Recall (cf. Slide 37)
The Relational Model is a specialization of First-Order Logic:

- no function symbols. Only constant literals.
- entity-type relations:
  collect data about instances of a certain entity type,
  property names given by the attributes of the relation schema
- relationship-type relations:
  represent instances (binary or \( n \)-ary) relationship types,
  relationship name given by the table name

THE SEMANTIC WEB DATA AND KNOWLEDGE MODEL

Combination of several concepts:

- Data Model: RDF (Resource Description Framework)
  - simple logical data model
  - things: resources; Web aspect: Web-wide Uniform Resource Identifiers
  - statements express properties: subject-predicate-object
- Metadata: RDF Schema
  - conceptual model: classes, subclasses and properties
  - classes and properties are also resources and can be described.
    (i.e., “second-order”)
  - descriptions about classes and properties allow to draw conclusions for their
    instances.
- database-style format + XML representation
- so far: more database-style than knowledge-base or ontology
- OWL: higher-level conceptual model
  based on description logic (a fragment of FOL), “sliced” in complexity levels
Design and Use of a Domain Ontology

Design

- define the concepts (using RDFS): names of classes, class hierarchy, properties
- define a schema for URIs
- all people then use this schema and these names

Use

- make statements with the given vocabulary about things identified by the URIs

Global Semantics

- ... just collect all statements.
- since all use the same URIs things will easily fit together.
⇒ The Web as a global data source
- query evaluation?
- incompleteness, inconsistencies, junk & fakes.

RDF: Resource Description Framework

RDF is another specialization of First-Order Logic without function symbols.

Extension of the ideas of conceptual modeling (ER-Model, UML) and a restricted subset of first-order logic:

- (globally) agreed identifiers of elements of the domain as constants,
- literal constants,
- concepts (as unary predicates, e.g., Country(\textit{germany}); usually with capital first letter),
- properties as binary relationships, e.g., capital(\text{\textit{germany}, \textit{berlin}}), or (\text{\textit{germany}, capital, \textit{berlin}}); usually with non-capital first letter.

Model: RDF Graph + derived knowledge

- derived knowledge: by RDFS and OWL built-in notions
4.1 RDF: Idea and Overview

- first Working Draft: 1997
- triple-based “RDF statements”: *(subject, predicate, object)*
- graphical representation (⇒ RDF graph), “N3” Unicode representation, XML representation
- *subject* and *object* are *resources* (cf. terminology for XLink)
  - the object can also be a literal value (of an XML Schema datatype).
- a resource is everything ... that can be described by a URI
  - a Web page, a book, a lecture ...
  - the resource identifiers are usually denoted with *<...>* to distinguish them from simple strings

```
<http://dbis.uni-goe.de/teaching/semweb/>
```

```
dc:creator
```

```
dc:subject
```

```
http://user.uni-goe.de/may ???

“Semantic Web”
```

Example: Dublin Core

“Dublin Core”: a set of properties for describing (HTML Web) documents (defined at a metadata workshop in Dublin/Ohio 1995).

- Title: A name given to the resource.
- Creator: An entity primarily responsible for making the content of the resource.
- Subject: The topic of the content of the resource.
- Description: An account of the content of the resource.
- Publisher: An entity responsible for making the resource available.
- Contributor: An entity responsible for making contributions to the content of the resource.
- Date
- Type: The nature or genre of the content of the resource.
- Language: A language of the intellectual content of the resource.

etc ... see <http://dublincore.org/documents/dces/>
RDF MODEL AND TRIPLES

- Initial goal: semantic description of things on the Web (i.e., Web pages and parts of them) author, keywords, annotation of links
goal: semantics-driven Web indexing and search engines

- Information about real (Web) resources (Triples, Statements):
  (<http://dbis.uni-goe.de> <dc:creator> <http://user.uni-goe.de/may> ???)
  (<http://dbis.uni-goe.de> <dc:subject> “Database Group”)
  (<http://dbis.uni-goe.de/teaching/semweb/> <dc:type> “Lecture”)

- The Semantic Web intention was not only to anote things, but also to relate them:
  (<http://dbis.uni-goe.de> <uni:offers-lecture> <http://dbis.uni-goe.de/teaching/semweb/> (<http://dbis.uni-goe.de> <dc:linksTo> <http://dbis.uni-goe.de/teaching/semweb/>)

- This requires relationships from other domains.
- some “things” are also not real HTTP Web pages (e.g., persons)
  (the above “early” examples mix Web pages, strings, and persons or lectures in a dirty way)

⇒ generic notion of resources
  (note: properties are also resources)
Things that are not “real” resources that have a URL (e.g. in HTTP) get a virtual *Unified Resource Identifier (URI)*, e.g.:

```
(http://user.cs.uni-goe.de/~may> dc:creator> de:person-D-12345678>)
```

and can then be described:

```
(de:person-D-12345678> uni:position> uni:Professor>)
(de:person-D-12345678> bla:lives> geo://country-de/city-goettingen>)
(de:person-D-12345678> bla:e-mail> mailto:may@cs.uni-goe.de>)
```

**RDF for Data Integration**

- information contained in the triples is independent from where they are located, and from their order.
- RDF files (knowledge bases) in different places that use the same URIs for “things” and the same “names” for properties can be easily combined.
- these “names” are taken from *domain ontologies*.

---

**URI according to RFC 2396**

- a URI can be a locator (URL), a “name” (URN), or a “general” URI.
  General form: `<scheme:hierarchical_part>`
- URIs: a sequence of characters; often with a hierarchical structure
- URLs are those URIs that identify resources via their physical access mechanism (i.e., http, ftp, gopher, file, ...), in general their `schema` part names the protocol.
  `<http://www.informatik.uni-goettingen.de/people.html>`
  `<mailto:may@informatik.uni-goettingen.de>`
  `<news:de.comp.text.tex>`
- URI schemes can be “registered” at `<http://www.IANA.org>` (Internet Assigned Numbers Authority).
  example (non-registered): `<isbn:3-89722-153-5>`
- URIs serve as agreed *identifiers* in a certain community
  some of them are valid URLs, some of them not.

RDF uses general URIs, not only URLs.
**Semantics of URIs and URLs**

URIs: mainly just identifiers, can be URLs

**“Typical” Objects**

- Objects are resources. Their URIs can be URLs or purely virtual.
- Applications that use them load some given RDF files that describe them and evaluate these files.

**Properties and Classes**

- They are also resources (i.e., not the name string like “City”, but some URI like `<geo://country-de/city-goettingen>`  
  (note for later: classes do not have “names”, since any user-defined properties must only be applied to first-order objects)
- For querying, their URIs are just identifiers
- For an application, there can be an actual description of a notion at the place that is identified by the URL,
- Then, an application can find this information.

---

**Syntax of URIs and URLs**

- General form: `scheme:hierarchical_part`
- Short form: when only a single document is used, `#name` is sufficient – it is expanded to `url-of-the-document#name`.
- All representations (N3, RDF/XML) allow to use prefixes (cf. XML Namespaces) that hide the actual schema, e.g.,
  - Declare `prefix monmeta: <http://www.semwebtech.org/mondial/10/meta#>`
  - Prefix `mon: <http://www.semwebtech.org/mondial/10/>`
  - Use `[monmeta:Country]` as class name  
    (stands for `<http://www.semwebtech.org/mondial/10/meta#Country>`)  
  - Use `[monmeta:capital]` as property name
  - Use `<http://www.semwebtech.org/mondial/10/countries/D>` as URI for Germany
  - Use `<http://www.semwebtech.org/mondial/10/countries/D/provinces/Berlin/cities/Berlin>` as URI for Berlin
- There is no commitment that these URLs actually exist or what can be found there!
URIS AS WEB-WIDE IDENTIFIERS

• The identifiers carry non-logical semantics as “names” throughout the (Semantic) Web.
• URIs are agreed in the same way as property names by the domain ontology designer. Real URLs and virtual URIs can be arbitrarily mixed.
• URIs to be used Web-wide can be defined freely
  – referring to the URL of the file where they are defined (this becomes more concrete with “Linked Open Data (LOD)” (since ~2010); cf. Slides 270 ff.).
  – defining a URI that is completely different from the file’s URL
• all members of a “community” can describe the resources in RDF.

RDF REPRESENTATIONS AND NOTATIONS

• as triple “database”: “N-triple” (next section)
• graphical: as a graph (illustrated in the next section)
• an RDF/XML representation that represents the RDF data, used for data exchange (for every RDF database there are multiple RDF/XML serializations)
• note that although RDF data looks like a large table with three columns, it cannot be stored directly in SQL: the subject and object column must hold URIs, strings and numbers.
  – similarity with vertically partitioned storage of relational databases (cf. Slide 37): each predicate is stored in a binary relation.
### Tools

**The W3C RDF Validator**

- [http://www.w3.org/RDF/Validator/](http://www.w3.org/RDF/Validator/)
- input: RDF/XML
- output: Graph and/or triples

**A lightweight JENA-based locally installed tool**

- JENA ([http://jena.sourceforge.net](http://jena.sourceforge.net)) is a Java-based Semantic Web Framework that supports RDF/RDFS, several types of OWL reasoning, and the SPARQL language; optionally an underlying relational database system can be used.
- see Web page of the lecture for further information.

### 4.2 The “N3” RDF Notation

- **Triple-“Statements”**  
  subj pred obj.
  (Alternatives: subj has pred obj . or obj is pred of subj .)
- URIs: as `<uri>`
- Literals may occur as objects: as numbers (e.g. 42) or strings (e.g., “John Doe”).
- Short form for \( x \ p \ y_1 \) and \( x \ p \ y_2 \): \( x \ p \ y_1 ; y_2 \).

  `<#john> <#child> <#alice>, <#bob> .`

- Short form for \( x \ p_1 \ x_1 \) and \( x \ p_2 \ x_2 \): \( x \ p_1 \ y_1 ; p_2 \ y_2 \).

  `<#alice> <#age> 10; <#name> “Alice” .`
EXAMPLE: N3

- example: use most simple “URI”s

```
<family:john> <family:name> "John"; <family:age> 35;
    <family:hasChild> <family:alice>, <family:bob> .
<family:alice> <family:name> "Alice"; <family:age> 10 .
<family:bob> <family:name> "Bob"; <family:age> 8 .
```

[Filename: RDF/family.n3]

Query language: SPARQL (details later)

```
select ?X ?N
from <file:family.n3>
where {?X <family:name> ?N}
```

[Filename: RDF/family.sparql]

- user interface of the local Jena-based tool: see Web page here: jena -q -qf family.sparql
- more on the use of URIs later.

LITERALS IN RDF

- Literals are strings (“John”) and numbers (42, 3.1415, 1.23E26)

```
select ?X ?N ?A
from <file:family.n3>
where {?X <family:name> ?N . ?X <family:age> ?A }
```

[Filename: RDF/family-age.sparql]

- Literals can be associated with XML Schema Datatypes xsd:int (numbers), xsd:decimal (with dec. point), xsd:double (with exponent/mantissa).
- full syntax "string"^datatype-url
  e.g. "42"^<http://www.w3.org/2001/XMLSchema#int>
  (note: the "..." must also be present for numeric datatypes)
- String literals can be associated with languages, e.g. “München”@DE, “Munich”@EN, “Monaco”@IT.
URIs as Identifiers across RDF Files

- URIs: allow for “referencing” things across RDF files
- Simplest “full” form: `<schema:name>`; typical for examples:

  ```n3
  <family:john> <family:name> "John"; <family:age> 35;
  <family:hasChild> <family:alice>, <family:bob> .
  <family:alice> <family:name> "Alice"; <family:age> 10 .
  <family:bob> <family:name> "Bob"; <family:age> 8 .
  [Filename: RDF/family.n3]
  
  <family:mary> <family:name> "Mary"; <family:age> 32;
  <family:married> <family:john>;
  <family:hasChild> <family:alice>, <family:bob> .
  [Filename: RDF/family2.n3]
  
  select ?X ?C ?A
  from <file:family.n3>
  from <file:family2.n3>
  where {?X <family:hasChild> ?C . ?C <family:age> ?A }
  [Filename: RDF/family-both.sparql]
  ```

URIs: Local Identifiers

- If only a local part of the URI is given, it is by default extended with the document URL (yielding a `local` uri):

  ```n3
  <#john> <#name> "John"; <#age> 35; <#child> <#alice>, <#bob> .
  <#alice> <#name> "Alice"; <#age> 10 .
  <#bob> <#name> "Bob"; <#age> 8 .
  [Filename: RDF/john-local.n3]
  
  select ?X ?Y ?N
  from <file:john-local.n3>
  where {?X ?Y ?N}
  [Filename: RDF/john-local.sparql]
  
  Result (among others):
  
  X / <file:///homewap1/may/teaching/SemWeb/RDF/john-local.n3#alice>
  Y / <file:///homewap1/may/teaching/SemWeb/RDF/john-local.n3#name>
  N / "Alice"
  ```
URIs: Local Identifiers in Queries

- files with local URIs can be queried via HTTP
- combine the URI of the file + local part

```sparql
select ?X ?N
from <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3>
where {?X <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3#name> ?N}
```

[Filename: RDF/john-http.sparql]

- change the default expansion: use “base” for a relative addressing in the query:

```sparql
base <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3>
select ?X ?Y
from <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3>
where {?X <#name> ?Y}
```

[Filename: RDF/john-base.sparql]

- result e.g.,
  
  X/ <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3#bob>, Y/ “Bob”

- Note: “base” is used in the application (here: SPARQL), not in N3 (although, RDF/XML has xml:base!)

Local URIs: Adding Statements to “Remote” Scopes

Even the use of local URIs cannot prevent others (by other RDF files) from adding statements to that scope:

```n3
<http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3#mary>
<http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3#married>
<http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3#john>.
```

[Filename: RDF/mary-remote.n3]

```sparql
base <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3>
select ?X ?Y ?A
from <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3>
from <file:mary-remote.n3>
where {?X <#married> ?Y . ?Y <#age> ?A}
```

[Filename: RDF/mary-http.sparql]

- note that adding resources or facts to things in “foreign” scopes is also a danger in RDF and in the Semantic Web.

⇒ “Trust” is an important keyword for SW applications.
URIs in the Semantic Web

Base URIs are often used for distinguishing ontologies

- e.g., `<http://purl.org/dc/elements/1.1/>` is the base URL for the Dublin Core Ontology that defines notions for annotating documents with authors etc.
- `<http://www.semwebtech.org/mondial/10/>` could be a base URI for defining the notions used in Mondial
  - entity types (Country, City) and property names (has-capital) as e.g. `.../10/meta#Country`
  - recall that entity instances are also resources that have a URI in this scope:
    - e.g. `.../10/countries/D/provinces/Berlin/cities/Berlin`
  - see later for details how to design this.
- ... first continue with syntax.

N3 Prefix Notation

define @prefix pre: <uri>.
and then use expressions pre:qname

These expressions expand to <uri qname>.

```n3
@prefix : <foo://bla/names#> .
@prefix family: <foo://bla/persons/> .
  family:john :name "John"; :age 35; :hasChild family:alice, family:bob.
  family:bob :name "Bob"; :age 8 .
```

[Filename: RDF/john.n3]

Equivalent:

```n3
<foo://bla/persons/john> <foo://bla/names#name> "John"; <foo://bla/names#age> 35;
```

[Filename: RDF/john-expanded.n3]

- Note: hierarchically structured URLs begin with “/”
  (the Jena SPARQL tool otherwise complains)
EXPANSION OF PREFIXES

illustrate the equivalence by stating the same query against both N3 “fact bases”:

```sparql
prefix : <foo://bla/names#> # we need only this for the :name predicate
select ?X ?N
from <file:john.n3>
where {?X :name ?N}
[Filename: RDF/john.sparql]
```

```sparql
prefix : <foo://bla/names#>
select ?X ?N
from <file:john-expanded.n3>
where {?X :name ?N}
[Filename: RDF/john-expanded.sparql]
```

• all positions are subject to expansion.

N3 PREFIX NOTATION: COMMENTS

• empty prefix: define “@prefix : uri” and just use “:qname”

• additionally, a set of non-empty prefixes can be defined as “@prefix pre: uri” and use “pre:qname”

• the results are URIs “below” the URI that is defined for the prefix

• usually, the prefix uris end with “#”, or with “/”
  – “#”: generates expressions like (“hash-namespace”) foo://bla/names#age (which is a property), or foo://bla/names#Country (which is a class), or foo://bla/names#alice (which is an individual).
  – “/”: generates expressions like (“slash-namespace”) <foo://bla/persons/alice> (which is an individual).
  – without these that results just in a string concatenation.

• note that after prefix: only qnames are allowed. Usage with paths like pre:qname1/qname2 is not allowed.

• note the similarity with the use of namespaces in XML:
  xmlns = uri use <elementname>
  xmlns:ns = uri use <uri:elementname>
MERGING RDF DATA

- easy Web-wide data integration when using agreed **URIs** and **notions**.
- just load several RDF files into one model: same URIs are identified.

```n3
@prefix : <foo://bla/names#> .
@prefix family: <foo://bla/persons/> .

family:jack :name "Jack"; :age 65; :hasChild family:john .
family:kate :name "Kate"; :hasChild family:john .
```

[Filename: RDF/parents.n3]

- `foo://bla/names#` is the same URL base as earlier in `john.n3`:

```sparql
prefix : <foo://bla/names#>
```

[Filename: RDF/parents.sparql]

URIS AND ONTOLOGIES

- the URIs will usually be organized in an **ontology**.
- note: these URIs can also be URLs that refer to something useful – the RDF+Metadata world is a complex one.
  - later with RDFS, OWL, and LOD.
4.3 SPARQL: An RDF Query Language

- Basis: conjunctive queries over triples,
- syntactically very similar and equivalent to Datalog queries,
- variables with multiple occurrences act as *join variables*,
- mixed with SQL-style syntax.

// SPARQL
select ?P ?N
from <url> # optional, multiple input graphs

is the same as
// DATALOG
?- name(_X, P), child(_X, _Y), name(_Y, N).

- instead ?X, also $X can be written.
- SPARQL’s FROM does not correspond to SQL’s selection of input relations, but to
accessing multiple databases (cf. SchemaSQL)

---

Basic SPARQL Syntax

```
SELECT result variables
FROM input
WHERE { dot-separated sequence of triple-patterns and FILTER expressions }
```

- capitalization of keywords is optional
- triple-pattern: `x y z` as above
- FILTER expression: `FILTER ( predicate expression over variables and literals )`
  (see also Functions & Operators in SPARQL)

Prefix: `<foo://bla/names#>`
select ?P ?A
from `<file:john.n3>

[Filename: RDF/age-test.sparql]
- Object lists: `x p y1, y2`
- Predicate lists: `x p1 y1; p2 y2;`
SPARQL on a Database: Playing with Mondial

• above family examples: minimal toy examples
• Mondial: mondial.n3 and mondial-europe.n3 see Web page

RDF and SPARQL: Class Membership

• RDF built-in predicate: rdf:type for “is a” in RDF namespace
  `<http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
  `family:john rdf:type family:Person .`
• Short form: “a” (to be used without prefix!)

```sparql
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>

SELECT ?C ?N
FROM <file:mondial-europe.n3>
WHERE { ?C a mon:Country; mon:name ?N }
```

[Filename: RDF/mondial-simple.sparql]

SPARQL: BASICS OF FORMAL SEMANTICS

• RDF Terms: (“terms” is a bad terminology; actually these are the constant symbols)
  – IRI: the set of all IRIs (Internationalized Resource Identifiers)
  – RDF-L: the set of all RDF Literals
  – RDF-B: the set of all blank nodes (see later)
  – RDF-T = IRI ∪ RDF-L ∪ RDF-B
• Var: set of variables
• Triple Pattern $P$: an element of $(\text{RDF-T} \cup \text{Var}) \times (\text{IRI} \cup \text{Var}) \times (\text{RDF-T} \cup \text{Var})$
  (this allows literals at subject position which does not match anything)
• Basic Graph Pattern: a set of triple patterns.
SPARQL: Basics of Formal Semantics (Cont'd)

... consider again BGPs over RDF-Terms and variables:

• a mapping $\mu$ from Var to RDF-T is a partial function $\mu : \text{Var} \rightarrow \text{RDF-T}$.
  (also called “(answer) substitutions” or “tuples of variable bindings”; cf. Datalog)
  for $(?X \text{ mon:capital } ?Y)$, e.g. $\mu_1 = \{ X \mapsto <\text{germany}>, Y \mapsto <\text{berlin}> \}$

• $\text{dom}(\mu) := \{ v \in \text{Var} : \mu(v) \text{ is defined} \}$

• for a BGP $P$, let $\text{var}(P) \subseteq \text{Var}$ denote the set of variables that occur in $P$.

• for a BGP $P$ and a mapping $\mu$ s.t. $\text{dom}(\mu) \supseteq \text{var}(P)$, $\mu(P)$ denotes the replacement of
  every $v \in \text{Var}$ in $P$ by $\mu(v)$.
  – $\mu(P)$ is a (ground) instance of $P$, i.e. a small RDF graph;
  – standard logic terminology: here the mapping is seen as a substitution (i.e., a
    syntactic mapping) that yields a ground instance of an expression.

• for a BGP $P$ and an RDF graph $G$, an answer substitution is any substitution
  $\mu : \text{var}(P) \rightarrow \text{RDF-T}$ such that $\mu(P) \subseteq G$
  (modulo blank node renaming, which can be done by using don’t care variables in the
  query).

• The answer set to a BGP wrt. a graph $G$ is the set of all answer substitutions.

SPARQL: Querying Metadata

It is totally natural in RDF and SPARQL to query also metadata:

• which properties does John have?

```
prefix : <foo://bla/names#>
prefix family: <foo://bla/persons/>
select ?P
from <file:john.n3>
where { family:john ?P ?Y }
```

[Filename: RDF/john-properties.sparql]

• cf. SchemaSQL and F-Logic (“history” part of the SSD&XML lecture)

• especially, F-Logic (1989) had a strong influence on later languages for semistructured
  data, knowledge representation and the Semantic Web.
**SPARQL: DISTINCT**

- which properties does John have, remove duplicates?

```
prefix : <foo://bla/names#>
prefix family: <foo://bla/persons/>
select distinct ?P
from <file:john.n3>
where { family:john ?P ?Y}
```

[Filename: RDF/john-distinct-properties.sparql]

- correctly removes duplicates.

---

**SPARQL: OPTIONAL CONJUNCTS**

```
{ · · · . OPTIONAL { group pattern . · · · } }
```

binds the variables in the inner pattern if it is satisfied:

```
prefix : <foo://bla/names#>
SELECT ?N ?A
FROM <file:john.n3>
FROM <file:parents.n3>
```

[Filename: RDF/optional-age.sparql]

will always bind ?N, but ?A will be bound only if the age is known.
SPARQL: OPTIONAL (Cont'd)

- note: Filter inside OPTIONAL have local scope:

```sparql
prefix : <foo://bla/names#>
SELECT ?N ?A
FROM <file:john.n3>
FROM <file:parents.n3>
     OPTIONAL { ?X :age ?A . FILTER (?A < 60) } }
```

[Filename: RDF/optional-age-2.sparql]

will always bind ?N, but ?A will be bound only if the age is known, and is less than 60. If the age of ?X is known, but ≥ 60, the OPTIONAL part will simply bind nothing, but it will not evaluate to “false”.

- How to express “list ?X if no age is known or if its age is known to be less than 60”? (i.e. “if the age may be less than 60”) (see later)

SPARQL: OPTIONAL (Cont'd)

Example

OPTIONAL is bound all-or-nothing:

```sparql
prefix : <foo://bla/names#>
FROM <file:john.n3>
FROM <file:parents.n3>
```

[Filename: RDF/age-pairs.sparql]

The result bindings contain the ages only if for both persons, the age is given.
SPARQL: Nested OPTIONAL

Nested OPTIONAL:

```sparql
prefix : <foo://bla/names#>
FROM <file:john.n3>
FROM <file:parents.n3>
WHERE {
  OPTIONAL { ?X :age ?A . FILTER (?A < 60)
}
```

[Filename: RDF/age-and-children.sparql]

Only when an age less than 60 is given, then also the children are optionally returned.

---

SPARQL FILTER CONDITIONS

- atomic comparisons (=, <, ≤, >, ≥) with variables and constants as usual, string, date etc. predicates (see manual),
- logical connectives NOT ("!"), AND ("&&"), OR ("||") as usual,
- BOUND(?X) ... evaluates to true if ?X is bound in an answer,
- special predicates isIRI(?X), isLiteral(?X), etc. (see manual)

Semantics of Predicates on unbound variables

- (?X=?Y) when one of them is not bound: unknown (W3C Doc: error)

```sparql
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
select ?N ?P
from <file:mondial-europe.n3>
where { {?C a mon:City; mon:name ?N . OPTIONAL { ?C mon:population ?P } }
  # FILTER (!(?P > 10000)) # try this also
  FILTER (!(?P > 10000)) }
```

[Filename: RDF/negatedcomparison.sparql]

- returns only those where population is known and < 10000.
### SPARQL: Negation

- `!BOUND(?X)` is the only form of negation in SPARQL.
- `OPTIONAL` together with filtering can be used for checking that a property is *not* satisfied:

```sparql
prefix : <foo://bla/names#>
SELECT ?N ?A
FROM <file:john.n3>
FROM <file:parents.n3>
  OPTIONAL { ?X :age ?A . FILTER (?A > 60) } 
  FILTER ( !BOUND(?A) ) } 

[Filename: RDF/optional-age-negated.sparql]
```

- `OPTIONAL` binds `?A` if an age is given and `> 60`, and then `FILTER` removes those where `?A` is bound.
- people without an age or with age `< 60` are returned.
- equivalent to “Negation as Failure” in Logic Programming.
- equivalent semantics in *defeasible logics*: “people where it is consistent to assume that their age could be `< 60`”.

---

### Another Example: Closed World Negation

- all countries that are not (known to be) neighbors of each other
- Close the predicate “neighbor” for the query: “neighbor does *only* hold for the pairs where it is explicitly known”.

```sparql
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
SELECT ?C1 ?C2
FROM <file:mondial-europe.n3>
WHERE { ?C1 a mon:Country . ?C2 a mon:Country
  OPTIONAL { ?C1 mon:neighbor ?X . FILTER ( ?X = ?C2 ) }
  FILTER ( !BOUND(?X) ) }

[Filename: RDF/no-neighbor1.sparql]
```

- The closing of the predicate is restricted to the *query*, but cannot be used in the OWL part.
- more intuitive syntactic sugar has been introduced with SPARQL 1.1 (cf. Slide 161).
Another Variant of the same Query

- all countries that are not (known to be) neighbors of each other.
- closed predicate "neighbor": "neighbor does only hold for the pairs where it is explicitly known".

```sparql
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>

SELECT ?C1 ?C2
FROM <file:mondial-europe.n3>
WHERE
  { ?C1 a mon:Country . ?C2 a mon:Country
  OPTIONAL
FILTER ( ! BOUND(?N) )
```

[Filename: RDF/no-neighbor2.sparql]

- note: the additional property used in the OPTIONAL must be present for every ?C1 (and should be easy to compute)

EXERCISE

- For each country, give the name, and the population. If more than 1/4 of the population are living in its capital, give also the name and the population of the capital.
- ... to be done before discussing the formal semantics.
- Give the same queries in SQL and in XML/XQuery.
SPARQL: UNION

- \{ subpattern_1 \} UNION \{ subpattern_2 \}

subpatterns can bind different variables (recall that in contrast, safety in the relational calculus requires that both disjuncts bind the same variables):

@prefix : <foo://bla/names#> . @prefix family: <foo://bla/persons/> .
family:sue :name "Sue"; :age 32; :mother family:kate.
family:peter :name "Peter"; :age 28; :father family:jack; :hasChild family:andy.
family:andy :name "Andy"; :age 4 . family:kate :name "Kate".

prefix : <foo://bla/names#>
from <file:father-mother.n3>

Exercise: if both parents are given, put both into the same answer.
(Note: in the relational algebra, this is called a full outer join).

SPARQL: NAMED GRAPHS

The RDF model can distinguish between stored graphs:

- Default Graph and Named Graphs
- Default Graph specified by FROM <uri>
- Named Graphs specified by FROM NAMED <uri>.

prefix : <foo://bla/names#>
FROM NAMED <file:john.n3>
FROM NAMED <file:father-mother.n3>
WHERE { { GRAPH <file:john.n3> { ?X :name ?N }}
  UNION
  { GRAPH <file:father-mother.n3> { ?Y :name ?M }}}
SPARQL: Named Graphs

- which individuals are listed with properties in both graphs?

```sparql
PREFIX : <foo://bla/names#>
SELECT ?X ?P1 ?P2
FROM NAMED <file:parents.n3>
FROM NAMED <file:father-mother.n3>
WHERE {
  GRAPH <file:parents.n3> { ?X ?P1 ?Y1 }
  .
  GRAPH <file:father-mother.n3> { ?X ?P2 ?Y2 }
}
```

[Filename: RDF/both-graphs.sparql]

- Can be used e.g. for loading multiple graphs and constraining properties to the data of only some of them.

---

SPARQL: Disjunction

- Recall: logical “or” is allowed inside filters on conditions.

```sparql
PREFIX : <http://www.semwebtech.org/mondial/10/meta#>
SELECT ?XN ?P ?A
FROM <file:mondial-europe.n3>
  FILTER (?P > 50000000 || ?A > 400000) }
```

[Filename: RDF/or.sparql]

How to express a more complex disjunction?

Give all cities that are the capital of a country or the headquarter of an organization, or both. Give also the respective country and/or organization

```sparql
PREFIX : <http://www.semwebtech.org/mondial/10/meta#>
SELECT ?XN ?CC ?A
FROM <file:mondial-europe.n3>
WHERE { ?X a :City; :name ?XN
  OPTIONAL { ?O :hasHeadq ?X; :abbrev ?A }
  FILTER (BOUND(?C) || BOUND(?O)) }
```

[Filename: RDF/disjunction.sparql]
SPARQL RESULT MODIFIERS

- formal semantics: yields a set of answer substitutions \( \mu : \text{var}(P) \rightarrow \text{RDF-T} \).
- cf. SQL: SELECT expression, XQuery: return expression
- ORDER BY: SELECT ... WHERE ... ORDER BY directives
  where directives is a sequence of expressions of the forms
  - variable, equivalent \( \text{ASC}(\text{variable}) \), and
  - \( \text{DESC}(\text{variable}) \)
- Projection: SELECT variables FROM ... WHERE ...
- DISTINCT: SELECT DISTINCT variables
- LIMIT \( n \): only a given number of results
- OFFSET \( k \): start with the \( k \)-th solution
- Syntax: SELECT ... WHERE ... [ORDER BY ...] [LIMIT \( n \)] [OFFSET \( k \)].
  (note that combination of ORDER BY, LIMIT and OFFSET can be used for e.g. returning the 3rd to 6th largest items etc.)

VARIABLE ALIASING (E.G. FOR ARITHMETIC EXPRESSIONS)

- similar as in SQL; but always in parentheses:
  (expression AS variable)

```sparql
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>

SELECT ?C (?P/?A AS ?Density) FROM <file:mondial-europe.n3>
WHERE
ORDER BY DESC(?Density)
```

[Filename: RDF/density.sparql]
sparql graph constructors

• use CONSTRUCT { dot-separated sequence of triple patterns } instead of simple SELECT
• returns an RDF Graph instead of variable bindings
• result triples that contain unbound variables (e.g. due to an OPTIONAL) are not added to the graph

prefix : <foo://bla/names#>
from <file:john.n3>
from <file:parents.n3>
[Filename: RDF/construct.sparql]

• just to test RDF/XML (see later for details):
jena -q -qf construct.sparql -ol RDF/XML -of grandchildren.rdf

shortcomings

Some can be solved by extended clause syntax (see SPARQL 1.1)

• No explicit difference/negation operation, only via OPTIONAL + FILTER + !BOUND(X),
• no grouping/aggregation.

Some are inherent to the language design

• SPARQL is not a closed language: input model (RDF) different from result model (sets of variable bindings).
⇒ no nested queries.
SPARQL AS A CLOSED LANGUAGE?

... would look like this:

```sparql
prefix : <foo://bla/names#>
select ?X
from
from <file:john.n3>
from <file:parents.n3>
}
where { ?X ?P ?Y }
```

[Filename: RDF/construct-nested.sparql]

- Note: this is not yet possible in SPARQL
- Possible Master’s Thesis: extend the JENA SPARQL module accordingly.

COMPARISON WITH MULTIDATABASES

- Multidatabases in the “old” times: a set of autonomous databases that provide correlated contents
- cf. the section on SchemaSQL in the SSD&XML lecture:
  ```sql
  SELECT ... FROM db1::rel1, db2::rel2, ...
  ```
  (and similar expressions)
- the SPARQL FROM clause also selects multiple input RDF data sources

Current Situation

Seeing the RDF Web as a multidatabase, RDF and SPARQL provide a unified model and language for data integration (via the URIs).

Semantic Web Vision

The actual origin of the RDF data is not specified by the user: users query “the Semantic Web” (via a portal) and relevant data sources will be selected transparently.
4.4 SPARQL: Formal Semantics

SPARQL is based on graph pattern matching:

• input: a graph $G$,

• query: consists of
  – Basic Graph Patterns (where clause) – conjunctive query (join)
  – Filters (where clause)
  – Optional (outer join)
  – Union

result modifiers:
  – order by (evaluated after query evaluation)
  – projection (in the select clause)
  – duplicate elimination (in the select clause)

• Similarity with the relational model and a similar algebraic characterization.

• Note that similar to SQL, actual evaluation does not use the algebra directly, but applies optimized algorithms.

Formal Semantics – Considerations

• strong similarity with relational model

• algebraic, compositional semantics desirable (cf. SQL, OQL, XQuery)

• SPARQL W3C Candidate Recommendation until v.20061004 gave a non-algebraic semantics informally in “natural language” – which is problematic for some cases.

  – develops an algebraic, compositional semantics,
  – gives a formalization of the W3C Semantics, compares both semantics.


• title says it all: Jorge Pérez: Semantic Web Research Inspired by W3C Standards, or the Hell of the Practice without Theory. AMW 2012: 19 (tutorial available at http://users.dcc.uchile.cl/~jperez/talks/amw2012-tutorial.pdf)
Formal Semantics – Comparison with SQL

- both are basically conjunctive queries (CQs) – joins;
- SQL: base relations and views: schema: \((A_1, \ldots, A_n)\) – attributes are (usually) DB column names;
- SPARQL: tuples of variable bindings (cf. Datalog: matching variables with patterns): \((X_1, \ldots, X_n)\);
- FILTER is a selection;
- select clause (SPARQL: only finally) applies a projection;
- SPARQL: no nesting of subqueries (but nesting of \{\ldots\}-groups with CQs, OPTIONAL and FILTER);
- \textbf{OPTIONAL} is an outer join (with was a rather rarely used operator in SQL) \Rightarrow \text{introduces lots of NULL values}
  (which are much more frequent in the scenario of incomplete knowledge in the Web than in SQL)
- details of NULL values have only been sketched (pragmatically) in the DB lecture (and in the SQL lab)!

NULL VALUES IN SQL

- NULLs do not violate any integrity constraint, e.g.,
  CREATE TABLE Country
  
  ( ... , population NUMBER CHECK (population \(\geq 0\)) , ...)

- note that this also holds for referential integrity constraints! e.g.
  INSERT INTO located VALUES('Berlin',null,'A','Rhein',null,null);
  is not forbidden!
- NULLs do not satisfy any WHERE condition (except IS NULL)
- Join conditions in SQL are \textit{null-rejecting};
  a join including a null-containing column must use separate IS NULL checks or COALESCE.
  (example see next slide)
CREATE VIEW OneProvCountry AS
SELECT country FROM province GROUP BY country HAVING COUNT(*) = 1;

CREATE TABLE city2 AS
(SELECT name, province, country, population FROM city
WHERE country NOT IN (SELECT country FROM OneProvCountry))
UNION
(SELECT name, NULL, country, population FROM city
WHERE country IN (SELECT country FROM OneProvCountry));

CREATE TABLE located2 AS
(SELECT city, province, country, river, lake, sea FROM located
WHERE country NOT IN (SELECT country FROM OneProvCountry))
UNION
(SELECT city, NULL, country, river, lake, sea FROM located
WHERE country IN (SELECT country FROM OneProvCountry));

SELECT c.name, c.population, l.river FROM city2 c, located2 l
WHERE c.name=l.city AND c.province=l.province AND c.country=l.country;
-- 958 results, the ones with NULL province are missing.

SELECT c.name, c.population, l.river FROM city2 c, located2 l
WHERE c.name=l.city
    AND COALESCE(c.province,'bla')=COALESCE(l.province,'bla')
-- note that the condition is not satisfied if one value is null, and the other is not!
    AND c.country=l.country;
-- 1157 results

• table “located”: (name/country/province of the city, river, lake, sea) for cities located at waters. Contains many null values.

select * from lake -- 189
select * from lake where name in (select lake from located) -- 39
select * from lake where name not in (select lake from located) -- 0
select * from lake where name not in (select lake from located where lake is not null) -- 150

• if a set contains “null”, every value might be equal to this “unknown” null.
4.4.1 SPARQL Algebra

(according to [PAG06])

Algebraic Syntax of SPARQL Queries

• deals with the graph pattern part, not the result modifiers (which contains projection and some operators that are also not covered by the relational algebra, like DISTINCT, ORDER BY, LIMIT, OFFSET)

• fully parenthesized

• Triple patterns (cf. Slide 112) in \((\text{RDF-T} \cup \text{Var}) \times (\text{IRI} \cup \text{Var}) \times (\text{RDF-T} \cup \text{Var})\) are graph patterns;

• For graph patterns \(P_1\) and \(P_2\), the expressions \((P_1 \text{ AND } P_2)\), \((P_1 \text{ OPT } P_2)\), and \((P_1 \text{ UNION } P_2)\) are graph patterns;

• For a graph pattern \(P\) and a condition \(R\), \((P \text{ FILTER } R)\) is a graph pattern.
  (Require filter-safeness: \(\text{var}(R) \subseteq \text{var}(P)\); cf. Slide 142)

• \(\text{var}(P)\) denotes the variables occurring in a graph pattern \(P\).

Filter-Safe Graph Patterns

Definition 4.1 ([PAG06, AG 08])

A SPARQL algebra expression is \textit{filter-safe}, if for every subexpression of the form \((P \text{ FILTER } R)\), \(\text{var}(R) \subseteq \text{var}(P)\).

• for the \textit{compositional} semantics the requirement is absolutely necessary – what should be the semantics of \((\{X \text{ p } Y\} \text{ FILTER } (?Y > ?Z))\) ?

• The commonly used SPARQL pattern

\[
\{ \text{?P1 a :Person; } :\text{age } \text{?A1. ?P2 a :Person} \\
\text{OPTIONAL } \{ \text{?P2 :age ?A2. FILTER } (\text{?A2 > ?A1 }) \} \}
\]

is not filter-safe. The filter actually formulates a join condition on the outer join:

\[
(P_1 \text{ a Person; age } A_1 . P_2 \text{ a Person}) \overset{\geq \text{1}}{\leftrightarrow}_{A_1 < A_2} (P_2 \text{ age } A_2)
\]

• [AG08] gives an algorithm that transforms a non-filter-safe SPARQL query into a filter-safe algebra expression.
  (DBTheory: similar to the transformation of general safe formulas into RANF when moving conjuncts into negated subformulas to make them \textit{self-contained}; cf. exercises).

• Non-compositional sideways-information-passing evaluation is often also much more efficient, cf. evaluation of SQL vs. relational algebra.
Exercise
Recall Slide 124:

• “For each country, give the name, and the population. If more than 1/4 of the population are living in its capital, give also the name and the population of the capital.”

• check whether your solution is filter-safe. If not, transform it into an equivalent filter-safe query.

SPARQL Algebra: Mappings/Answer Substitutions

• a mapping $\mu$ from $\text{Var}$ to RDF-T is a partial function $\mu : \text{Var} \rightarrow \text{RDF-T}$. (also called “answer substitutions” or “tuples of variable bindings”; cf. Datalog) for $(?X \text{mon:capital} ?Y)$, e.g. $\mu_1 = \{ X \mapsto \langle \text{germany} \rangle , Y \mapsto \langle \text{berlin} \rangle \}$

• $\text{dom}(\mu) := \{ v \in \text{Var} : \mu(v) \text{ is defined} \}$

• for a SPARQL query $P$, let $\text{var}(P) \subseteq \text{Var}$ denote the set of variables that occur in $P$.

• for a triple pattern $t$, and a mapping $\mu$ s.t. $\text{dom}(\mu) \supseteq \text{var}(P)$, $\mu(t)$ denotes the triple obtained by replacing all variables $v \in \text{Var}$ in $P$ by $\mu(v)$.

• two mappings $\mu_1$ and $\mu_2$ are compatible if for all $v \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, $\mu_1(v) = \mu_2(v)$, i.e., when $\mu_1 \cup \mu_2$ is also a mapping (with $\text{dom}(\mu) = \text{dom}(\mu_1) \cup \text{dom}(\mu_2)$).
  – cf. definition of joining tuples $\mu_1 \mu_2$ in the relational algebra,
  – here using $\text{dom}(\mu)$ (which is flexible wrt. null values) instead of the fixed schema/format of an expression.

• now consider sets $\Omega$ of such mappings (instead of sets of fixed-format tuples in the relational algebra) ...
Semantics of Expressions and Operations on Sets of Mappings

Recall: Relational Algebra defines operators \( \pi, \sigma, \bowtie \) etc. on sets of tuples.

Common general notation: \([P]\) for “semantics of \(P\)”; \([P]_D\) for “wrt. a given graph \(D\)”.

- base case: triple patterns:
  \([t]_D = \{ \mu \mid \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in D \}\).

- operations on sets of mappings:
  - projection: not needed here; as a result modifier, it is external to the core algebra.
  - selection: next slide.
- \([\{P_1 \cup P_2\}]_D = [P_1]_D \cup [P_2]_D\);
- \([\{P_1 \cap P_2\}]_D = [P_1]_D \bowtie [P_2]_D\) with
  \(\Omega_1 \bowtie \Omega_2 = \{ \mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \text{ and } \mu_2 \in \Omega_2 \text{ are compatible} \}\);
- \([\{P_1 \oplus P_2\}]_D = [P_1]_D \leftrightarrow [P_2]_D\) with
  \(\Omega_1 \leftrightarrow \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)\) where \(\setminus\) is defined as
  \(\Omega_1 \setminus \Omega_2 = \{ \mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible} \}\);

(\(\text{note that } \"\setminus\" \text{ here does not denote set difference, but has a different definition}).

Semantics of Expressions and Operations on Sets of Mappings: Filters

Three-valued logic for evaluating conditions when variables are not bound:

- truth values \(t\) (true, 1), \(u\) (undefined, 0.5), \(f\) (false, 0), ordered by \(t > u > f\).

- All three-valued logics coincide in the definition of \(\land, \lor, \text{ and } \lnot\):

<table>
<thead>
<tr>
<th>(A \land B = \min(A, B))</th>
<th>(A \lor B = \max(A, B))</th>
<th>(\lnot A = 1 - a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B)</td>
<td>(f)</td>
<td>(u)</td>
</tr>
<tr>
<td>(A)</td>
<td>(f)</td>
<td>(f)</td>
</tr>
</tbody>
</table>

- selection: \([\{P \text{ FILTER } R\}]_D = \{ \mu \in [P]_D \mid \text{val}(R, \mu) = t \}\) with
  - \(\text{val}(\text{bound}(?X), \mu) = (X \in \text{dom}(\mu))\) with \((X \in \text{dom}(\mu))\) as truth value \(t, f \in \{t, u, f\}\);
  - \(\text{val}(?X = ?Y, \mu) = \begin{cases} t & \text{if } X \in \text{dom}(\mu), Y \in \text{dom}(\mu), \text{ and } \mu(X) = \mu(Y) \\ f & \text{if } X \in \text{dom}(\mu), Y \in \text{dom}(\mu), \text{ and } \mu(X) \neq \mu(Y) \\ u & \text{if } X \notin \text{dom}(\mu) \text{ or } Y \notin \text{dom}(\mu); \end{cases}\)
  - \(\text{val}(?X = c, \mu)\) analogously;
  - for \((R_1 \land R_2), (R_1 \lor R_2), \text{ and } (\lnot R_1)\) see above truth tables.
Bag vs. Set Semantics

- SPARQL (like SQL) has a bag (=multiset) semantics.
- the above algebra (like the relational algebra) is defined wrt set semantics;
- can be generalized to bag semantics.

Exercises

1. define a relational “null-tolerant join” that acts like $\bowtie$ above.
2. which SQL construct is similar to the “\” operator in the above SPARQL algebra?
3. in the above algebra, OPT is expressed via left outer join, which is defined via “\” (while a corresponding MINUS does not exist in the SPARQL syntax).
   Such a MINUS (cf. Exercise (2)) provides a more intuitive idea of negation than “!bound(\?X)”. Give a general pattern how to express $(P_1 \text{ MINUS } P_2)$ in SPARQL syntax.
4. recall the definition of $\bowtie$ in the relational algebra (DB lecture) and define SPARQL’s $\bowtie$ in a similar way.

Exercise

Prove or show a counterexample:

The statement (from W3C SPARQL Working Draft 20061004)

If OPT(A, B) is an optional graph pattern, where A and B are graph patterns, then S is a solution of OPT(A,B) if

- S is a pattern solution of A and of B, or
- S is a solution to A, but not to A and B.

describes the same semantics as above.
Theorem 4.1 ([AG08])

- SPARQL (=version 1.0, = the above algebra) has the same expressive power as non-recursive safe Datalog with negation.
  (recall: non-recursive with negation is stratifiable)
- SPARQL (=version 1.0, = the above algebra) has the same expressive power as the relational algebra (under set semantics, and under bag semantics).

Proof ([AG08]):

- mapping from SPARQL to the filter-safe algebra expressions;
- mappings between the (filter-safe) SPARQL algebra and Datalog/relational algebra.

Normal Form

Theorem 4.2 ([PAG06])

Every graph pattern is equivalent to a pattern in the normal form

\[(P_1 \text{ UNION} \cdots \text{ UNION} P_n)\]

where each \(P_i\) is a UNION-free graph pattern.

Proof: UNION is distributive wrt. each of the operators:

- \((P_1 \text{ AND} (P_2 \text{ UNION} P_3)) \equiv (P_1 \text{ AND} P_2) \text{ UNION} (P_1 \text{ AND} P_3),\)
- \((P_1 \text{ UNION} P_2) \text{ OPT} P_3 \equiv (P_1 \text{ OPT} P_3) \text{ UNION} (P_2 \text{ OPT} P_3),\)
- \((P_1 \text{ UNION} P_2) \text{ FILTER} R \equiv (P_1 \text{ FILTER} R) \text{ UNION} (P_2 \text{ FILTER} R),\)
- note that the case for \((P_1 \text{ OPT} (P_2 \text{ UNION} P_3))\) is harder to prove [PAG09].
Complexity

Consider the problem “is $\mu \in [P]_D$”? So-called “data complexity” for fixed $P$ as a function of the size of $D$:

• The problem is in LOGSPACE (which is a subset of the complexity class P, i.e., it is polynomial).

So-called “combined complexity” as a function of $P$ and $D$:

• Conjunctive queries: the problem is in $O(|P| \cdot |D|)$ for graph patterns containing only AND and FILTER. [PAG06]

• the problem is NP-complete for graph patterns containing only AND, FILTER, and UNION. [PAG06]

• the problem is PSPACE-complete for graph patterns in general. [PAG06]

• the problem is PSPACE-complete for graph patterns over OPT only. [SML10]


Well-Designed Graph Patterns

Definition 4.2 ([PAG06]) A graph pattern is well-designed if for every subpattern $P’ = (P_1 \text{ OPT } P_2)$: every variable $?X$ that occurs inside $P_2$ and also outside $P’$, also occurs in $P_1$.

• Some non-well-designed graph patterns exhibit weird behavior (cf. exercises).

• Note that “well-designedness” according to [PAG06] excludes negation: usage of

{ pattern(?X_1,\ldots,?X_n) \text{ OPTIONAL } \{ pattern(?X_{i_1},\ldots,?X_{i_k},?Y) \} }

FILTER (!BOUND(?Y))

requires $?Y$ to occur inside $P_2$, outside $P’$, but not inside $P_1$. ([AP11] argues that (SPARQL closed-world) negation is not appropriate for the open Web scenario with incomplete data)

• The role of well-designed patterns for SPARQL is comparable to the role of conjunctive queries for the relational algebra.

Theorem 4.3 ([PAG09]) Evaluation is coNP-complete for the fragment of SPARQL consisting of well-designed patterns. (which is significantly cheaper than PSPACE for whole SPARQL)
Theorem 4.4 ([PAG06])

Every union-free well-designed graph pattern (without filters) is equivalent to a pattern in the normal form

\[((\cdots((t_1 \text{ AND} \cdots \text{ AND} t_k) \text{ OPT } O_1) \text{ OPT } O_2) \cdots ) \text{ OPT } O_n\]

where each \(t_i\) is a triple pattern and each \(O_j\) is also in this normal form.

\[\blacksquare\]

- + UNION: push up/out according to Theorem 4.2,
- + FILTERS: apply them as early as possible.

**Aside: Weak Monotonicity**

Recall Monotonicity of queries:
- A query \(Q\) is monotonic if for any databases \(D_1 \subseteq D_2\), \([Q]_{D_1} \subseteq [Q]_{D_2}\).

**Example 4.1 (A monotonic query)**

\(Q_1 = \text{select } ?X \text{ where } {?X \text{ a} : \text{Person}}\)

\(D_1 = \{\{\text{john} \text{ a} : \text{Person}\}\}\)

\([Q_1]_{D_1} = \{\{?X \mapsto \text{john}\}\}\)

\([Q_1]_{D_2} = \{\{?X \mapsto \text{john}\}, \{?X \mapsto \text{mary}\}\}\)

\([Q_1]_{D_1} \subseteq [Q_1]_{D_2}\). \[\square\]

**Example 4.2**

\(Q_2 = \text{select } ?X ?A \text{ where } {?X \text{ a} : \text{Person} \text{ OPTIONAL } {?X : \text{age} ?A}}\)

\(D_3 = \{\text{john} \text{ a} : \text{Person}; \text{ age } 35}\)

Again, \([Q_2]_{D_1} = \{\{?X \mapsto \text{john}\}\}\)

\([Q_2]_{D_3} = \{\{?X \mapsto \text{john}, ?A \mapsto 35\}\}\)

Here, \([Q_2]_{D_1} \not\subseteq [Q_2]_{D_3}\), but “nearly”.

The answer tuple in \([Q_2]_{D_3}\) extends the tuple in \([Q_2]_{D_1}\) - “more information” \[\square\]
Weak Monotonicity (cont’d)


Definition 4.3 ([AP11])
• For mappings $\mu_1$ and $\mu_2$, $\mu_1$ is subsumed by $\mu_2$, denoted by $\mu_1 \preceq \mu_2$, if $\text{dom}(\mu_1) \subseteq \text{dom}(\mu_2)$ and $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1)$.

• For sets $\Omega_1$ and $\Omega_2$ of mappings, $\Omega_1$ is subsumed by $\Omega_2$, denoted by $\Omega_1 \subseteq \Omega_2$, if for every $\mu_1 \in \Omega_1$, there exists $\mu_2 \in \Omega_2$ such that $\mu_1 \preceq \mu_2$.

• A SPARQL graph pattern $P$ is said to be weakly monotonic if for every pair $G_1, G_2$ of RDF graphs such that $G_1 \subseteq G_2$, it holds that $[P]_{G_1} \subseteq [P]_{G_2}$.

The above $Q_2$ is weakly monotonic.

Theorem 4.5 ([AP11])
Every well-designed graph pattern is weakly monotonic.

• the converse direction is an open question, but is expected to hold.

Some Observations
• Recall: Closed-World negation is nonmonotonic (additional knowledge may invalidate before conclusions)

• Recall: Open-World negation is monotonic (learning something that was not known to hold or not to hold before may not invalidate any conclusion made before).

• Recall: Default reasoning (like “normally ... holds”) is nonmonotonic (learning that something is an expection from normality may invalidate some conclusions made before).

• weak monotonicity itself excludes that any conclusion made before may be invalidated.
Some Observations (cont’d)
Consider queries $Q$ with an arbitrary construct clause instead of a select clause (using the same variables):

- for every input RDF graph $G$, their result $[Q]_{G}^{\text{construct}}$ is an RDF graph,
- $[Q]_{G_1} \subseteq [Q]_{G_2}$ implies $[Q]_{G_1}^{\text{construct}} \subseteq [Q]_{G_2}^{\text{construct}}$.

$\Rightarrow$ weak monotonicity wrt. sets of tuples of variable bindings implies monotonicity wrt. the constructed graph.
$\Rightarrow$ the definition of “weak monotonicity” is only necessary because of the (relational-style) output format of SPARQL queries.
- from the informational point of view, weak monotonicity means monotonicity.

BACK TO “WELL-DESIGNED” QUERIES

Theorem 4.5 then reads as
- Every well-designed graph pattern with construct output is monotonic. (which is rather obvious from its normal form)
- if the converse direction –which is expected to hold– holds, every monotonic SPARQL graph mapping can be expressed in the above normal form.

4.5 SPARQL 1.1 - Syntactic Extensions

- path expressions (by “/”) that consist of two or more subexpressions
- $^\wedge expr$ for inverse (and $expr^\wedge expr$ for expr/$expr$),
- $expr | expr$ for alternative,
- $expr^*$ and $expr^+$ for transitive closure, $expr\{n,m\}$ for $n$ to $m$ expr steps.

```sparql
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
# http://jena.apache.org/documentation/query/property_paths.html

FROM <file:mondial-europe.n3>
WHERE
{ ?C a mon:Country ;
  mon:capital/mon:population ?P ;
  mon:capital/^mon:hasHeadq ?O ;
  # mon:neighbor+ ?C2 ;
  mon:neighbor{2,4} ?C2 }
```

[Filename: RDF/mondial-path-exprs.sparql]
SPARQL 1.1 - Syntactic Extensions

• arithmetic and string expressions + aliasing in the select clause
• \( \text{str(expr)} \) for converting URIs into strings, \( \text{uri(string)} \) for creating URIs from strings.

```sparql
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
# http://jena.apache.org/documentation/query/select_expr.html

SELECT ?C (?P/?A as ?D) (uri(concat(str(?CAP),'bla')) as ?U)
FROM <file:mondial-europe.n3>
WHERE
  mon:capital ?CAP }
```

[Filename: RDF/mondial-select-exprs.sparql]

• “if(a,b,c)” for “if a is true, then use b, else c”,
• “\text{coalesce(expr}_1\ldots,\text{expr}_n\text{)}” returns the value of the first \text{expr}_i that evaluated without error (unbound counts as error).

SPARQL 1.1 - Syntactic Extensions

• NOT EXISTS pattern in FILTER (syntactic sugar for the \text{!BOUND(var)} construct)

```sparql
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
# http://jena.apache.org/documentation/query/negation.html

SELECT ?C
FROM <file:mondial-europe.n3>
WHERE
{ ?C a mon:Country
  FILTER NOT EXISTS { ?C mon:neighbor ?C2 } }
```

[Filename: RDF/mondial-not-exists.sparql]

• Official syntax:
  – NOT EXISTS is a boolean truth value function used as filter,
  – some implementations accept SQL-style NOT EXISTS in a BGP without the FILTER keyword.
NOT EXISTS as Relational Difference

Consider again Slide 122:

- “all pairs of countries that are not (known to be) neighbors of each other”.
- The FILTER NOT EXISTS pattern can be used for complex negation (relational “anti-join”, including the relational difference).

```sparql
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
SELECT ?C1 ?C2
FROM <file:mondial-europe.n3>
WHERE
{ ?C1 a mon:Country . ?C2 a mon:Country
  FILTER NOT EXISTS { ?C1 mon:neighbor ?C2 }
}
```

[Filename: RDF/no-neighbor3.sparql]

- Recall: this is closed-world negation.

GROUP BY + HAVING and aggregations

```sparql
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
# http://jena.apache.org/documentation/query/group-by.html
SELECT ?C (count(?Cty) AS ?CT)
FROM <file:mondial-europe.n3>
WHERE
{ ?C a mon:Country ;
  (mon:hasCity|(mon:hasProvince/mon:hasCity)) ?Cty
}
GROUP BY ?C
HAVING (count(?Cty) > 1)
```

[Filename: RDF/mondial-group-by.sparql]
Subqueries

- WHERE \{ basic graph pattern . \{ SELECT ... [FROM ...] WHERE ... \} \}
- cf. SQL: subqueries in the SQL FROM clause contributing to the join,
- implicit join variables,
- no subqueries in the FILTER (SQL's where clause).
- recall from SQL/rel.Algebra: every SQL query with a subquery can be rewritten into an algebra query (which does not allow for subqueries in selection predicates).
- main use: compute some aggregate in the subquery:

```sparql
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
SELECT ?X ?MAXPOP
FROM <file:mondial.n3>
WHERE {
  \{ SELECT (MAX(?YP) AS ?MAXPOP)
  WHERE \{ ?Y a mon:Country; mon:population ?YP \} \}
}
[Filename: RDF/biggestcountry.sparql]
```

Example with Subquery + GROUP BY

- for each country, give its biggest city, if this has more than 1000000 inhabitants:

```sparql
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
FROM <file:mondial.n3>
WHERE {
  ?Y a mon:Country ; mon:carCode ?C ; 
  mon:hasProvince/mon:hasCity
  [ mon:name ?CN ; mon:population ?MP ].
  \{ SELECT ?Y (MAX(?YP) AS ?MP)
  WHERE \{
  ?Y a mon:Country;
  mon:hasProvince [ mon:hasCity [ mon:population ?YP ]].
  \#\# mon:hasProvince/mon:hasCity/mon:population ?YP. does not work!
  \}
  \} GROUP BY ?Y
  HAVING (?MP > 1000000)
}
ORDER BY ?MP
[Filename: RDF/biggestcities.sparql]
```
4.6 RDF vs. Object Identity

- So far, RDF is just a restricted variant of ground (sorted) relational First-Order Logic atoms:
  (cf. Slides 202 for a more detailed consideration)
  - only binary predicates
  - literal values and relationships carry semantics
  - URIs as constant symbols
  - no function symbols other than constants
- Sorted domain: URIs and Literals

Existential Knowledge

Sometimes the existence of things is relevant, without actually naming and giving an ID to an object.

- “A parent is a person that has at least one child”
- “John has a child of 12 years”

Blank Nodes – Existential Quantification

- Short form for \( x \, p \, y_1 \), \( y_1 \, q_1 \, z_1 \), and \( x \, p \, y_2 \), \( y_2 \, q_2 \, z_2 \) when the URIs for \( y_1 \) and \( y_2 \) are not relevant (purely existential “blank nodes”):
  \[ x \, p \, [q_1 \, z_1], [q_2 \, z_2] \, . \]

RDF “Model”

Equivalent expression in FOL:

\[
\exists j : (\text{name}(j, \text{“John”}) \land \text{age}(j, 35) \land \\
\quad \land \exists a, b, c : (\text{child}(j, a) \land \text{child}(j, b) \land \text{child}(j, c) \land \\
\quad \land \text{name}(a, \text{“Alice”}) \land \text{age}(a, 10) \land \text{name}(b, \text{“Bob”}) \land \text{age}(b, 8) \land \text{age}(c, 12))
\]
Example: Blank Nodes

```sparql
@prefix : <foo://bla/names#>.
[ :name "John"; :age 35;
   :hasChild [:name "Alice"; :age 10],
       [:name "Bob"; :age 8], [:age 12] ] .
```

[Filename: RDF/john-blank.n3]

```sparql
prefix : <foo://bla/names#>
select ?X ?N
from <file:john-blank.n3>
where {?X :name ?N ; :hasChild [:age 12] }
```

[Filename: RDF/john-blank.sparql]

- generated by “[...]” above: “something that satisfies”
- nested “term” syntax for implicit conjunction of atoms, without using identifiers or key references.
  [Note that F-Logic terms (1989) and JSON expressions (2006, 2013) have a very similar structure]

Example:
Extend the database such that John is married to Mary, who is the mother of both children.

```
@prefix : <foo://bla/names#>.
[ :name "John"; :age 35;
   :hasChild _:a, _:b ;
_:a :name "Alice"; :age 10 .
_:b :name "Bob"; :age 8 .
```

[Filename: RDF/john-married.n3]

```sparql
prefix : <foo://bla/names#>
select ?N ?C ?A
from <file:john-married.n3>
```

[Filename: RDF/john-married.sparql]

NAMED BLANK NODES

- blank nodes can also be named by local identifiers of the form “:_:localname” that allow for “referencing” things again (existential variables).

```sparql
@prefix : <foo://bla/names#>.
[ :name "John"; :age 35;
   :hasChild _:a, _:b ;
_:a :name "Alice"; :age 10 .
_:b :name "Bob"; :age 8 .
```

[Filename: RDF/john-married.n3]

```sparql
prefix : <foo://bla/names#>
select ?N ?C ?A
from <file:john-married.n3>
```

[Filename: RDF/john-married.sparql]
Consider again John's family and its logical formalization (Slides 167 and 166).

```prolog
@prefix : <foo://bla/names#>.
[ :name "John"; :age 35;
 :hasChild [[:name "Alice"; :age 10] ,
            [:name "Bob"; :age 8] , [:age 12 ]].
```

[Filename: RDF/john-blank.n3]

How many children does John have?

• what does a human reader understand?
• is this entailed by the logical formalization?

Pitfalls of Existential Knowledge (Cont’d)

• Alice:
  [ :name "John"; :hasChild [:name "Alice"] ]

• His health insurance:
  [ :name "John"; :hasChild [:age 10] ]

• Grandmother:
  [ :name "John"; :hasChild [:nickname "My sunshine"] ]

• A neighbor:
  [ :name "John"; :hasChild [:nickname "The little beast"] ]

How many children does John have?
– merging this yields one RDF graph.
– the logical formalization is very similar to the one from the previous slide.
Pitfalls of Existential Knowledge (Cont’d)

• from the first example, most people conclude that John has three children.
  (note that “at least three” would be more exact)

• from the second example (text) many people conclude that all statements describe Alice

• the RDF graph of the second example, most people conclude that there are four children

• formally in all cases: the knowledge bases entail only that John has at least one child!

• for the first example:
  – there is a logical model where there is only one child with three names and a property
    “age” that has three values.
  – people use meta knowledge that age and name are functional properties. This is not
    contained in the RDF data!

• OWL allows to express such additional information.

• the next slides anticipate some OWL stuff that will be discussed in detail later.

Show that it is consistent to assume that John has exactly one child:

• define a class OneChildParent that restricts the cardinality of child to one and make John
  an instance of it (using OWL; see later):

```n3
@prefix : <foo://bla/names#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
[ :name "John"; :age 35;
  :hasChild [:name "Alice"; :age 10], [:name "Bob"; :age 8], [:age 12 ];
  rdf:type :OneChildParent].
:OneChildParent owl:equivalentClass [rdf:type owl:Restriction;
  owl:onProperty :hasChild; owl:cardinality 1].
```

[Filename: RDF/john-silly-example.n3]

• how old is Alice (invoke with -pellet)?

```sparql
prefix : <foo://bla/names#>
select ?X ?A from <file:john-silly-example.n3>
where {?X :name "Alice" . ?X :age ?A}
```

[Filename: RDF/john-silly-example.sparql]
Pitfalls of Existential Knowledge (Cont’d)

• Assert that age is a functional property:

```
@prefix : <foo://bla/names#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.

[:name "John"; :age 35; 
 :hasChild [:name "Alice"; :age 10], [:name "Bob"; :age 8], [:age 12 ];
 rdf:type :OneChildParent].

:OneChildParent owl:equivalentClass [rdf:type owl:Restriction;
 owl:onProperty :hasChild; owl:cardinality 1].

```

[Filename: RDF/john-silly-example-2.n3]

```
prefix : <foo://bla/names#>
select ?X ?A from <file:john-silly-example-2.n3>
where {?X :name "Alice" . ?X :age ?A}
```

[Filename: RDF/john-silly-example-2.sparql]

• pellet complains about an inconsistent ontology.

Aside: Reasoning under Inconsistency

Consider the situation of the previous slide.

• Semantics of boolean implication:
  Any formula of the form \( \text{false} \rightarrow \varphi \) is true in any interpretation (equivalent to \( \neg \text{false} \lor \varphi \)),
  which is in turn equivalent to \( \text{true} \lor \varphi \).

• Given an inconsistent ontology formalization \( \Phi \), for every formula \( \varphi \), \( \Phi \models \varphi \) holds:
  for every model \( M \) such that \( M \models \Phi \) (note that there are no such \( M \)), also \( M \models \varphi \) holds.

• Tableau Calculus: Given an ontology formalization \( \Phi \),
  prove \( \Phi \models \varphi \) by starting a tableau over \( \Phi \land \neg \varphi \) and trying to close it.
  If already \( \Phi \) is inconsistent, any tableau over \( \Phi \) can be closed without using \( \neg \varphi \). Thus, \( \Phi \vdash \text{Tableau} \varphi \).

For that reason, it is reasonable that a prover even does not start working with an inconsistent ontology.

• research issue: reduce an inconsistent ontology to a consistent subset and prove or refute \( \varphi \) from this.
Pitfalls of Existential Knowledge (Cont’d)

• Assert that age is a functional property and look what is entailed ...

```r
@prefix : <foo://bla/names#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.

[ :name "John"; :age 35;
  :hasChild [:name "Alice"; :age 10], [:name "Bob"; :age 8], [:age 12 ]].
:OneChildParent owl:equivalentClass [rdf:type owl:Restriction;
  owl:onProperty :hasChild; owl:cardinality 1].
:TwoChildrenParent owl:equivalentClass [rdf:type owl:Restriction;
  owl:onProperty :hasChild; owl:cardinality 2].
:ThreeChildrenParent owl:equivalentClass [rdf:type owl:Restriction;
  owl:onProperty :hasChild; owl:minCardinality 3].
```

[Filename: RDF/john-three-children.n3]

(continue next slide)

```
prefix : <foo://bla/names#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?X ?C
from <file:john-three-children.n3>
where { ?X :name "John" . ?X rdf:type ?C }
```

[Filename: RDF/john-three-children.sparql]

• SPARQL counts the bindings:

```r
prefix : <foo://bla/names#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?X count(?C)
from <file:john-blank.n3>
where { ?X :name "John" . ?X rdf:type ?C }
group by ?X
```

[Filename: RDF/john-count.sparql]

• count(?C) is 3.
  
  SPARQL semantics is closed-world, and “unique blank node assumption”
```
**UNIQUE NAME ASSUMPTION (DATALOG)**

*Datalog* makes the *Unique Name Assumption*: different constants denote different things, e.g.

\[
\text{_sibling} (X, Y) \leftarrow \text{child} (_Z, X), \text{child} (_Z, Y), X \neq Y.
\]

will derive sibling(alice,bob) when facts child(john,alice) and child(john,bob) are given.

**NO UNIQUE NAME ASSUMPTION IN RDF**

- RDF model theory: objects described by different URIs may be the same!
- even if the children are explicitly described by URIs, the consequences are the same as in the previous example,
- in OWL, it can be explicitly stated that two URIs denote the same or different things.
- equivalence: e.g. used in ontology mapping.
- if no such information is explicitly given, both is assumed to be possible.
- equality or non-equality can be derived (e.g., by declaring a property to be functional)
  (Note that the same is the case for F-Logic model theory with functional methods.)

**ASIDE: SOME THEORY**

Consider the N3 fragment consisting of expressions as follows:

- no explicit subjects,
- no URIs at object positions (only literals and [...] expressions),
- no named blank nodes,
- i.e., constants are only literals and property name URIs.

Expressiveness:

- Only existentially quantified variables.
- can only represent tree-like relationship structures. No cycles (cf. the “mary” example).
- The expressiveness is exactly that of FOL formulas over an alphabet of only two variables.
  (note that these can be used several times)

Exercise: express the RDF/N3 “database” given on Slide 166 by a first-order formula that uses only two variables.
**Some Theory (Cont’d)**

- FOL with two variables is decidable,
- satisfiability of first-order formulas that use three variables is in general undecidable ("3-SAT")!
- restriction to two variables: guarantees a tree-like “structure” of the models (“tree model property”). This locality guarantees decidability.
- for the same reason, also many variants of Description Logics (see later) are decidable.

Proofs use the equivalence to modal logics over tree-like Kripke Structures.

This RDF fragment will be relevant later when considering decidable fragments of Description Logics and OWL.

**Semantics: Negation**

- compare the FOL axiomatizations of John and his family – they always describe only the things that are there, and say nothing about the (non)-existence of other things:

  Has John a child of the age of 14?

  The database does not find any, but the formula

  \[ \exists j : (\ldots \land \exists d : \text{child}(j, d) \land \text{age}(d, 14)) \]

  is consistent – thus it has models. It is actually possible that John has such a child which is simply unknown to the database ("Open-World-Assumption" (OWA)).

- another RDF Web source could add RDF triples that describe additional children of John.
- Minimal-Model-based reasoning or SQL databases would answer “no” “Closed-World-Assumption” (CWA).
- SPARQL has CWA; it would find (and count) three children of John (kind of “unique blank node assumption”)
- in general: OWA is more appropriate to the Web.
- so far: SPARQL does only matching on an RDF graph. Later: RDF/RDFS/OWL model that extends the graph by further knowledge that is also presented in RDF format.
4.7 RDF Conceptual Model and Built-in Predicates

... so far, RDF defines just an edge-labeled graph. RDF provides also a simple type system (which is extended later by RDF Schema):

- the RDF conceptual model knows about class/type membership and properties,
- `rdf:type`: is a special property that assigns a type (means: class membership) to something

TYPES AND PROPERTIES AS RESOURCES

Extend the “Persons” running example from Slide 104:

```n3
@prefix : <foo://bla/names#> .
@prefix family: <foo://bla/persons/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
family:john rdf:type :Person; :name "John"; :age 35;
         :hasChild family:alice, family:bob.
family:bob :name "Bob"; :age 8 .
```

- `<foo://bla/persons/john>` (and others) are the URIs of objects that are described by the database,
- `<foo://bla/names#Person>` is the URI of the class “Person”
- `<foo://bla/names#name>` and `<foo://bla/names#age>` are the URIs of the property names.

Design ontologies in RDF like this.
• separate part for “notions” (often as a “#”-namespace)
• separate part for objects (either as “#”-namespace or as “/”-namespace)
• note that after prefix: only qnames are allowed. Usage with paths like pre:qname1/qname2 is not allowed.
• [later in RDF/XML xml:base can be used together with paths]

```xml
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <foo://bla/names#> .
@prefix persons: <foo://bla/persons/> .
@prefix cats: <foo://bla/cats/> .
persons:john rdf:type :Person; :name "John"; :age 35;
   :has_cat cats:garfield.
```

[Filename: RDF/john-and-the-cat.n3]

• There are <foo://bla/persons/john> and <foo://bla/cats/garfield>.

---

**Ontology Design and Query Formulation**

```sparql
prefix terms: <foo://bla/names#>;
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?X ?T ?A
from <file:john-and-the-cat.n3>
where { ?X rdf:type ?T ; terms:age ?A }
```

[Filename: RDF/john-and-the-cat.sparql]

• note that most queries only need the prefix for the metadata
• the prefix/base for data is only used when a query explicitly accesses an object by its URI

```sparql
base <foo://bla/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?T ?A
from <file:john-and-the-cat.n3>
where { <persons/john> rdf:type ?T ; <names#age> ?A }
```

[Filename: RDF/sparql-base.sparql]

• Alternative: value-based selection of an object
```
... where { [ :name "John" ; rdf:type ?T; :age ?A ] }
```
METADATA AS RESOURCES

... on the way to more expressive ontology languages:

- Some metadata notions are defined in the rdf namespace
  - `@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
  - `@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>`
  - `rdf:type` as a property: `:john rdf:type :Person`
    (short syntax in N3: `:john a :Person`)
  - `rdf:Property` as a class: `:hasChild rdf:type rdf:Property`
  - `rdfs:Class` (and `owl:Class`) as a class: `:Person rdf:type rdfs:Class`
- transition/not-so-well-defined language border to RDF Schema (RDFS)
- RDF tools know about `rdf:type`, `rdf:Property`, `rdfs:Class` etc.

note: `rdf:type` non-capitalized (is a property), `rdf:Property` (is a type) capitalized.

TYPES AND PROPERTIES

- these things are built-in notions that have in pure RDF still no meaning.
- Types are *not* XML Schema complex types (structural typing), but *semantical types*.
- Types/classes and properties are also resources that can be described this way.
- nevertheless, in pure RDF there is no kind of signature or schema; especially also no type constraints on properties.
- it’s just data.
RDF Instance Data and Metadata Examples

- Consider
  - reasonable: (http://.../geo#germany rdf:type http://.../geo#Country)
  - reasonable: (http://.../geo#berlin rdf:type http://.../geo#City)
  - non-reasonable: (http://.../geo#germany rdf:type http://.../geo#berlin)
  ⇒ things that are of some type should not be types themselves?

- but this crashes with meta-metadata:
  - reasonable: (http://.../bio#Penguin rdf:type http://.../bio#Species)
  - reasonable: (http://.../bio#tweety rdf:type http://.../bio#Penguin)

- Consider
  - reasonable: (http://.../#has_capital rdf:type rdf:Property)
  - not reasonable: (http://.../#has_capital http://.../#has_population 42)
  ⇒ a property does not have other properties?

- but this crashes with meta-metadata:
  - reasonable: (http://.../geo#has_capital rdf:type rdf:Property)
  - reasonable: (http://.../geo#has_capital http://.../meta#has_range http://.../geo#City)

⇒ these “legal” and “non-legal” usages are not enforced by anything. Non-legal usage can lead to severe problems (see Slide 210).

4.8 Example: Mondial in RDF

- RDF level: structure of URIs, types and relationships
- later: RDFS and OWL add further metadata information and knowledge

Design of the Mondial Ontology

- we chose classes and properties of the Mondial ontology to be identified by
  - http://www.semwebtech.org/mondial/10/meta#classname

- the URIs for identifying countries, organizations etc. are actual URLs formed like
  - http://www.semwebtech.org/mondial/10/countries/code
  - http://www.semwebtech.org/mondial/10/countries/code/cities/name

- Alternatives will be discussed later (with RDF/XML).
EXAMPLE FRAGMENT

@prefix monmeta: <http://www.semwebtech.org/mondial/10/meta#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
<http://www.semwebtech.org/mondial/10/countries/D>
  rdf:type monmeta:Country ;
  monmeta:name "Germany" ;
  monmeta:code "D" ;
  monmeta:population 83536115 ;
  monmeta:capital

• Mondial N3 available on the Web site  
  (note: not yet stable ... still experimenting)

• the type “Country” is  <http://www.semwebtech.org/mondial/10/meta#Country>

• the property “capital” is  <http://www.semwebtech.org/mondial/10/meta#capital>

• it would be nice to be able to set @base <http://www.semwebtech.org/mondial/10/> for
  the individuals’ URIs, but this is not supported in N3; see later for RDF/XML.

base <http://www.semwebtech.org/mondial/10/>  
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>  
select ?Y  
from <file:a-bit-mondial.n3>  
where { <countries/D> rdf:type ?Y }

Relationship with XML Namespaces

Consider the following XML fragment:

<mon:mondial xmlns:mon="http://www.semwebtech.org/mondial/10/meta#">
  <mon:Country car_code="D">
    <mon:name>Germany</mon:name>
  </mon:Country>
</mon:mondial>

There’s more behind these namespaces, which is not used in plain XML.  
See later with RDF/XML.
**RDF SUMMARY**

- more or less simple data model,
- possibility to use property and class names that are agreed Web-wide,
- the idea of URIs,
- the contents of RDF files can be integrated in a natural way via URIs and names.

**4.9 Further Topics**

- some additional RDF syntax: Reification and Collections. Later.
- Schema information: RDF Schema
- RDF/RDFS Model Theory + Reasoning
- More than schema information: Ontology specification by Description Logics and OWL
- How to provide RDF data and metadata on the Web?
  Later: RDF/XML. It's just another representation of RDF, RDF Schema and OWL data in a special XML syntax.
- What's missing?
- Rules?