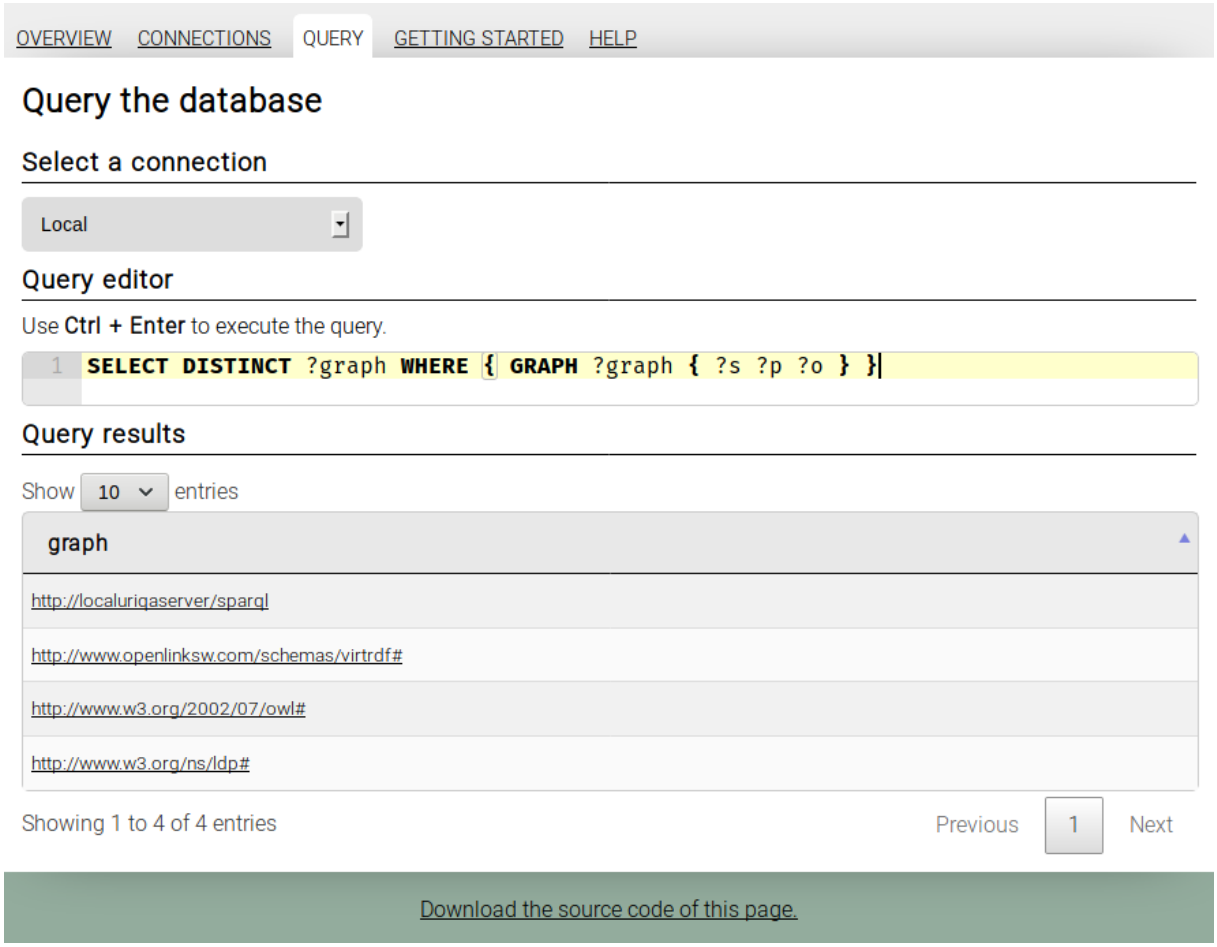

Figure 4.2: The *query* page enables users to execute a query against a SPARQL endpoint. The connections configured at the *connections* page can be chosen from the drop-down menu.

Chapter 5

Information retrieval with SPARQL

In section 3.1 'Preparing variant call data with `vcf2rdf`' we discussed how to extract triples from common data formats. In section 3.4 '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 your own programs or scripts — something we will discuss in chapter 6 '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 4 'Web interface').

5.1 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 5.2 'Federated querying'.

5.1.1 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 5.1: 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.

5.1.2 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 5.1.1 '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 5.2 'Federated querying' we will look at querying over multiple RDF stores.

5.1.3 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 }
```

[1] Examples 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 5.2: 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://rdf.umcutrecht.nl/vcf2rdf/filename
http://rdf.umcutrecht.nl/vcf2rdf/foundIn
http://rdf.umcutrecht.nl/vcf2rdf/sample
http://rdf.umcutrecht.nl/vcf2rdf/VariantCall/ALT
http://rdf.umcutrecht.nl/vcf2rdf/VariantCall/FILTER
...

Table 5.3: Results of the query to list predicates.

5.1.4 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://rdf.umcutrecht.nl/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 5.4: Results of the query to list samples and their originating filenames.

Notice how most predicates for vcf2rdf in table 5.3 start with http://rdf.umcutrecht.nl/vcf2rdf/. In the above query, we used this to shorten the query. We started the query by writing a PREFIX rule for http://rdf.umcutrecht.nl/vcf2rdf/, which we called vcf2rdf:. This means that whenever we write vcf2rdf:FOO, the SPARQL endpoint interprets it as if we would write <http://rdf.umcutrecht.nl/vcf2rdf/FOO>.

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

5.1.5 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://rdf.umcutrecht.nl/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 5.5: 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.

5.1.6 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://rdf.umcutrecht.nl/vcf2rdf/>
PREFIX vc:      <http://rdf.umcutrecht.nl/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.

5.1.7 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://rdf.umcutrecht.nl/vcf2rdf/>
PREFIX vc:    <http://rdf.umcutrecht.nl/vcf2rdf/VariantCall/>
PREFIX seq:   <http://rdf.umcutrecht.nl/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://rdf.umcutrecht.nl/vcf2rdf/Sample/REF0047
5	http://rdf.umcutrecht.nl/vcf2rdf/Sample/TUMOR0047

Table 5.6: Query results of the above query.

5.1.8 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://rdf.umcutrecht.nl/vcf2rdf/>
PREFIX vc:      <http://rdf.umcutrecht.nl/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.

5.2 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.

5.2.1 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 5.7: Query results of the above query.

Chapter 6

Using SPARQL with other programming languages

6.1 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://rdf.umcutrecht.nl/vcf2rdf/>
PREFIX vc:    <http://rdf.umcutrecht.nl/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
```

6.1.1 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
```

6.2 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