sparqling-genomics

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

v0.99.0, July 9, 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	Web interface	Documentation
GNU C compiler	GNU Guile	LaTeX distribtion
libgcrypt		
HTSLib		
`raptor2`		

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

We suggest cURL to import RDF to a triple store. 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 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 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
```

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 Installation instructions

After installing the required tools (see section 1.1 'Preparing variant call data with `vcf2rdf`'), building involves running the following commands:

```
autoreconf -vfi && ./configure
make && 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.

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;
- 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 3.3 '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 sequence data with `fasta2rdf`

Resources like pre-composed reference genomes are often distributed in the FASTA file format. The `fasta2rdf` program generates RDF that describes each nucleotide, its position (where the first nucleotide is at position 1, not 0), and to which sequence the nucleotide belongs.

Its main aim is to describe a sequence to allow for querying the sequence context of a variant.

3.2.1 Knowledge extracted by `fasta2rdf`

The `fasta2rdf` program extracts a nucleotide and describes it along with its position in the sequence.

In addition to the knowledge from the FASTA file, `fasta2rdf` 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.
:Sequence	rdf:type	owl:Class	`:Sequence` is used to identify a sequence within the file. This is typically a chromosome or contig
: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 `fasta2rdf`.

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

```
<http://rdf.umcutrecht.nl/fasta2rdf/14f2b609b>
    :convertedBy :vcf2rdf ;
    :filename "grch37.fasta.gz"^^xsd:string ;
    a :Origin .

sample:grch37
    :foundIn <http://rdf.umcutrecht.nl/fasta2rdf/14f2b609b3> ;
    a :Sample .

sample:CLONE_TUMOR
    :foundIn <http://rdf.umcutrecht.nl/fasta2rdf/14f2b609b3> ;
    a :Sample .
```

3.3 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.

3.3.1 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
```

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;
- Keep track of different SPARQL endpoints.

4.1 Running the web interface

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

```
sg-web
```

By default, it will be accessible on `http://localhost:5000`.

4.1.1 Configurating connections

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

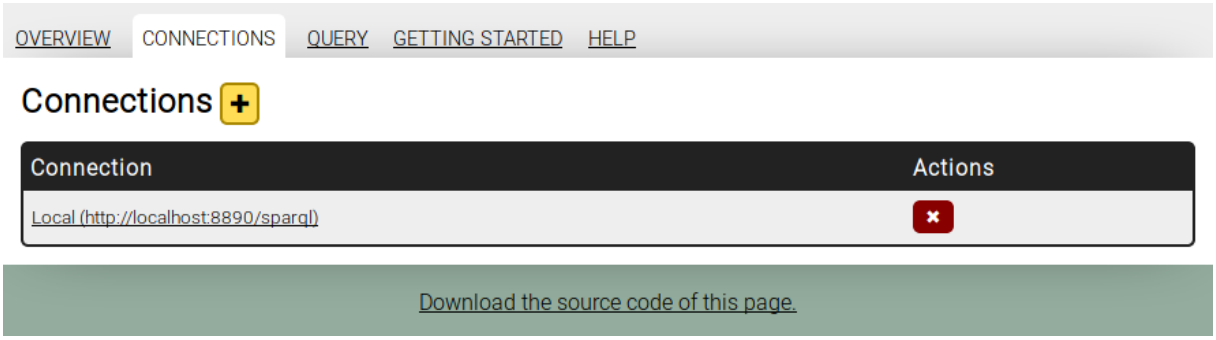

Figure 4.1: The *connections* page enables users to configure accessible SPARQL endpoints. Adding a connection here will provide an option to query it on the *query* page.

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

4.1.2 Executing queries

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

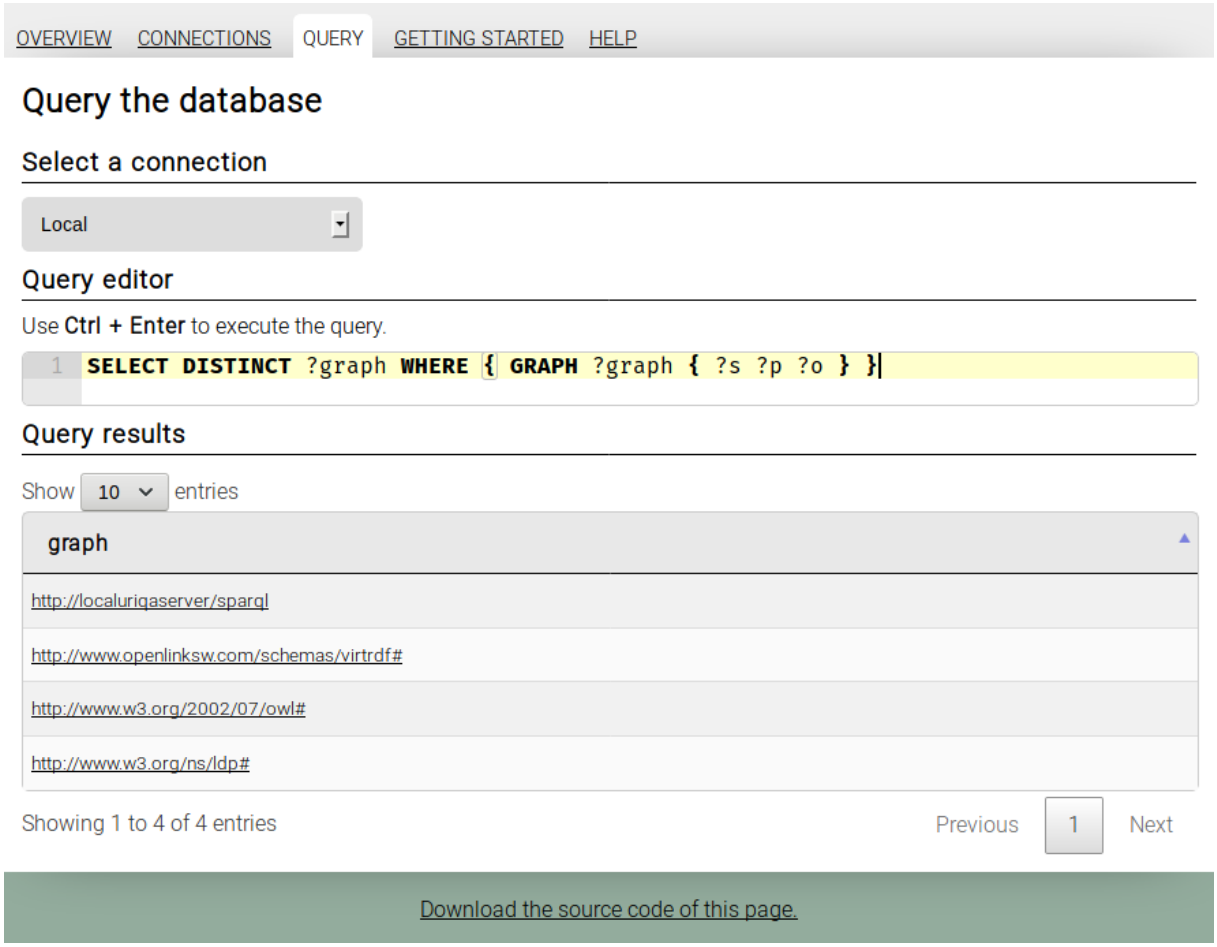

Figure 4.2: The *query* page enables users to execute a query against a SPARQL endpoint. The connections configured at the *connections* page can be chosen from the drop-down menu.

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 3.3 '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 'Programming in Python, Perl, R, Scheme, C, and/or C++'.

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

The promise from "linked data" is to make data available in such a way that one query can retrieve information from multiple SPARQL endpoints. We call querying over multiple SPARQL endpoints *federated querying*. But before we do that, let's look at simple queries that only look at our own data.

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 graphs in a SPARQL endpoint:

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

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 }
```

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. One useful method of finding out which patterns exist in a graph is to look for predicates:

```
SELECT DISTINCT ?predicate
FROM <graph-name>
WHERE {
  ?subject ?predicate ?object .
}
```

The `graph-name` must be one of the graphs returned by the query from section 5.1.1 'Listing non-empty graphs'.

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 .
}
```

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 .
}
```

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.

5.2 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
  }
}
```

Chapter 6

Programming in Python, Perl, R, Scheme, C, and/or C++

6.1 Using SPARQL with R

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

```
install.packages('SPARQL')
```

Once we're set up, we can query like so:

```
# Load the library
library('SPARQL')

# Define the endpoint to query.
endpoint <- "http://localhost:8890/sparql"

# Define the actual query to run.
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";

# Run the query
query_data <- SPARQL (endpoint, query)
```

```
# Put the results (a data frame) in a separate variable.
query_results <- query_data$results
```