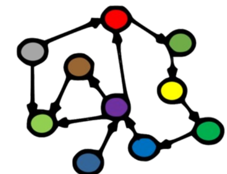
Welcome to INFO216:
Knowledge Graphs

Spring 2022

Andreas L Opdahl
<Andreas.Opdahl@uib.no>

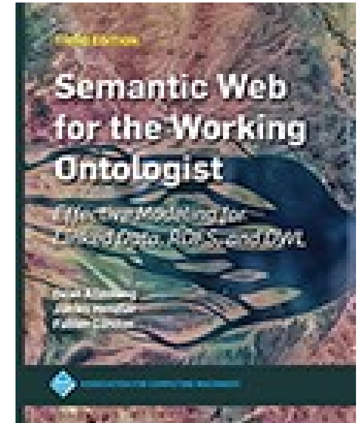
Session 3: Querying and updating KGs (SPARQL)

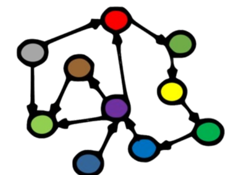
- Themes:
  - SPARQL queries
    - SPARQL 1.1
  - SPARQL Update
  - SPARQL 1.1 Update Language
  - Programming SPARQL queries and updates



Readings

- Sources:
  - Allemang, Hendler & Gandon (2020): Semantic Web for the Working Ontologist, 3rd edition chapter 6 (SPARQL, but chapter 5 in the 2nd edition)
  - Blumauer & Nagy (2020): Knowledge Graph Cookbook – Recipes that Work (for example around pages 54-55 and 133)
  - W3C resources:
    - SPARQL 1.1 Cheat Sheet
    - http://www.w3.org/TR/sparql11-query/
    - http://www.w3.org/TR/sparql11-update/
  - RDFLib documentation
  - *As always: wiki.uib.no/info216*





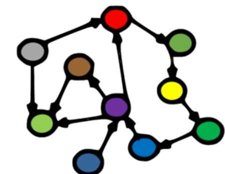
SPARQL

SPARQL queries

- SPARQL Protocol and RDF Query Language
- Example:

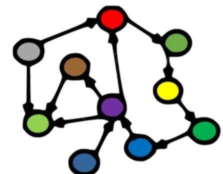
```
SELECT ?child ?sister WHERE {
    ?child fam:hasParent ?parent .
    ?parent fam:hasSister ?sister .
}
```

- **Main idea:**
  - give SPARQL an underspecified RDF graph: a **pattern**...
  - some of the nodes are **variables**:
    - other nodes are **IRIs**, or **literal** values as before
  - SPARQL tries to **match** the pattern to an RDF graph
  - ...and returns each match as a **result**



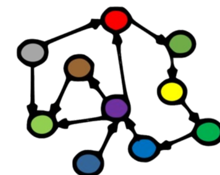
SPARQL organisation

- SPARQL 1.1 comprises:
  - *Query Language* – the core of SPARQL
  - result formats, including: XML, JSON, CSV…
  - federated queries (involving several *endpoints*)
  - *SPARQL Update Language* – since 1.1 *(previously called "SPARUL")*
  - service descriptions and protocols
    - *queries and results over HTTP*
  - test cases
- RDF and SPARQL offer a *common abstraction level* for data interoperability

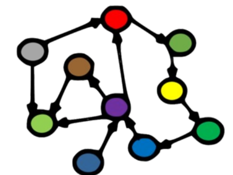


SPARQL queries and updates

- Four types of dataset *queries*:
  - SELECT: returns table
  - ASK: returns yes/no
  - CONSTRUCT: returns a graph
  - DESCRIBE: returns a graph
- *Two groups of manipulations:*
  - *graph store updates:*
    - DELETE, INSERT, *LOAD, CLEAR*
  - *graph store management:*
    - *CREATE, DROP, COPY, MOVE, ADD*
- *All are written in TURTLE-like style*
  - *...the variations are similar too*

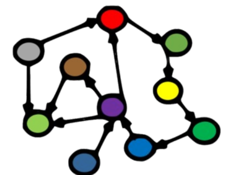


SPARQL queries



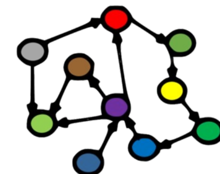
The next slides are intended as "schemas" to provide a quick overview of stuff you already know something about...

The links to W3C documents in the wiki provide lots of examples!



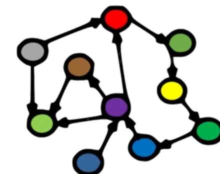
Example KG

- A knowledge graph of research literature on "Knowledge Graphs for the News"
  - built to support an ongoing literature study
  - 78 main papers with 291 authors
  - 4086 other papers with 8990 authors
  - 100s of topics and themes, >300k triples
- Available at sandbox.i2s.uib.no (UiB internal, need VPN):
  - a Blazegraph triple store (RDF database)
  - very simple web front end, read only
- Online sandbox:
  - sandbox.i2s.uib.no – web front end + SPARQL endpoint
  - query *and update* – use namespaces!



SELECT queries: Basic forms

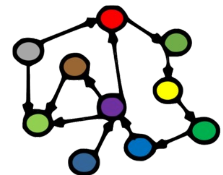
- Basic form:
  - **SELECT** *projection* **WHERE {** *pattern* **}**
  - the *projection* is a list of *variables*
  - the *pattern* is (essentially) a list of *triples*
  - returns a table with *one row per result* and *one column per projection variable*
- Optional and combinable variations:
  - SELECT ***** WHERE { ... }
  - SELECT **DISTINCT** ... WHERE { ... }
  - SELECT ... WHERE { ... } **LIMIT *n***
  - SELECT ... WHERE { ... } **LIMIT *n* OFFSET *m***
  - SELECT ... WHERE { ... } **ORDER BY** *...variable...*
  - SELECT ... WHERE { ... } **ORDER BY DESC(***...variable...***)**



SELECT: Grouping

- Grouping of results:
  - SELECT *...grouping or aggregate variables...* WHERE { ... } **GROUP BY** *...grouping variables...* **[HAVING ...]**
  - example: counting students in courses

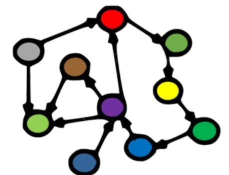
```
SELECT COUNT(?student) WHERE {
	?student ex:takes ?course .
}
```



SELECT: Grouping

- Grouping of results:
  - SELECT *...grouping or aggregate variables...* WHERE { ... } **GROUP BY** *...grouping variables...* **[HAVING ...]**
  - example: counting students in courses

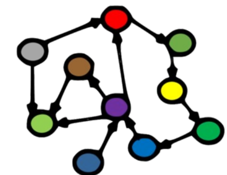
```
SELECT (COUNT(?student) AS ?count) WHERE {
      ?student ex:takes ?course .
}
```



SELECT: Grouping

- Grouping of results:
  - SELECT *...grouping or aggregate variables...* WHERE { ... } **GROUP BY** *...grouping variables...* **[HAVING ...]**
  - example: counting students in courses

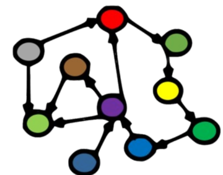
```
SELECT ?course (COUNT(?student) AS ?count) WHERE {
      ?student ex:takes ?course .
}
GROUP BY ?course
```



SELECT: Grouping

- Grouping of results:
  - SELECT *...grouping or aggregate variables...* WHERE { ... } **GROUP BY** *...grouping variables...* **[HAVING ...]**
  - example: counting students in courses

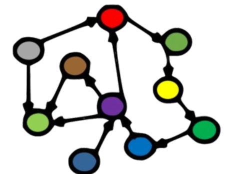
```
SELECT ?course (COUNT(?student) AS ?count) WHERE {
      ?student ex:takes ?course .
}
GROUP BY ?course
 HAVING (?count >= 10)
```



SELECT: Grouping

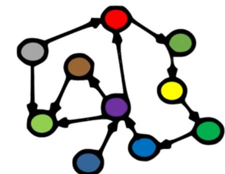
- Grouping of results:
  - SELECT *...grouping or aggregate variables...* WHERE { ... } **GROUP BY** *...grouping variables...* **[HAVING ...]**
  - example: counting students in courses

```
SELECT ?course (COUNT(?student) AS ?count) WHERE {
      ?student ex:takes ?course .
}
GROUP BY ?course
 HAVING (?count >= 10)
 ORDER BY DESC(?count)
```



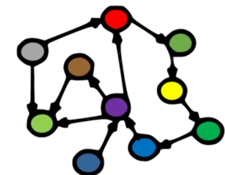
SELECT: Grouping

- Grouping of results:
  - SELECT *...grouping or aggregate variables...* WHERE { ... } **GROUP BY** *...grouping variables...* **[HAVING (...)]**
  - *grouping variables:*
    - regular variables
    - used to group the rows in the results
  - *aggregate variables:*
    - the results of aggregate functions: SUM(), COUNT(), MIN(), MAX(), AVG(), GROUP_CONCAT(), SAMPLE()



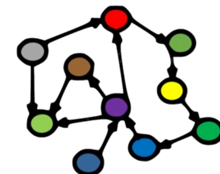
SELECT: Bindings

- In the list of projection variables:
  - SELECT **(SUM (...) AS ...)** ... WHERE { ... }
- In the graph pattern:
  - SELECT ... WHERE { ... **BIND(... AS ...)** ... }
- In the GROUP BY variable list
  - SELECT ... WHERE { ... }
    GROUP BY ... **(... AS ...)** ...



SELECT: Composite patterns

- WHERE-variants with multiple *pattern groups*:
  - ... WHERE { **{ ... } UNION { ... }** }
  - ... WHERE { ... **OPTIONAL { ... }** }
  - ... WHERE { ... **MINUS { ... }** }
  - ... WHERE { ... **FILTER (...)** }
  - ... WHERE { ... **FILTER [NOT] EXISTS { ... }** }
- *Filters* are *logical expressions*:
  - standard logic operators: !, &&, ||
  - (in-)equality operators: =, !=, <, <=, >, >=
  - arithmetic: +, -, /. *
  - built-in, imported *(xsd:...)* and self-defined functions

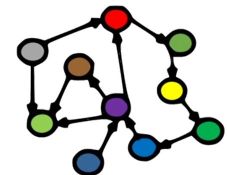


(c) Andreas L Opdahl, 2022

SELECT: Built-in functions

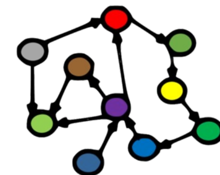
- Examples of built-ins:
  - bound, if
  - exists, not exists
  - in, not in
  - IRI, bnode
  - isIRI, isBlank, isLiteral, isNumeric
  - str, lang, strlang ("for language tagged literals"@en)
  - regex, strlen, contains, substr... (from XPath)
  - replace
  - abs, rand, ceil, floor
  - now, year, month, days, hours, minutes, seconds

https://en.wikibooks.org/wiki/SPARQL/Expressions_and_Functions



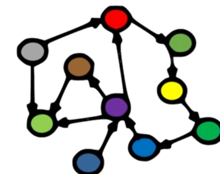
SELECT queries: Naming graphs

- SELECT ... WHERE { ... }
  - matching a pattern from the default graph in the dataset
- SELECT ... WHERE { ... **GRAPH <iri> { ... }** . ... }
  - matching a pattern from the *named graph* <*iri*> in the dataset
- SELECT ... **FROM <irl1>** WHERE { ... }
- SELECT ... **FROM <irl1> ... <irlN>** WHERE { ... }
  - the *union* of named graphs <*irl1*>...<*irlN*> becomes a *default graph* to be matched by the WHERE-pattern
- SELECT ... **FROM NAMED <irl>**
  WHERE { ... **GRAPH <irl> { ... }** . ... }
  - the *named graph* <*irl*> becomes a *named graph* matched by a part of the WHERE-pattern



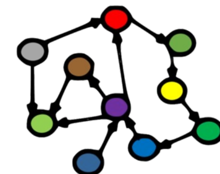
SELECT: Federated queries

- Local queries to non-default graphs:
  SELECT ... WHERE { ... **GRAPH <iri> { ... }** . ... }
- Nested queries:
  SELECT ... WHERE { ... **{ SELECT ... WHERE { ... } }** }
- Remote queries:
  SELECT ... WHERE { ... **SERVICE <irl> { ... }** }
  SELECT ... WHERE { ... **SERVICE SILENT <irl> { ... }** }
- In-line values:
  SELECT ... WHERE {
  ...
  VALUES *var* { *value1 value2 ... valuen* }
  ...
  }
  var and ***valuen*** can be tuples ***(v1, v2, … vm)***



SELECT: Composite properties

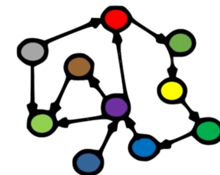
- *Property paths (in SPARQL 1.1)*:
  - Concatenation: ***a / b*** (means "first ***a***, then ***a***'s ***b***")
  - Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
  - Alternative: ***a | b*** (means "***a*** or ***b***")
  - Inversion: ***^a*** (means "***a*** backwards")
  - Negation: ***!a*** (means "any other predicate than ***a***")
  - Grouping: ***(...)*** (nested composite properties)



SELECT: Composite properties

- *Property paths (in SPARQL 1.1)*:
- Concatenation: ***a / b*** (means "first ***a***, then ***a***'s ***b***")
  - example:
    ?nephew :hasParent ?parent .
    ?parent :hasBrother ?uncle .
  - can be written as:
    ?nephew :hasParent / :hasBrother ?uncle .

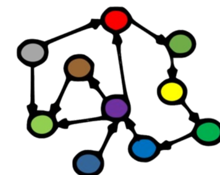
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means "***a*** or ***b***")
- Inversion: ***^a*** (means "***a*** backwards")
- Negation: ***!a*** (means "any other predicate than ***a***")
- Grouping: ***(...)*** (nested composite properties)



SELECT: Composite properties

- *Property paths (in SPARQL 1.1)*:
- Concatenation: ***a / b*** (means "first ***a***, then ***a***'s ***b***")
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
  - example:
    ?person :hasParent* ?ancestor .
    ?person :hasParent+ ?ancestor .
    ?person :hasSibling? ?sibling .

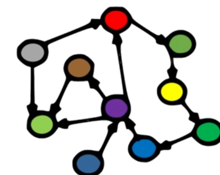
- Alternative: ***a | b*** (means "***a*** or ***b***")
- Inversion: ***^a*** (means "***a*** backwards")
- Negation: ***!a*** (means "any other predicate than ***a***")
- Grouping: ***(...)*** (nested composite properties)



SELECT: Composite properties

- *Property paths (in SPARQL 1.1)*:
- Concatenation: ***a / b*** (means "first ***a***, then ***a***'s ***b***")
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
  - these two pattern-lines always match:
    ?anyResource :anyProperty* ?anyResource .
    ?anyResource :anyProperty? ?anyResource .
    - for any ?anyResource
    - and any :anyProperty

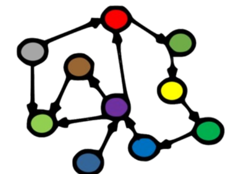
- Alternative: ***a | b*** (means "***a*** or ***b***")
- Inversion: ***^a*** (means "***a*** backwards")
- Negation: ***!a*** (means "any other predicate than ***a***")
- Grouping: ***(...)*** (nested composite properties)



SELECT: Composite properties

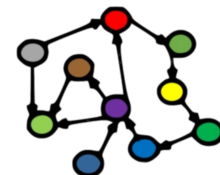
- *Property paths (in SPARQL 1.1)*:
- Concatenation: ***a / b*** (first *a*, then *a*'s *b*)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means ***a*** or ***b***)
  - example:
    ?child :hasFather | :hasMother ?parent .
    ?person :hasBrother | :hasSister ?sibling .

- Inversion: ***^a*** (means ***a*** backwards)
- Negation: ***!a*** (any other predicate than a)
- Grouping: ***(...)*** (nested composite properties)



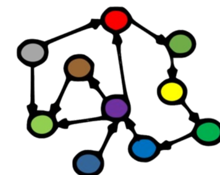
SELECT: Composite properties

- *Property paths (in SPARQL 1.1)*:
- Concatenation: ***a / b*** (first *a*, then *a*'s *b*)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means ***a*** or ***b***)
- Inversion: ***^a*** (means ***a*** backwards)
  - example:
    ?parent ^:hasParent ?child .
- Negation: ***!a*** (any other predicate than a)
- Grouping: ***(...)*** (nested composite properties)



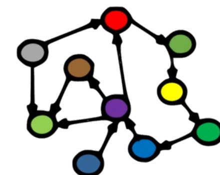
SELECT: Composite properties

- *Property paths (in SPARQL 1.1)*:
- Concatenation: ***a / b*** (first *a*, then *a*'s *b*)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means ***a*** or ***b***)
- Inversion: ***^a*** (means ***a*** backwards)
- Negation: ***!a*** (any other predicate than a)
- Grouping: ***(...)*** (nested composite properties)
  - example:
    ?parent ^(:hasFather | :hasMother) ?child .



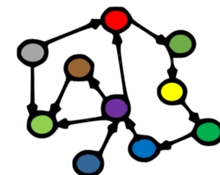
SELECT: Composite properties

- *Property paths (in SPARQL 1.1)*:
- Concatenation: ***a / b*** (first *a*, then *a*'s *b*)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means ***a*** or ***b***)
- Inversion: ***^a*** (means ***a*** backwards)
- Negation: ***!a*** (any other predicate than a)
- Grouping: ***(...)*** (nested composite properties)
  - example:
    ?parent ^(:hasFather | :hasMother) ?child .
    ?parent (^:hasFather | ^:hasMother) ?child .



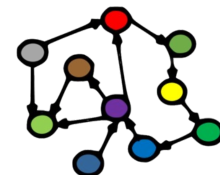
SELECT: Composite properties

- *Property paths (in SPARQL 1.1)*:
- Concatenation: ***a / b*** (first *a*, then *a*'s *b*)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means ***a*** or ***b***)
- Inversion: ***^a*** (means ***a*** backwards)
- Negation: ***!a*** (any other predicate than a)
- Grouping: ***(...)*** (nested composite properties)
  - example:
    ?uncle ^(:hasParent / :hasBrother) ?nephew .
    ?uncle ?nephew .



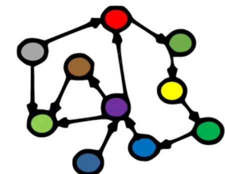
SELECT: Composite properties

- *Property paths (in SPARQL 1.1)*:
- Concatenation: ***a / b*** (first *a*, then *a*'s *b*)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means ***a*** or ***b***)
- Inversion: ***^a*** (means ***a*** backwards)
- Negation: ***!a*** (any other predicate than a)
- Grouping: ***(...)*** (nested composite properties)
  - example:
    ?uncle ^(:hasParent / :hasBrother) ?nephew .
    ?uncle (^:hasBrother / ^:hasParent) ?nephew .

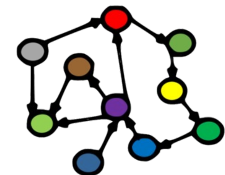


The other query types

- ASK { ... }
  - *can the pattern be matched, yes or no?*
- CONSTRUCT { *?s1 ?p1 ?o1 . ?s2 ?p2 ?o2 . ...* } WHERE { ... }
  - *returns triples*, e.g., for copying a graph:
    CONSTRUCT { ?s ?p ?o }
    WHERE { GRAPH <iri> { ?s ?p ?o } . }
- DESCRIBE ?resource
  - *returns a "relevant excerpt" of the graph for ?resource*
  - not well defined: *all triples where a resource is subject? all triples where a resource is subject or object? concise bounded descriptions (CBDs)? symmetric CBDs?*
- Most variations of SELECT can also be used for ASK/CONSTRUCT/DESCRIBE when they give meaning!

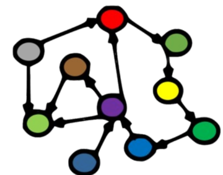


Programming SPARQL in RDFLib and SPARQLWrapper



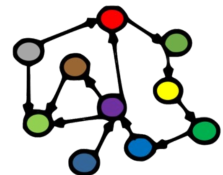
SPARQL Query in RDFLib

```
import rdflib
g = rdflib.Graph()
g.parse("family.ttl", format='ttl')
qres = g.query("""
    PREFIX fam: <http://example.org/family#>
    SELECT ?child ?sister WHERE {
        ?child fam:hasParent ?parent .
        ?parent fam:hasSister ?sister .
    }""")
for row in qres:
    print("%s has aunt %s" % row)
```



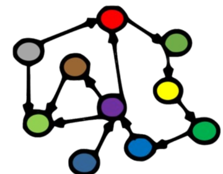
SPARQL Query in RDFLib

```
q = rdflib.plugins.sparql.prepareQuery(
        """SELECT ?child ?sister WHERE {
                ?child fam:hasParent ?parent .
                ?parent fam:hasSister ?sister .
        }""",
      initNs = { "fam": "http://example.org/family#"})

sm = rdflib.URIRef("http://example.org/royal#SverreMagnus")

for row in g.query(q, initBindings={'child': sm}):
        print(row)
```

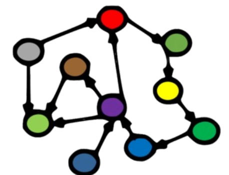


Remote SPARQL Query with SPARQLWrapper

```
from SPARQLWrapper import SPARQLWrapper

endpoint = 'http://sandbox.i2s.uib.no/bigdata/namespace/kb/sparql'
client = SPARQLWrapper(endpoint)
client.setReturnFormat('json')
client.setQuery("""
    SELECT DISTINCT ?t WHERE {
        ?r rdf:type ?t .
    }
""")
res = client.queryAndConvert()
print(res)
```

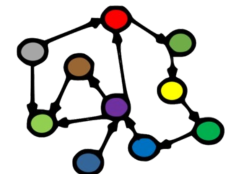


SPARQL Update



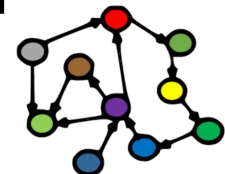
Management of graph stores

- CREATE GRAPH <http://example.org/testGraph>
  CREATE SILENT GRAPH <http://example.org/testGraph>
  - creates a new empty graph with the specified name in the graph store
- DROP GRAPH <http://example.org/testGraph>
  DROP DEFAULT
  DROP NAMED
  DROP ALL
  - removes the specified graph(s) from the graph store



Management of graph stores

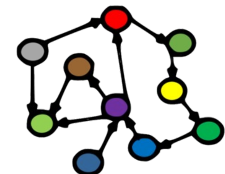
- COPY <http://example.org/testGraph> TO DEFAULT
  COPY DEFAULT TO <http://example.org/otherGraph>
  COPY <http://example.org/otherGraph> TO
  <http://example.org/yetAnotherGraph>
  - inserts all data from the source into the destination graph
  - source graph not affected
  - destination graph emptied before insertion
- MOVE <http://example.org/testGraph> TO DEFAULT
  - like COPY, but source graph is removed afterwards
- ADD <http://example.org/testGraph> TO DEFAULT
  - like COPY, but destination graph is not emptied beforehand



Updates: LOAD and CLEAR

- Read an RDF document from an IRL:
  LOAD <file:/home/andreas/myGraph.ttl>
  LOAD <file:/home/andreas/myGraph.ttl> INTO GRAPH
  <http://example/org/testGraph>

- Remove all triples from a graph in the graph store:
  CLEAR GRAPH <http://example/org/testGraph>



Updates: INSERT DATA

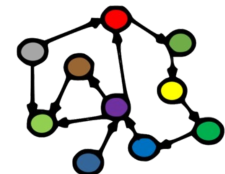
- Create new triples, e.g.:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA
{      <http://example/book3>     dc:title    "A new book" ;
                                  dc:creator  "A.N.Other" .
}
```

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
INSERT DATA
{      GRAPH <http://example/bookStore>
            { <http://example/book1>  ns:price  42 }
}
```

A *dataset* contains one *default graph* and zero or more *named graphs*

- *Suitable for creating small data sets...*

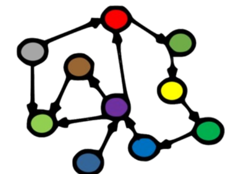


Updates: DELETE DATA

- Remove given triples, e.g.:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
DELETE DATA
{
        <http://example/book2>    dc:title "David Copperfield" ;
                                  dc:creator "Edmund Wells" .
}

PREFIX dc: <http://purl.org/dc/elements/1.1/>
DELETE DATA
{
   GRAPH <http://example/bookStore> {
          <http://example/book2> dc:title "David Copperfield" ;
                                 dc:creator "Edmund Wells" .
   }
}
```

A *dataset* contains one *default graph* and zero or more *named graphs*



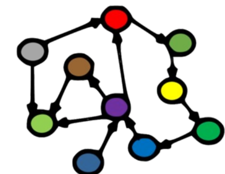
Updates: DELETE/INSERT from pattern

- Remove and create triples from patterns, e.g.:

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>

WITH <http://example/addresses>
DELETE { ?person foaf:givenName 'Bill' }
INSERT { ?person foaf:givenName 'William' }
WHERE {
      ?person foaf:givenName 'Bill'
}
```

A *dataset* contains one *default graph* and zero or more *named graphs*

- The *optional* WITH specifies which *named graph* to modify

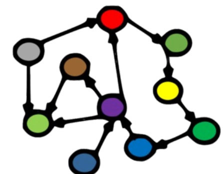


Updates: DELETE from pattern

- Create triples from pattern, e.g.:

```
PREFIX dc:  <http://purl.org/dc/elements/1.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

WITH <http://example/bookStore>
DELETE {
	?book ?p ?v
} WHERE {
	?book dc:date ?date .
	FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )
	?book ?p ?v
}
```

A *dataset* contains one *default graph* and zero or more *named graphs*

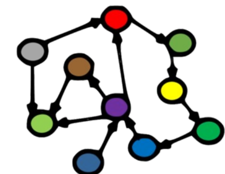


Updates: INSERT from pattern

- Create triples from pattern, e.g.:

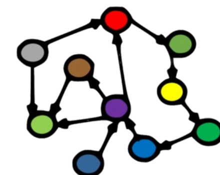
```
PREFIX dc:  <http://purl.org/dc/elements/1.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

INSERT {
        GRAPH <http://example/bookStore2> {
              ?book ?p ?v
        }
} WHERE {
        GRAPH  <http://example/bookStore> {
              ?book dc:date ?date .
        FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )
        ?book ?p ?v
        }
}
```

A *dataset* contains one *default graph* and zero or more *named graphs*

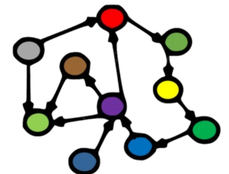


Semantic graphs and data sets

- *Graph*:
  - a collection of *triples/statements* (possibly none)
- *Data set (or "Conjunctive graph")*:
  - a collection of graphs (at least one)
  - one of the graphs is *default/unnamed*
  - the others are *named*
  - from triples/statements:
    - *(subject, predicate, object)*
  - to quadruples *(quads):*
    - *(graph/"context", subject, predicate, object)*

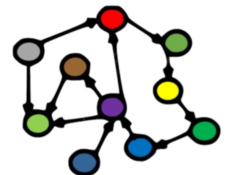


Programming SPARQL Update in RDFLib and SPARQLWrapper



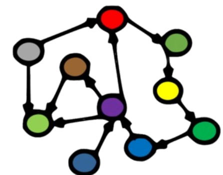
SPARQL Update in RDFLib

```
import rdflib

g = rdflib.Graph()
g.parse("family.ttl", format='ttl')  # format='turtle' also works
g.update('''
    INSERT { ?child fam:hasAunt ?sister . } WHERE {
        ?child fam:hasParent ?parent .
        ?parent fam:hasSister ?sister .
    }''',
    initNs={
        'fam': 'http://example.org/family#'
    })
```



Remote SPARQL Update with SPARQLWrapper

```
from SPARQLWrapper import SPARQLWrapper

endpoint = 'http://sandbox.i2s.uib.no/bigdata/namespace/kb/sparql'
client = SPARQLWrapper(endpoint)
client.setMethod('POST')
client.setQuery('''
    INSERT DATA {
        <http://ex.org/Abra> <http://ex.org/ka> <http://ex.org/Dabra> .
    }
''')
res = client.queryAndConvert()
```



Next week:
Storing and sharing KGs