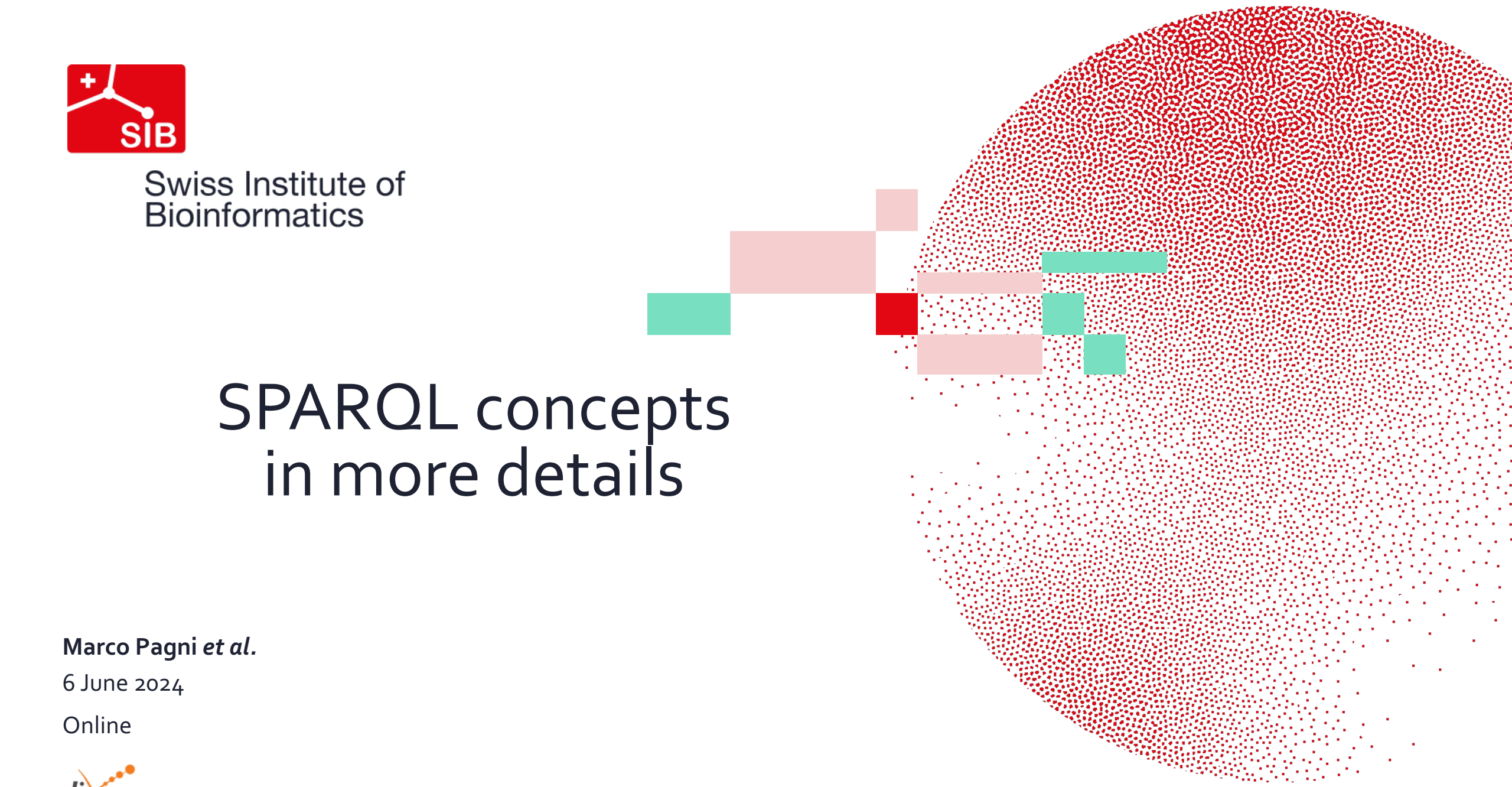


Swiss Institute of Bioinformatics

SPARQL concepts in more details

Marco Pagni *et al.*

6 June 2024

Online

elixir SWITZERLAND

RDF / SPARQL are specifications of W3C

- The World Wide Web Consortium (W3C) is the main international standards organization for the World Wide Web. Founded in 1994 and led by Tim Berners-Lee …
- W3C specifications are the foundations of the internet. They are public and open source. They are not linked to a specific implementations or a particular vendors

- Berners-Lee, Tim; Hendler, James; Lassila, Ora (May 17, 2001). "The Semantic Web". Scientific American. Vol. 284, no. 5. pp. 34–43.

- An RDF primer with many links to W3C specifications for RDF:

  https://www.w3.org/TR/rdf11-primer/

- SPARQL 1.1 specifications:

  https://www.w3.org/TR/sparql11-overview/

- RDF 1.2 and SPARQL 1.2 specifications have not yet been finalized

Specifications vs. technologies / implementations

W3C specifications for SPARQL (and RDF, RDFS) are essentially **blueprints** for how to build technologies (i.e. what constraints / requirements should these technologies satisfy)

Caution: in the “Wild Wild Web”, there are many implementations claiming to comply with W3C standards. Some don’t, some do, some are more reliable than others. (...) If there is any doubt about W3C conformance for specific implementations, the W3C specs are the “go to” ground truth resources and there are test specs.

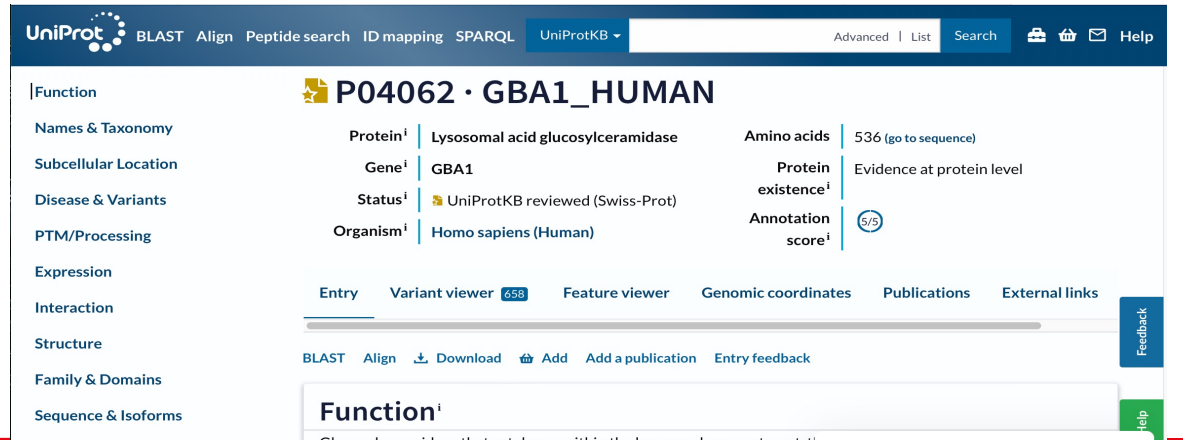
IRI - Internationalized Resource Identifier

- In the RDF world, IRIs are used as “names”, or an equivalent of “IDs”, for graph nodes.
- IRI often looks like URL, and indeed can often be used such as (this is convenient, but not mandatory)
- For example:

`http://purl.uniprot.org/uniprot/P04062`

is the legacy IRI of GBA1_HUMAN. When search in a browser, the UniprotProt server redirect it to

`https://www.uniprot.org/uniprotkb/P04062/entry`



IRI, URI, URL and URN definitions

The Internationalized Resource Identifier (**IRI**) is an internet protocol standard which builds on the Uniform Resource Identifier (**URI**) protocol by greatly expanding the set of permitted characters.

$$\{IRI\} \supset \{URI\}$$

URIs which provide a means of locating and retrieving information resources on a network (either on the Internet or on another private network, such as a computer filesystem or an Intranet) are Uniform Resource Locators (**URL**s). Other URIs provide only a unique name, without a means of locating or retrieving the resource or information about it; these are Uniform Resource Names (**URN**s).

$$\{URI\} = \{URL\} \cup \{URN\}$$

IRI, URI, URL and URN examples

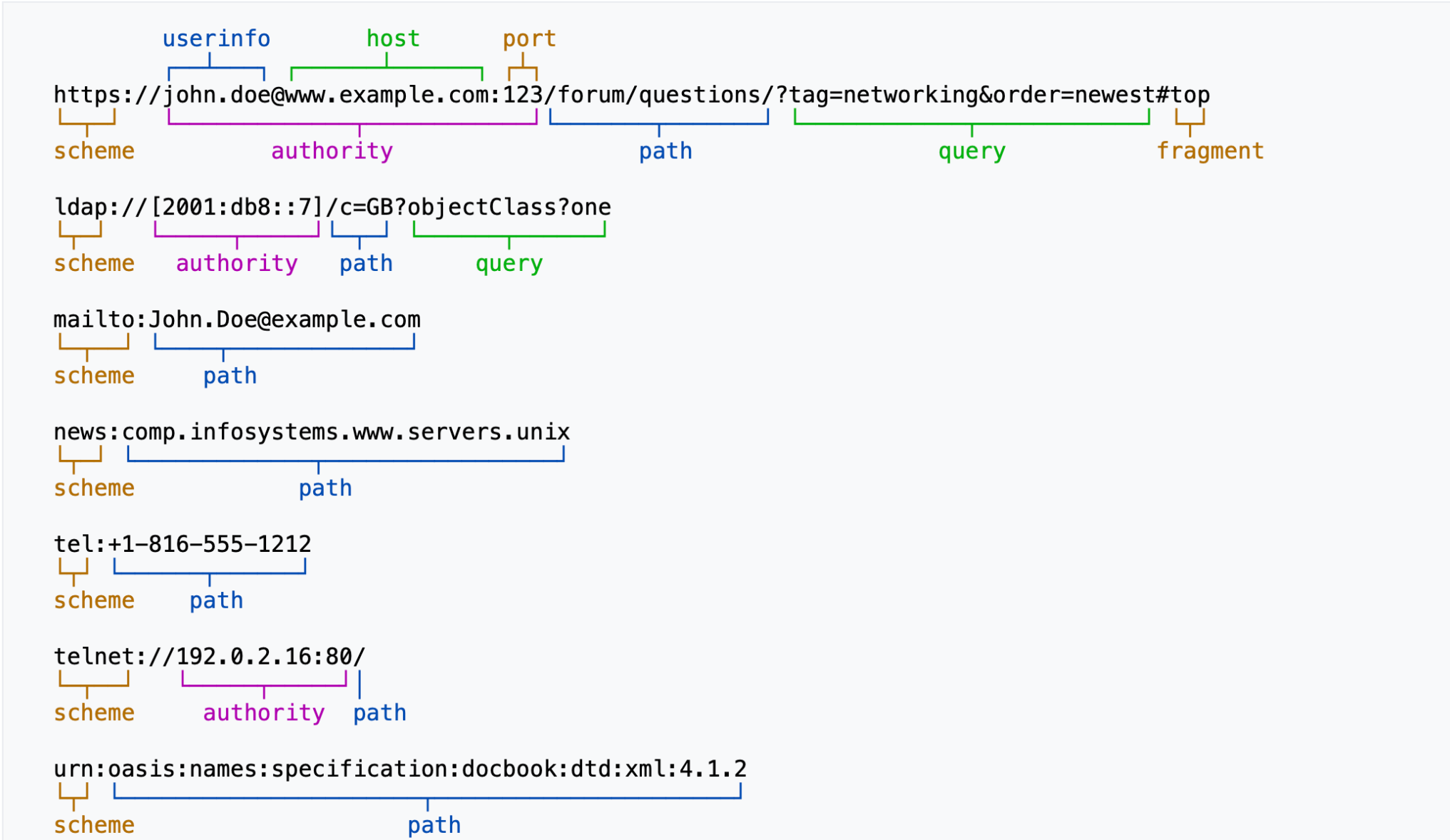
example	IRI	URI	URL	URN	comment
`https://www.uniprot.org/uniprotkb/P04062/entry`	(✓)[1]	✓	✓		SwissProt page of GBA1_HUMAN
`http://purl.uniprot.org/uniprot/P04062`	✓	✓	(✓)[2]		IRI of GBA1_HUMAN
`https://en.wiktionary.org/wiki/Πόδος`	✓	(✓)[3]	(✓)[3]		features UTF-8
`https://en.wiktionary.org/wiki/%E1%BF%AC%CF%8C%CE%B4%CE%BF%CF%82`	✓	✓	✓		the above example URL-encoded: same URL but different IRI !!!
`ISBN:0-395-36341-1`	✓	✓		✓	a book reference
`http://example.org/my_own_cat`	✓	✓		✓	http://example.org is a reserved domain name, for defining private URN

1: should not be used as an IRI in RDF
2: redirected to https://www.uniprot.org/uniprotkb/P04062/entry in browser
3: modern browser can URL-endcode it

Nota Bene about "http:" link:

- migration of URL to "https:" is recommended
- preservation of "http:" IRI ensure backward compatibility of existing RDF

URI examples from Wikipedia



Long and short forms of IRIs

In the Turtle serialization of RDF, IRI must be "quoted" using **<>:**

```
<http://purl.uniprot.org/uniprot/P04062>
```

which is known as **long-form** syntax of IRIs.

By using a prefix definition, one can rewite IRI in a **short-form** notation:

```
@prefix up: <http://purl.uniprot.org/uniprot/> .
up:P04062
```

Very important for RDF:

- The long form is the reference one. It is the only form that matters for data exchanges.
- The short form is human friendly, but
  - the prefix declaration is local to the file or client software (*i.e.* it is not publicly defined).
  - different short-form identifiers may actually refer to the same long form identifier.

PREFIXES and vocabularies

Prefix definitions are local, but there existst some generally accepted conventions for widely use vocabularies

short	long
rdf:	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
rdfs:	`http://www.w3.org/2000/01/rdf-schema#`
owl:	`http://www.w3.org/2002/07/owl#`
skos:	`http://www.w3.org/2004/02/skos/core#`
foaf:	`http://xmlns.com/foaf/0.1/`

identifiers.org

identifiers.org has attempted to define universal **short-form** identifiers:

- This is a complete heresy with respect to the semantic-web principles, for which **the long forms is the reference**
- Identifiers.org has systematically recreated new long-form identifiers (to redirect the web traffic through their servers?), ignoring previously defined IRIs.
- Identifiers.org URL have changed at least three times during the last ten years!

☠☠☠☠

- Very, very unfortunately, identifiers.org short-form identifiers have been adopted in the SBML standard ☹ ☹ ☹ ☹
- With respect to systems biology, MetaNetX is attempting to keep links between legacy stable IRIs and the most recent version of identifiers.org URLs, but this is a daunting task with no added scientific value ☹ ☹ ☹ ☹

RDF Triple

The simplest possible RDF graph is made of a single triple, for example in Turtle syntax:

```
<http://purl.uniprot.org/uniprot/P04062>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
                <http://purl.uniprot.org/uniprot/Protein>
```

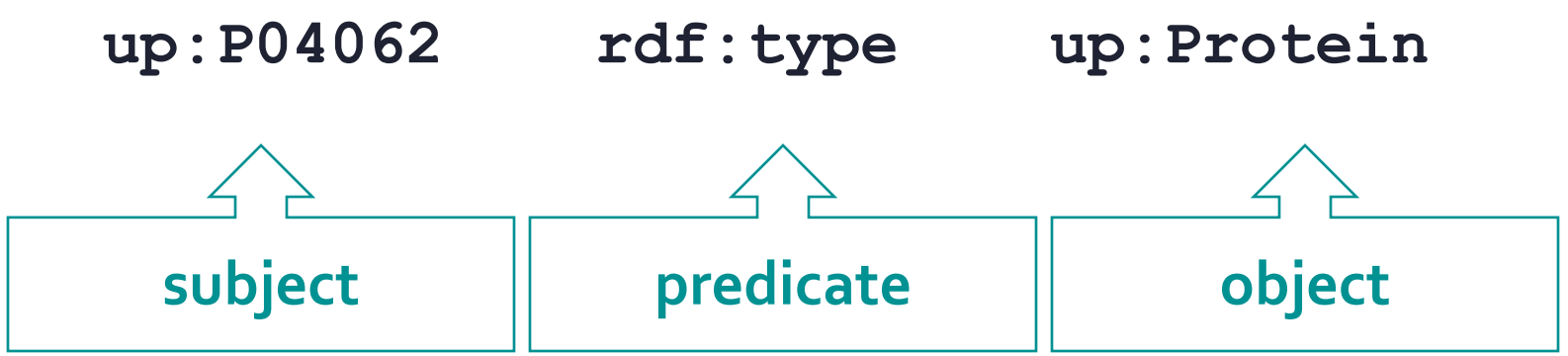
That can be rewritten using short-form notations

```
@prefix up:  <http://purl.uniprot.org/uniprot/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
up:P04062 rdf:type up:Protein
```

which can be further simplifies as Turtle support **a** as syntactic sugar for **rdf:type**

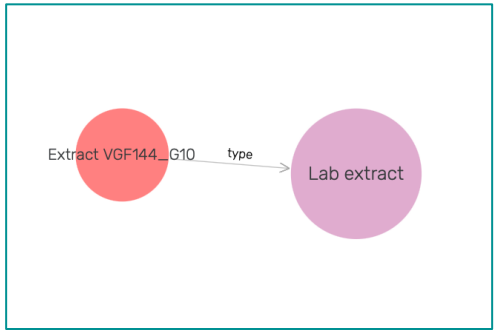
```
up:P04062 a up:Protein
```

RDF triple

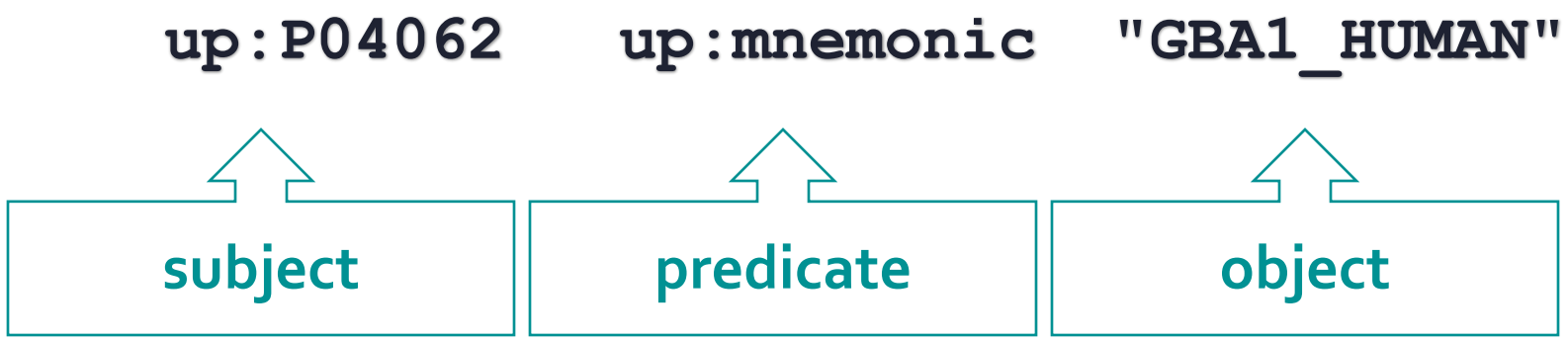


GraphDB FREE

	subject	predicate	object
1	jlw:VGF144_G10	rdf:type	jlw:LabExtract



Literal



Literals are only permitted to occur as the object position

Literal can be optionnally typed:

`"GBA1_HUMAN"^^xsd:string` is the same as `"GBA1_HUMAN"`

`"5"^^xsd:integer` is the same as `5`

`"2018-04-09T12:00:00"^^xsd:dateTime`

Language tag

String can be endowed with a language tag, *e.g*. the name of Galway in wikidata:

```
wd:Q129610 wdt:P31 "Galway"@en, "Gaillimh"@ga .
```

Show titles of wikipedia articles about Ukrainian villages on Romanian Wikipedia, plus English and Ukrainian labels in Wikidata items:

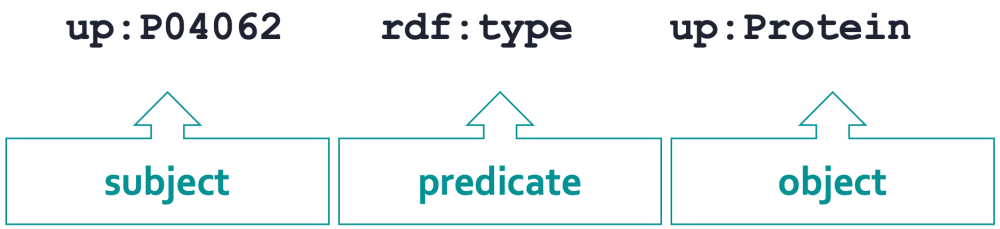
```
#added in 2017-05
SELECT DISTINCT ?item ?LabelEN ?LabelUK ?page_titleRO WHERE {
  # item: is a - village
  ?item wdt:P31 wd:Q532 .
  # item: country - Ukraine
  ?item wdt:P17 wd:Q212 .
  # exists article in item that is ro.wiki
  ?article schema:about ?item ;
           schema:isPartOf <https://ro.wikipedia.org/> ;
           schema:name ?page_titleRO .
  # wd labels
  ?item rdfs:label ?LabelEN FILTER (lang(?LabelEN) = "en") .
  ?item rdfs:label ?LabelUK FILTER (lang(?LabelUK) = "uk") .
}
```

item	LabelEN	LabelUK	page_titleRO
wd:Q100114	Ivanivka	Іванівка	Ivanivka, Karlivka
wd:Q149173	Sokyriany	Сокиряни	Secureni
wd:Q146542	Perechyn	Перечин	Perecin
wd:Q146510	Rakhiv	Рахів	Rahău
wd:Q146474	Irshava	Іршава	Iloşva
wd:Q110668	Stebnyk	Стебник	Stebnîk
wd:Q1977281	Nemyriv	Немирів	Nemîriv, Iavoriv
wd:Q650064	Semenivka	Семенівка	Semenivka, Lenine
wd:Q629892	Nizhni Otrozhki	Нижні Отрожки	Nîjni Otrojkî, Djankoi
wd:Q581689	Brazhenets'	Браженець	Brajeneț, Korostîşiv
wd:Q532838	Perekop	Перекоп	Perekop, Armeansk
wd:Q261418	Mizhhirya	Міжгір'я	Boureni, Transcarpatia
wd:Q474871	Zolote Pole	Золоте Поле	Zolote Pole (Kirovske)
wd:Q241910	Velykyi Bereznyi	Великий Березний	Velîkîi Bereznîi
wd:Q304276	Ivanivka	Іванівка	Ivanivka, Lenine

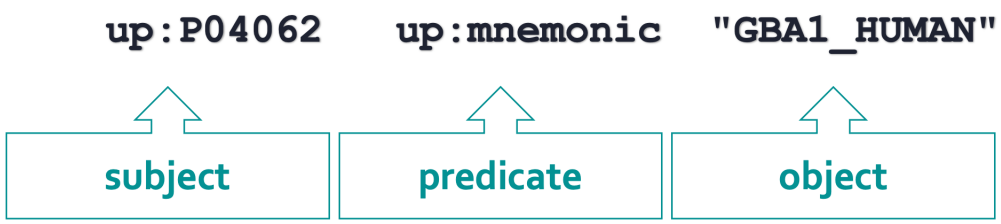
Object property and data property

A **property** is an IRI that is used as a predicate.

An **object property** is an IRI that is used as a predicate and which object is an IRI.



A **data property** is an IRI that is used as a predicate and which object is a Literal.



Blank nodes

- A blank node is an "anonymous placeholder" node in a RDF graph.
- Blank nodes have no IRI that can be used to refer to them.
- Blank nodes are allowed as subject or object of triples, exclusively.
- There is two syntaxes for blank nodes:
  - `_` is a predefined prefix for blank nodes
  - The `[]` construct can be used
- There is no clear benefits in using blank nodes in large RDF graphs. They can be safely replaced with opaque IRI.
- Blank nodes appear from time to time in SPARQL queries.

Here a snippet of Turtle to express that Anna knows someone (she unfortunately does not remember its name), who is also a friend of Bob:

```
:Anna knows _:nobody .
_:nobody :isFriendOf :Bob
```

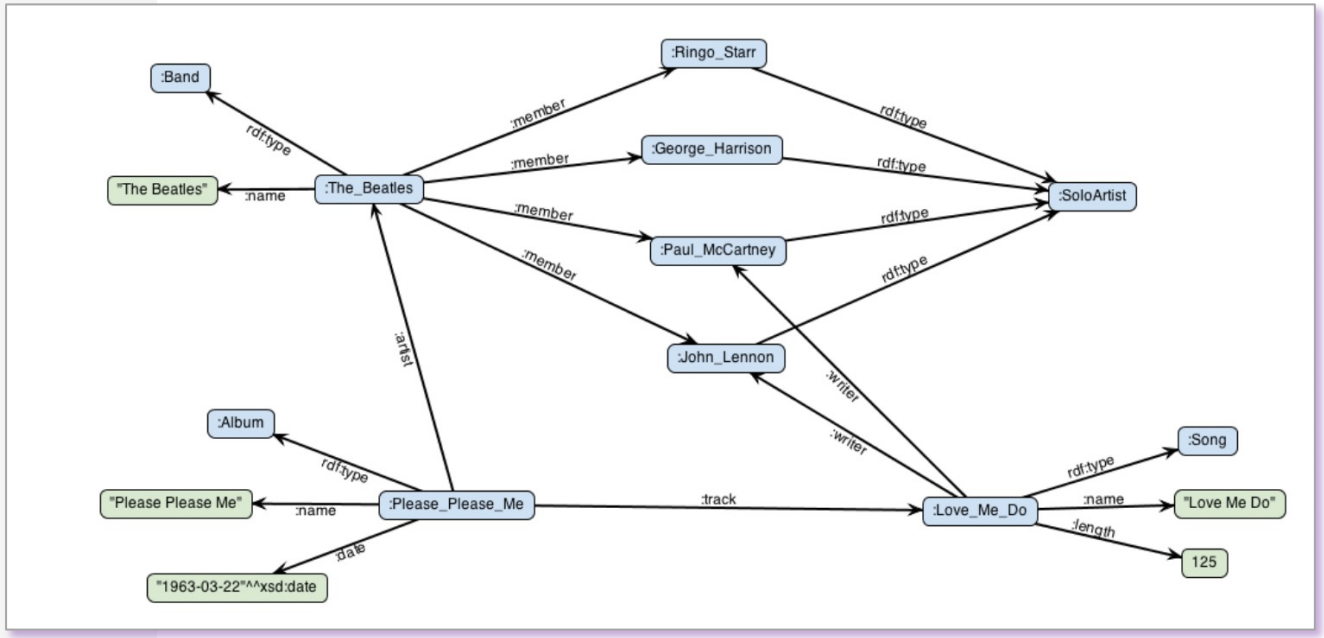
or

```
:Anna knows [ :isFriendOf :Bob ]
```

RDF graph

```
PREFIX : <http://contextualise.dev/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

:The_Beatles      rdf:type  :Band .
:The_Beatles      :name     "The Beatles" .
:The_Beatles      :member   :John_Lennon .
:The_Beatles      :member   :Paul_McCartney .
:The_Beatles      :member   :Ringo_Starr .
:The_Beatles      :member   :George_Harrison .
:John_Lennon      rdf:type  :SoloArtist .
:Paul_McCartney   rdf:type  :SoloArtist .
:Ringo_Starr      rdf:type  :SoloArtist .
:George_Harrison  rdf:type  :SoloArtist .
:Please_Please_Me rdf:type  :Album .
:Please_Please_Me :name     "Please Please Me" .
:Please_Please_Me :date     "1963-03-22"^^xsd:date .
:Please_Please_Me :artist   :The_Beatles .
:Please_Please_Me :track    :Love_Me_Do .
:Love_Me_Do       rdf:type  :Song .
:Love_Me_Do       :name     "Love Me Do" .
:Love_Me_Do       :length   125 .
:Love_Me_Do       :writer   :John_Lennon .
:Love_Me_Do       :writer   :Paul_McCartney .
```



Punctuation in Turtle syntax

```
ex:Anna a foaf:Person                    .
ex:Anna foaf:knowns ex:Bob               .
ex:Bob a foaf:Person                     .
ex:Bob foaf:mBox mail:bob@gmail.com      .
ex:Bob foaf:mBox mail:bob@github.com
```

dot is the triple separator

```
ex:Anna a foaf:Person                    ;
        foaf:knowns ex:Bob               .
ex:Bob a foaf:Person                     ;
       foaf:mBox mail:bob@gmail.com      ;
       foaf:mBox mail:bob@github.com
```

semicolon is a triple separator, with implicit subject

```
ex:Anna a foaf:Person                    ;
        foaf:knowns ex:Bob               .
ex:Bob a foaf:Person                     ;
       foaf:mBox mail:bob@gmail.com      ,
                 mail:bob@github.com
```

comma is a triple separator, with implicit subject and object

Common pitfalls in SPARQL programming

```
SELECT *
WHERE{
    ?prot_AC rdf:type up:Protein .
    ?prot_AC up:mnemonic ?prot_ID
}
```

prot_AC	prot_ID
http://purl.uniprot.org/uniprot/A0A0G2QSQ0	"A0A0G2QSQ0_9MURI"^^xsd:string
http://purl.uniprot.org/uniprot/A0A0G2YPK2	"A0A0G2YPK2_BACIU"^^xsd:string
http://purl.uniprot.org/uniprot/A0A0G2YSX8	"A0A0G2YSX8_BACIU"^^xsd:string
http://purl.uniprot.org/uniprot/A0A0G2YW19	"A0A0G2YW19_BACIU"^^xsd:string
http://purl.uniprot.org/uniprot/A0A0G2YWE2	"A0A0G2YWE2_BACIU"^^xsd:string
http://purl.uniprot.org/uniprot/A0A0G2YXG9	"A0A0G2YXG9_BACIU"^^xsd:string
http://purl.uniprot.org/uniprot/A0A0G2YY17	"A0A0G2YY17_BACIU"^^xsd:string
http://purl.uniprot.org/uniprot/A0A0G2YY83	"A0A0G2YY83_BACIU"^^xsd:string
http://purl.uniprot.org/uniprot/A0A0G3BCA7	"A0A0G3BCA7_9BURK"^^xsd:string
http://purl.uniprot.org/uniprot/A0A0G3BDG2	"A0A0G3BDG2_9BURK"^^xsd:string

424'647'286 lines

```
SELECT *
WHERE{
    ?prot_AC rdf:type up:Protein .
    ?prot_AC up:mnemonics ?prot_ID
}
```

∅

```
SELECT *
WHERE{
    ?prot_AC rdf:type up:Protein .
    ?prot_AD up:mnemonic ?prot_ID
}
```

$(424'647'286)^2 = 1.8e+17$ lines

Common pitfalls in SPARQL programming

```
SELECT *
WHERE{
     ?prot_AC rdf:type up:Protein ;
              up:mnemonic ?prot_ID
}
```

Mitigation:

- use ";" and "," punctuations
- cut-and-paste from template
- use editor auto-completion
- progressively build and run large queries
- SPARQL sub-queries help

Anatomy of a SPARQL query

```
PREFIX dbo:  <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?name ?author ?pages
WHERE {
  ?book a dbo:Book ;
    dbo:author ?author ;
    dbo:numberOfPages ?pages ;
    rdfs:label ?name .
  FILTER (?pages > 500)
  FILTER (langMATCHES(LANG(?name),"en"))
}
ORDER BY ?pages
LIMIT 10
```

Prefix declarations

Variables to display in the results

Where clause to define the basic graph pattern (BGP)
Match and filter specific triples

“Triple patterns” to match in the graph

Filter triples based on the values of some entities

Solution sequence modifiers:
Order by, group by, offset, limit clauses

A few ubiquitous predicates

predicate	object type	comment
rdf:type	class IRI	• usually abreviated with 'a' in Turtle • every IRI should belong to a class • multiple types are permitted
rdfs:label	string literal	• should be short • should have a unique value
rdfs:comment	string literal	• should be more descriptive than label
foaf:depiction	image URL	• used by graphdb-workbench to identify images
owl:sameAs	IRI	• Subject and object IRIs referer to exactly the same thing

Data, schema and documentation

In most programming languages and in relational databases, there exists a clear distinction between **data**, **schema** and **documentation**. Usually, each one has its own syntax**.**

There is no such distinction in RDF: data, schema and documentation are expressed as RDF ! In the music example, the "schema" was limited to type assignment, *e.g.*

```
:ABBA:_The_Album rdf:type :Album
```

it could be completed with a class definition and documentation:

```
:Album rdf:type rdfs:Class ;
       rdfs:comment "An album is a collection of
audio recordings (e.g., music) issued on a medium
such as compact disc (CD), vinyl (record), audio tape
(like 8-trackor cassette), or digital." .
```

Benefit: RDF is totally open for semantic innovations

Drawback: anything is possible – the distinction between data and schema is often blurred – automated validation is not part of the specifications (SHACL address this)

Important vocabularies

- **RDF/RDFS**
  - Ubiquitously used to define types, classes and properties
  - allows for basic reasoning (e.g. type inheritence through class definition)
  - https://www.emse.fr/~zimmermann/Teaching/SemWeb/rdfs.pdf
- **OWL**
  - Extension of RDF/RDFS, to build ontologies and to perform reasoning
- **SKOS**
  - A popular vocabulary to organize thesaurus and ontologies.
- **SHACL**
  - A vocabulary to validate RDF schema

https://github.com/dgarijo/Widoco

WIzard for DOCumenting Ontologies (WIDOCO)

DOI 10.5281/zenodo.10447662 JitPack 1.4.21 repo status Active

WIDOCO helps you to publish and create an enriched and customized documentation of your ontology automatically, by following a series of steps in a GUI.

Author: Daniel Garijo Verdejo (@dgarijo)

More vocabularies

https://lov.linkeddata.es/dataset/lov/

Reuse existing vocabularies… or create a new one?

Reusing existing vocabulary is a recommended practice, however it comes at the risk of semantic alteration, and this is very detrimental!

For example, consider:

```
chebi:57972 chebislash:InChI "InChI=1S/C3H7NO2/c1-2(4)3(5)6/h2H,4H2,1H3,(H,5,6)/t2-/m0/s1"
```

Does the data property `chebislash:InChI` implies that the InChI string must be a standard InChI? Does it imply a particular version of the inchi-1 software? It is impossible to answer these questions, and the ChEBI documentation lacks these details. A solution would to create our own property

```
chebi:57972 reconxkg:InChI "InChI=1S/C3H7NO2/c1-2(4)3(5)6/h2H,4H2,1H3,(H,5,6)/t2-/m0/s1"
```

and document it with:

```
reconxkg:InChI a rdf:Property ;
        rdfs:label   "has InChI" ;
        rdfs:comment "Standard InChI, computed with inchi-1 version 1.07" ;
        owl:equivalentProperty chebislash:InChI .
```

No information is lost, precisions are given and `chebislash:InChI` is referenced ;-)

SPARQL endpoints

"*A **SPARQL Endpoint** is a Point of Presence on an HTTP network that's capable of receiving and processing **SPARQL Protocol** requests*."

- It is identified by an URL commonly referred to as a SPARQL Endpoint URL.
- It expects a **`query`** parameter which value is SPARQL code (`SELECT`, `DESCRIBE`, ...)
- It can return the request results (if any) under different formats, *e.g.* `TSV`, `JSON`, `Turtle`, ...

A few SPARQL Endpoint URL:

- **`https://sparql.rhea-db.org/sparql`** (RHEA)
- **`https://sparql.uniprot.org/sparql`** (UniProt)
- **`https://reconx.vital-it.ch/graphdb/sparql`** (ReconxKG provisional public site)
- **`http://localhost:7200/repositories/ReconXKG`** (ReconxKG deployed on my local instance of GraphDB)

SPARQL federated queries, example 1

Generate a draft human metabolome *is a* RHEA demo query to be run on the RHEA SPARQL endpoint:
https://sparql.rhea-db.org/sparql

```
SELECT
  ?uniprot ?mnemonic ?rhea ?chebi ?smiles ?inchiKey
WHERE
{
  ?rhea rh:side/rh:contains/rh:compound ?compound .
  ?compound (rh:chebi|(rh:reactivePart/rh:chebi)|(rh:underlyingChebi/rh:chebi)) ?chebi .
  ?chebi chebislash:smiles ?smiles ;
         chebislash:inchikey ?inchiKey .
  SERVICE <https://sparql.uniprot.org/sparql/> {
     ?uniprot up:annotation/up:catalyticActivity/up:catalyzedReaction ?rhea ;
              up:organism taxon:9606 ;
              up:mnemonic ?mnemonic .
   }
}
```

SPARQL federated queries, example 2

Use IDSM Sachem to find ChEBIs with a a Cholestane skeleton (in SMILES). Then match returned ChEBIs to Rhea undirected reactions:

```
PREFIX sachem: <http://bioinfo.uochb.cas.cz/rdf/v1.0/sachem#>
PREFIX rh: <http://rdf.rhea-db.org/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT
  ?rhea
  ?chebi
WHERE {
    SERVICE <https://idsm.elixir-czech.cz/sparql/endpoint/chebi> {
        ?chebi sachem:substructureSearch [
             sachem:query "[C@]12(CCC3CCCC[C@]3(C)[C@@]1([H])CC[C@]1(C)[C@@]([H])([C@@](C)([H])CCCC(C)C)CC[C@@]21[H])[H]"
         ]
    }
    ?rhea rh:side/rh:contains/rh:compound/rdfs:subClassOf ?chebi
}
```

SPARQL subquery, example

The example below calculates the population of each country in the world, expressing the population as a percentage of the world's total population. In order to calculate the world's total population, it uses a subquery.

```
SELECT ?countryLabel ?population (round(?population/?worldpopulation*1000)/10 AS ?percentage)
WHERE { ¨
   ?country wdt:P31     wd:Q3624078; # is a sovereign state
                wdt:P1082 ?population.
   {    # subquery to determine ?worldpopulation
       SELECT (sum(?population) AS ?worldpopulation)
       WHERE {
           ?country wdt:P31     wd:Q3624078; # is a sovereign state
                        wdt:P1082 ?population.
       }
    }
    SERVICE wikibase:label {
       bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en".
   }
}
ORDER BY desc(?population)
```

SPARQL subqueries

- SPARQL subqueries are guaranteed to be executed first, starting from the innermost, which permits to exert some control on the query execution plan.
  - SPARQL is a declarative language, like SQL or Prolog. SPARQL **basic graph pattern** specifies a list of constraints to be satisfied by the results.
  - The database engine is responsible for establishing the execution plan, i.e. the order in which the constraints to be satisfied.
  - Sometimes the database engine takes a wrong decision: a query may never finish.

- SPARQL subqueries permits to structure large SPARQL queries and facilitate their debugging.
- SPARQL subqueries expand SPARQL capabilities by allowing the formulation of complex constrains.

Named graphs

Named graphs:

- permit to identify a set of triples with an IRI; there exists a default named graph in any triplestore.
- are known as "context" in the RDF4J/GraphDB world
- are meant to facilitate the handling of triples, i.e. they must not be used to bring additional semantics.
- SPARQL syntax provides full support for named graphs

```
SELECT *
FROM <http:example.org/toto>
WHERE{
     ?s ?p ?o
}
```

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

SPARQL CONSTRUCT …

```
SELECT ?person_1 ?person_3
WHERE{
    ?person_1 :hasSex      :Male      ;
              :hasSibling ?person_2 .
    ?person_2 :isParentOf ?person_3
}
```

```
CONSTRUCT{
         :isUncleOf a rdf:Property .
         ?person_1 :isUncleOf ?person_3
}
WHERE{
    ?person_1 :hasSex      :Male      ;
              :hasSibling ?person_2 .
    ?person_2 :isParentOf ?person_3
}
```

```
INSERT{
    GRAPH <:family_relationships> {
         :isUncleOf a rdf:Property .
         ?person_1 :isUncleOf ?person_3
    }
}
WHERE{
    ?person_1 :hasSex      :Male      ;
              :hasSibling ?person_2 .
    ?person_2 :isParentOf ?person_3
}
```

The WHERE clause stays the same ;-))

Types of SPARQL queries

SPARQL (read only)

- **ASK:** Returns a boolean answer (true/false) to specified graph pattern
- DESCRIBE
- **SELECT:** Retrieve entities matching identified variables from graph pattern
- **CONSTRUCT:** create a target graph from graph pattern

SPARQL update (read/write)

- **LOAD:** load triples from an URL and insert them into the (specified) graph.
- **INSERT:** creates triples and inserts the constructed triples into the (specified) graph.
- **DELETE:** similar structure to both CONSTRUCT and INSERT - deletes triples from the graph!
- **CLEAR**
- **…**

SPARQL update