Reminder: RDF triples

- The **RDF data model** is similar to classical conceptual modelling approaches such as **entity–relationship or class diagrams**
- it is based on the idea of making statements about resources (in particular web resources) in the form of **subject-predicate-object triples**
- **Resources** are identified by **IRIs**
- A **triple** consists of **subject**, **predicate**, and **object**
- The **subject** can be a **resource** or a **blank node**
- The **predicate** must be a **resource**
- The **object** can be a **resource**, a **blank node**, or a **literal**

Reminder: RDF literals

- Objects of triples can be **literals**

  (subjects and predicates of triples **cannot** be literals)

- **Literals** can be

  Plain, without a language tag:

  ```
  geo:berlin  geo:name  "Berlin" .
  ```

  Plain, with a language tag:

  ```
  geo:germany  geo:name  "Deutschland"@de .
  geo:germany  geo:name  "Germany"@en .
  ```

  Typed, with a IRI indicating the type:

  ```
  geo:berlin  geo:population  "3431700"^^xsd:integer .
  ```

more details at `https://www.w3.org/2007/02/turtle/primer/`

`https://www.w3.org/TR/turtle/`

Reminder: RDF blank nodes

- **Blank nodes** are **anonymous resources**
- A **blank node** can only be used as the subject or object of an RDF triple

```
_:x  a  geo:City  .
_:x  geo:containedIn  geo:germany  .
_:x  geo:name  "Berlin"  .
```

SPARQL Protocol And RDF Query Language

(pronounced as **sparkle**)

- SPARQL is a **W3C Recommendation** since 15/01/2008; uses SQL-like syntax
  SPARQL 1.1 is a **W3C Recommendation** since 21/03/2013
- SPARQL is a **query language** for RDF graphs (supported by many graph databases)
- Simple RDF graphs are used as **query patterns**
- These query graphs are represented using the Turtle syntax
- SPARQL additionally introduces **query variables** to specify parts of a query pattern that should be returned as a result
- Does not support RDFS, only RDF (SPARQL 1.1 supports RDFS entailment regime)

more details: `http://www.w3.org/TR/rdf-sparql-query/`

Tutorials: `http://www.ibm.com/developerworks/xml/library/j-sparql/`
`http://jena.apache.org/tutorials/sparql.html`

Library data in RDF

```
@prefix lib: <http://www.lib.org/schema#> .
@prefix : <http://www.bremen-lib.org/> .

:library lib:location "Bremen" .
:jlb lib:name "Jorge Luis Borges" .
:b1 lib:author :jlb ;
    lib:title "Labyrinths" .
:b2 lib:author :jlb ;
    lib:title "Doctor Brodie's Report" .
:b3 lib:author :jlb ;
    lib:title "The Garden of Forking Paths" .
:abc lib:name "Adolfo Bioy Casares" .
:b4 lib:author :abc ;
    lib:title "The Invention of Morel" .
:jc lib:name "Julio Cortázar" .
:b5 lib:author :jc ;
    lib:title "Bestiario" .
_:b6 lib:author :jc ;
    lib:title "Un tal Lucas" .
:jc lib:bornin "Brussels" .
```

SPARQL: simple query

Query over the library RDF document: find the names of authors

```
PREFIX lib: <http://www.lib.org/schema#>
SELECT ?author
WHERE
{
  ?x  lib:name  ?author  .
}
```

query variable

query pattern

variable identifier

There are three triples having the form of the query pattern:

```
:jlb  lib:name  "Jorge Luis Borges"  .
:abc  lib:name  "Adolfo Bioy Casares"  .
:jc   lib:name  "Julio Cortázar"  .
```

Answer (assignments to ?author)

author
"Jorge Luis Borges"
"Adolfo Bioy Casares"
"Julio Cortázar"

the choice of variable names is arbitrary: for example, you can use ?y in place of ?author

SPARQL: basic graph pattern

Query over the library RDF document: find the names of authors and the titles of their books

```
SELECT ?author, ?title
WHERE
{
  ?b  lib:author  ?a  .
  ?a  lib:name  ?author  .
  ?b  lib:title  ?title  .
}
```

query variables

query pattern

aka **basic graph pattern** or **BGP**

variable identifiers

Answer (assignments to ?author and ?title)

author	title
"Jorge Luis Borges"	"Labyrinths"
"Jorge Luis Borges"	"Doctor Brodie's Report"
"Jorge Luis Borges"	"The Garden of Forking Paths"
"Adolfo Bioy Casares"	"The Invention of Morel"
"Julio Cortázar"	"Bestiario"
"Julio Cortázar"	"Un tal Lucas"

variables may appear as subjects, predicates and objects of RDF triples

COUNT, LIMIT, DISTINCT

Find up to ten people whose daughter is a professor:

```
PREFIX eg: <http://example.org/>
SELECT ?parent
WHERE
{ ?parent  eg:hasDaughter  ?child  .
  ?child  eg:occupation  eg:Professor  .
}
LIMIT 10
```

Count all triples in the database: (COUNT(*) counts all results)

```
SELECT (COUNT(*) AS ?count)
WHERE { ?subject  ?predicate  ?object  . }
```

Count all predicates in the database:

```
SELECT (COUNT(DISTINCT ?predicate) AS ?count)
WHERE { ?subject  ?predicate  ?object  . }
```

The shape of a SPARQL query

SELECT queries consist of the following major blocks:

- **Prologue:** for `PREFIX` and `BASE` declarations (work as in Turtle)
- **Select clause:** `SELECT` (and possibly other keywords) followed either by a list of variables (e.g., `?person`) and variable assignments (e.g., `(COUNT(*) as ?count)`), or by * (selecting all variables)
- **Where clause:** `WHERE` followed by a pattern (many possibilities)
- **Solution set modifiers:** such as `LIMIT` or `ORDER BY`

SPARQL supports further types of queries, which primarily exchange the SELECT clause for something else:

- **ASK query:** to check whether there are results at all (without returning any)
- **CONSTRUCT query:** to build an RDF graph from query results
- **DESCRIBE query:** to get an RDF graph with additional information on each query result (application dependent)

Basic SPARQL syntax

RDF terms are written like in Turtle:

- **IRIs** may be abbreviated using `qualified:names` (requires `PREFIX` declaration) or `<relativeIRIs>` (requires `BASE` declaration)
- **Literals** are written as usual, possibly also with abbreviated datatype IRIs
- **Blank nodes** are written as usual

In addition, SPARQL supports variables:

A **variable** is a string that begins with `?` or `$`, where the string can consist of letters (including many non-Latin letters), numbers, and the symbol `_`

The **variable name** is the string after `?` or `$`, without this leading symbol.

The variables `?var1` and `$var1` have the same variable name (and same meaning across SPARQL).

Convention: Using `?` is widely preferred these days!

Basic Graph Patterns

We can now define the simplest kinds of patterns:

A **triple pattern** is a triple `s p o .` where **s** and **o** are arbitrary RDF terms or variables, and **p** is an IRI or a variable.

A **basic graph pattern** (BGP) is a set of triple patterns.

NB. These are semantic notions, which are not directly defining query syntax. Triple patterns describe query conditions where we are looking for matching triples. BGPs are interpreted conjunctively, i.e., we are looking for a match that fits all triples at once.

Syntactically, SPARQL supports an extension of Turtle (that allows variables everywhere and literals in subject positions). All Turtle shortcuts are supported.

Convention: We will also use **triple pattern** and **basic graph pattern** to refer to any (syntactic) Turtle snippet that specifies such (semantic) patterns.

Blank nodes in SPARQL

Remember: blank node (bnode) IDs are syntactic aids to allow us serialising graphs with such nodes. They are not part of the RDF graph.

What is the meaning of blank nodes in query patterns?

- They denote an unspecified resource (in particular: they do not ask for a bnode of a specific node id in the queried graph!)
- In other words: they are like **variables** but cannot be used in `SELECT`
- Turtle bnode syntax can be used (`[]` or `_:nodeId`), but any node id can only appear in one part of the query (we will see complex queries with many parts later)

What is the meaning of blank nodes in query results?

- Such bnodes indicate that a variable was matched to a bnode in the data
- The same node id may occur in multiple rows of the result table, meaning that the same bnode was matched
- However, the node id used in the result is an auxiliary id that might be different from what was used in the data (if an id was used there at all!)

Blank nodes in SPARQL (cont.)

– There is no reason to use blank nodes in a query:
you can get the same functionality using variables

```
SELECT ?a ?b
WHERE
{
  ?a  :predicate  _:blanknode  .
  _:blanknode  :otherPredicate  ?b  .
}
```

=

```
SELECT ?a ?b
WHERE
{
  ?a  :predicate  ?variable  .
  ?variable  :otherPredicate  ?b  .
}
```

Blank node example

Data

```
_:a   foaf:name   "Alice"   .
_:b   foaf:name   "Bob"   .
```

SPARQL query

```
SELECT ?x ?name
WHERE
{
?x   foaf:name   ?name   .
}
```

Answer

x	name
_:c	"Alice"
_:d	"Bob"

Answers to BGPs

What is the result of a SPARQL query?

A **solution mapping** is a partial function μ from variable names to RDF terms.

A **solution sequence** is a list of solution mappings.

NB. When no specific order is required, the solutions computed for a SPARQL query can be represented by a **multiset**

(= 'a set with repeated elements' = 'an unordered list').

Given an RDF graph G and a BGP P, a solution mapping μ is a **solution to P over G** if it is defined exactly on the variable names in P and there is a mapping σ from blank nodes to RDF terms such that $\mu(\sigma(P)) \subseteq G$.

The cardinality of μ in the multiset of solutions is the number of distinct such mappings σ. The multiset of these solutions is denoted by $\text{eval}_G(P)$, where we omit G if clear from the context

NB. Here, we write $\mu(\sigma(P))$ to denote the graph given by the triples in P after first replacing bnodes according to σ, and then replacing variables according to μ.

Example 1

```
eg:Arrival  eg:actorRole  eg:aux1, eg:aux2 .
eg:aux1  eg:actor  eg:Adams  ;  eg:character  "Louise Banks" .
eg:aux2  eg:actor  eg:Renner ;  eg:character  "Ian Donnelly" .
eg:Gravity  eg:actorRole  [ eg:actor  eg:Bullock;
                            eg:character  "Ryan Stone" ] .
```

The BGP `?film eg:actorRole []` has the solution multiset:

film	cardinality
`eg:Arrival`	2
`eg:Gravity`	1

The cardinality of the first solution mapping is 2 since the bnode can be mapped to two resources, `eg:aux1` and `eg:aux2`, to find a subgraph.

Example 2

```
eg:Arrival  eg:actorRole  eg:aux1, eg:aux2 .
eg:aux1  eg:actor  eg:Adams  ;  eg:character  "Louise Banks" .
eg:aux2  eg:actor  eg:Renner ;  eg:character  "Ian Donnelly" .
eg:Gravity  eg:actorRole  [ eg:actor  eg:Bullock;
                            eg:character  "Ryan Stone" ] .
```

The BGP `?film eg:actorRole [ eg:actor ?person ]`

has the solution multiset:

film	*person*	cardinality
`eg:Arrival`	`eg:Adams`	1
`eg:Arrival`	`eg:Renner`	1
`eg:Gravity`	`eg:Bullock`	1

GROUP, ORDER, FILTER

Find the person with most friends:

```
SELECT ?person (COUNT(*) AS ?friendCount)
WHERE
{ ?person  <http://example.org/hasFriend>  ?friend  . }
GROUP BY ?person
ORDER BY DESC(?friendCount) LIMIT 1
```

The GROUP BY clause allows aggregation over one or more properties

(partition results into groups based on the expression(s) in the GROUP BY clause)

The ORDER BY clause establishes the order of a solution sequence

Find pairs of siblings:

```
SELECT ?child1 ?child2
WHERE
{ ?parent <http://example.org/hasChild> ?child1, ?child2  .
FILTER (?child1 != ?child2)
}
```

SELECT clauses

SELECT clauses

- specify the bindings that get returned
- may define additional results computed by functions
- may define additional results computed by aggregates

Find cities and their population densities:

```
SELECT ?city (?population/?area AS ?populationDensity)
WHERE {
  ?city  rdf:type  eg:city  ;
         eg:population  ?population  ;
         eg:areaInSqkm  ?area  .
}
```

The keyword `DISTINCT` can be used after `SELECT` to remove duplicate solutions

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ( CONCAT(?G, " ", ?S) AS ?name )
WHERE { ?P foaf:givenName ?G ; foaf:surname ?S }
```

what are the results?

Solution set modifiers

SPARQL supports several expressions after the query's WHERE clause:

- `ORDER BY` defines the desired order of results
  - Can be followed by several expressions (separated by space)
  - May use order modifiers `ASC()` (default) or `DESC()`
- `LIMIT` defines a maximal number of results
- `OFFSET` specifies the index of the first result within the list of all results

NB. Both `LIMIT` and `OFFSET` should only be used on explicitly ordered results

In Wikidata, find the largest German cities, rank 6 to 15: (see Wikidata identifiers)

```
SELECT ?city ?population
WHERE {
?city   wdt:P31   wd:Q515   ;    # instance of city
        wdt:P17   wd:Q183   ;    # country Germany
        wdt:P1082   ?population   .    # get population
} ORDER BY DESC(?population) OFFSET 5 LIMIT 10
```

OPTIONAL

Get the names of authors (in the dataset on page 5) and also
the places where they were born, if this information is available

```
SELECT ?author, ?birthplace
WHERE
{
  ?x  lib:name   ?author  .
  OPTIONAL { ?x  lib:bornin  ?birthplace }
}
```

optional pattern

Answer

author	birthplace
"Jorge Luis Borges"	
"Adolfo Bioy Casares"	
"Julio Cortázar"	"Brussels"

because the triple pattern for birthplace is **optional**, there is a pattern solution for
the authors who do not have information about their birthplace.
Without OPTIONAL, there would be only one solution: "Julio Cortázar" "Brussels"

UNION

an RDF graph containing information about people's names from FOAF and vCard

```
prefix  foaf:  <http://xmlns.com/foaf/0.1/>  .
prefix  vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>  .

_:a  foaf:name  "Matt Jones"  .
_:b  foaf:name  "Sarah Jones"  .
_:c  vcard:FN  "Becky Smith"  .
_:d  vcard:FN  "John Smith"  .
```

a SPARQL query that retrieves the names regardless of the format:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name
WHERE
{
  { [ ]  foaf:name  ?name  . } UNION { [ ]  vcard:FN  ?name  . }
}
```

or { ?x foaf:name ?name . } UNION { ?y vcard:FN ?name . }

Answer

name
"Matt Jones"
"Sarah Jones"
"Becky Smith"
"John Smith"

MINUS

(A query part within { } is called a **group graph pattern** in SPARQL)

The `MINUS` operator allows us to remove the results of one group graph pattern from the results of another

In Wikidata, find living people who are composers by occupation:

```
SELECT ?person
WHERE {
{ ?person   wdt:P106   wd:Q36834   . }    # ?person occupation: composer
MINUS
{ ?person   wdt:P570   [ ]   . }    # ?person date of death: some value
}
```

Similar results can often be achieved with `FILTER NOT EXISTS`, but the two are used differently:

`MINUS` and `FILTER NOT EXISTS` behave differently,
e.g., when applied to a group graph patterns that do not share any variables.

Testing For the Absence/Presence of a Pattern

Data

```
@prefix : <http://example/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

:alice rdf:type foaf:Person .
:alice foaf:name "Alice" .
:bob rdf:type foaf:Person .
```

Query 1

```
SELECT ?person
WHERE
{ ?person rdf:type foaf:Person .
FILTER NOT EXISTS { ?person foaf:name ?name . }
}
```

Answer

person
<http://example/bob>

Query 2

```
SELECT ?person
WHERE
{ ?person rdf:type foaf:Person .
FILTER EXISTS { ?person foaf:name ?name . }
}
```

Answer

person
<http://example/alice>

Filters

Data

```
:book1  dc:title  "SPARQL Tutorial"  .
:book1  ns:price  42  .
:book2  dc:title  "The Semantic Web"  .
:book2  ns:price  23  .
```

SPARQL query that retrieves the titles of books whose price is less than 30.5

```
SELECT ?title ?price
WHERE
{
?x  ns:price  ?price  .
FILTER (?price < 30.5)
?x  dc:title  ?title  .
}
```

Answer

title	price
"The Semantic Web"	23

Available filters:

- logical: && || !
- maths: + − * /
- SPARQL tests: isURI, isBlank, isLiteral, bound
- ...

Optional and filters

What does the following query mean?

```
SELECT ?person ?spouse
WHERE {
?person   wdt:P106   wd:Q36834   ;    # ?person occupation: composer
          wdt:P569   ?bd   .    # ?person date of birth: ?bd
OPTIONAL {
?person   wdt:P26   ?spouse   .    # ?person spouse: ?spouse
?spouse   wdt:P569   ?bd2   .    # ?spouse date of birth: ?bd2
FILTER (year(?bd)=year(?bd2))    # born in same year
}
}
```

‘Composers, and, optionally, their spouses that were born in the same year.’

Subqueries

Subqueries allow us to use results of queries within queries, typically to achieve results that cannot be accomplished using other patterns.

In Wikidata, find universities located in one of the 15 largest German cities:

```
SELECT DISTINCT ?university ?city
WHERE {
{ SELECT DISTINCT ?city ?population
  WHERE { ?city   wdt:P31/wdt:P279*   wd:Q515   ;    # instance of: city
                  wdt:P17   wd:Q183   ;    # country: Germany
                  wdt:P1082   ?population   .    # population: ?population
  } ORDER BY DESC(?population) LIMIT 15    # get top 15 by ?population
}
?university   wdt:P31/wdt:P279*   wd:Q3918   ;    # instance of: university
              wdt:P131+   ?city   .    # located in+: ?city
}
```

(the meaning of `property paths' * and + will be explained later)

Bound variables

SPARQL query to return the URIs that identify cities of type 'Cities in Texas' and their total population in descending order (i.e., bigger cities first)

Only those cities that do not have a metro population will be returned

At most 10 results will be returned

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>

SELECT *
WHERE
{
  ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> .
  ?city dbp:populationTotal ?popTotal .
  OPTIONAL {?city dbp:populationMetro ?popMetro .}
  FILTER(! bound(?popMetro))
}
ORDERED BY desc(?popTotal)
LIMIT 10
```

bound(var) evaluates to TRUE iff var is bound to some value

Aggregate functions

Data

```
:org1  :affiliates  :auth1  .
:org1  :affiliates  :auth2  .
:auth1  :writesBook  :book1  .
:auth1  :writesBook  :book2  .
:book1  :price  9  .
:book2  :price  5  .
:auth2  :writesBook  :book3  .
:book3  :price  7  .
:org2  :affiliates  :auth3  .
:auth3  :writesBook  :book4  .
:book4  :price  7  .
```

Answer

org	totalPrice
:org1	21

SPARQL query

```
SELECT ?org SUM(?lprice) AS ?totalPrice
WHERE
{
?org  :affiliates  ?auth  .
?auth  :writesBook  ?book  .
?book  :price  ?lprice  .
}
GROUP BY ?org
HAVING (SUM(?lprice) > 10)
```

Find the total price of books written by authors affiliated with some organisation:
output the organisation id and total price only if the total price is > 10

aggregate functions: COUNT, SUM, MIN, MAX, AVG

Property paths

– Find the name of any person that Alice knows:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
?x  foaf:mbox  <mailto:alice@example>  .
?x  foaf:knows/foaf:name  ?name  .
}
```

– Find the names of people two "foaf:knows" links away

```
SELECT ?name
WHERE {
?x  foaf:mbox  <mailto:alice@example>  .
?x  foaf:knows/foaf:knows/foaf:name  ?name  .
}
```

Exercise: rewrite these queries without using /

Property paths (cont.)

– Find all the people :x connects to via the foaf:knows relationship (using a path of an arbitrary length)

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?person
WHERE
{ :x   foaf:knows+   ?person   . }
```

+ means 'one or more occurrences'

– Find all types, including supertypes, of each resource in the dataset

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?x ?type
WHERE
{ ?x   rdf:type/rdfs:subClassOf*   ?type   . }
```

/ denotes sequence, *** means 'zero or more occurrences'**

CONSTRUCT

SELECT creates a **table** with the assignments to the selected variables

SELECT * selects **all** variables in the query

Keyword CONSTRUCT returns a set of triples (that is, an RDF graph)

```
1  PREFIX lib: <http://www.lib.org/schema>
2  CONSTRUCT {
3    ?b lib:author ?a .
4    ?a lib:name ?author .
5    ?b lib:title ?title .
6  }
7  WHERE {
8    ?b lib:author ?a .
9    ?a lib:name ?author .
10   ?b lib:title ?title .
11   FILTER regex(?author, "^Julio")
12 }
```

Answer

RDF triples
:jc lib:name "Julio Cortázar" .
:b5 lib:author :jc .
:b5 lib:title "Bestiario" .
:b6 lib:author :jc .
:b5 lib:title "Un tal Lucas" .

Keyword FILTER imposes additional restrictions on queries:

?author should begin with Julio

Exercise

What does the following query construct?

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX vcard:   <http://www.w3.org/2001/vcard-rdf/3.0#>
CONSTRUCT   { <http://example.org/person#Alice> vcard:FN ?name }
WHERE       { ?x foaf:name ?name }
```

The answer to this query over the RDF graph

```
@prefix  foaf:  <http://xmlns.com/foaf/0.1/> .
_:a    foaf:name   "Alice" .
_:a    foaf:mbox   <mailto:alice@example.org> .
```

is the RDF graph

```
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
<http://example.org/person#Alice> vcard:FN "Alice" .
```

SPARQL endpoint

- A SPARQL endpoint enables users (human or other) to query a knowledge base via SPARQL
- Results are typically returned in one or more machine-processable formats.
- Therefore, a SPARQL endpoint is mostly conceived as a machine-friendly interface towards a knowledge base.
- Both the formulation of the queries and the human-readable presentation of the results should typically be implemented by the calling software
- Several Linked Data sets exposed via SPARQL endpoint: send your query, receive the result!

  DBpedia and Wikidata
  Musicbrainz
  World Factbook
  LinkedMDB
  DBLP ...