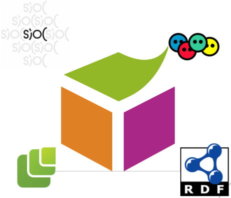
Transforming between RDF and XML with XSPARQL

Net2 Tutorial

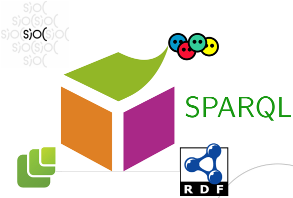
Nuno Lopes

December, 2010

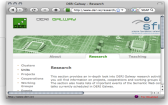
Integration of Heterogeneous Sources



Integration of Heterogeneous Sources



Integration of Heterogeneous Sources



DERI Galway - Research

http://www.deri.ie/research/

SOAP

DERI GALWAY

About Research Teaching

- Clusters
- Units
- Projects
- Cooperations
- Working Groups
- Events

Research

This section provides an in-depth look into DERI Galway research activiti
you will find information on projects, cooperations and working groups D
The section also hosts lists of important events of the Semantic Web con
talks currently scheduled in DERI Galway.

http://www.deri.ie/

Tor Disabled

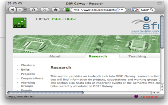
<XML/>

SOAP/WSDL

SPARQL

Integration of Heterogeneous Sources

XSLT/XQuery



DERI Galway - Research

http://www.deri.ie/research/

SOAP

DERI GALWAY

About Research Teaching

- Clusters
- Units
- Projects
- Cooperations
- Working Groups
- Events

Research

This section provides an in-depth look into DERI Galway research activiti you will find information on projects, cooperations and working groups D The section also hosts lists of important events of the Semantic Web con talks currently scheduled in DERI Galway.

http://www.deri.ie/

Tor Disabled

<XML/>

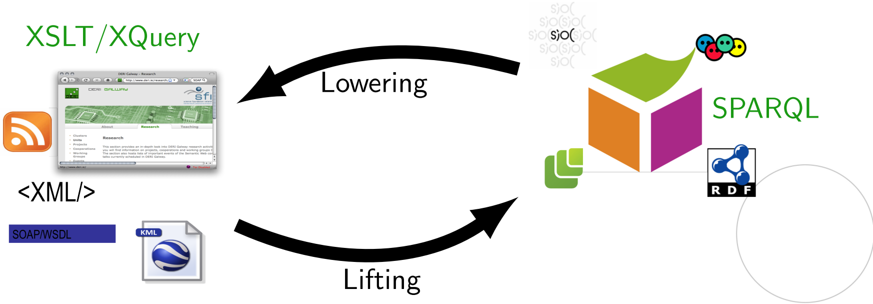
SOAP/WSDL

KML

SPARQL

RDF

Integration of Heterogeneous Sources



XSLT/XQuery

Lowering

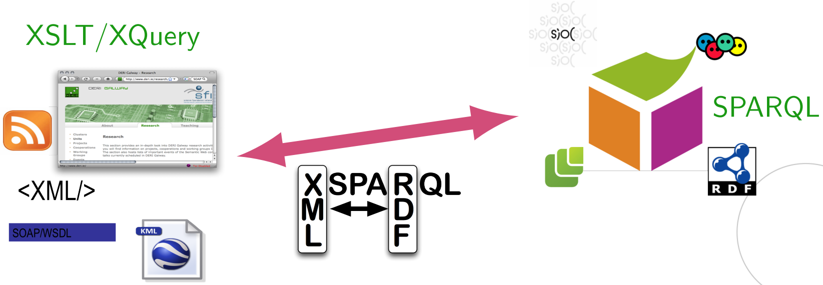
<XML/>

Lifting

SPARQL

Transformations between XML and RDF are not easy, mainly due to the heterogeneity of RDF/XML serialisations

Integration of Heterogeneous Sources

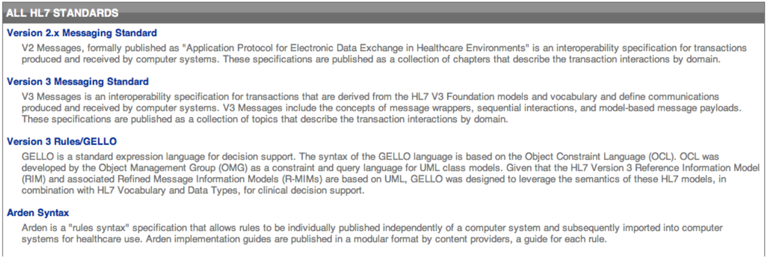


Transformations between XML and RDF are not easy, mainly due to the heterogeneity of RDF/XML serialisations

Objective: language capable of integrating heterogeneous sources for the Semantic Web

Why are such transformations needed? (I)

Standards in Health Care and Life Sciences



ALL HL7 STANDARDS

Version 2.x Messaging Standard

V2 Messages, formally published as "Application Protocol for Electronic Data Exchange in Healthcare Environments" is an interoperability specification for transactions produced and received by computer systems. These specifications are published as a collection of chapters that describe the transaction interactions by domain.

Version 3 Messaging Standard

V3 Messages is an interoperability specification for transactions that are derived from the HL7 V3 Foundation models and vocabulary and define communications produced and received by computer systems. V3 Messages include the concepts of message wrappers, sequential interactions, and model-based message payloads. These specifications are published as a collection of topics that describe the transaction interactions by domain.

Version 3 Rules/GELLO

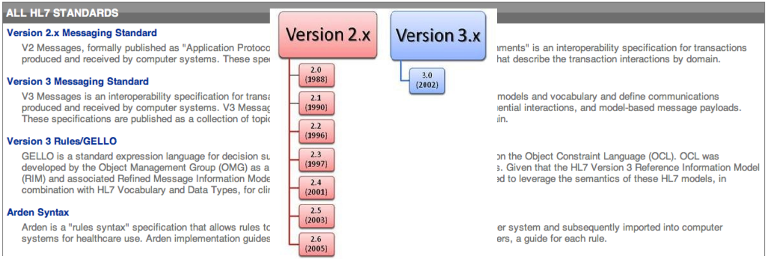
GELLO is a standard expression language for decision support. The syntax of the GELLO language is based on the Object Constraint Language (OCL). OCL was developed by the Object Management Group (OMG) as a constraint and query language for UML class models. Given that the HL7 Version 3 Reference Information Model (RIM) and associated Refined Message Information Models (R-MIMs) are based on UML, GELLO was designed to leverage the semantics of these HL7 models, in combination with HL7 Vocabulary and Data Types, for clinical decision support.

Arden Syntax

Arden is a "rules syntax" specification that allows rules to be individually published independently of a computer system and subsequently imported into computer systems for healthcare use. Arden implementation guides are published in a modular format by content providers, a guide for each rule.

Why are such transformations needed? (I)

Standards in Health Care and Life Sciences



ALL HL7 STANDARDS

Version 2.x Messaging Standard

V2 Messages, formally published as "Application Protoco ıments" is an interoperability specification for transactions
produced and received by computer systems. These spe hat describe the transaction interactions by domain.

Version 3 Messaging Standard

V3 Messages is an interoperability specification for trans models and vocabulary and define communications
produced and received by computer systems. V3 Messag ıential interactions, and model-based message payloads.
These specifications are published as a collection of topi ıin.

Version 3 Rules/GELLO

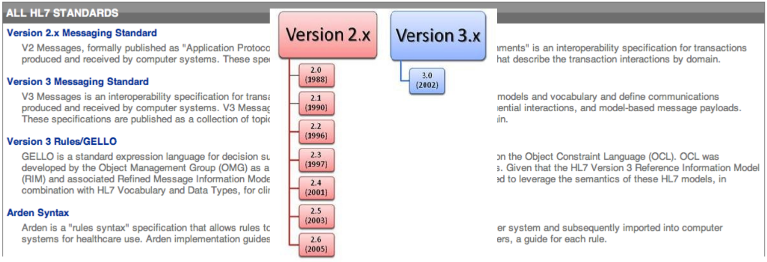
GELLO is a standard expression language for decision su on the Object Constraint Language (OCL). OCL was
developed by the Object Management Group (OMG) as a s. Given that the HL7 Version 3 Reference Information Model
(RIM) and associated Refined Message Information Mode ed to leverage the semantics of these HL7 models, in
combination with HL7 Vocabulary and Data Types, for cli

Arden Syntax

Arden is a "rules syntax" specification that allows rules tc er system and subsequently imported into computer
systems for healthcare use. Arden implementation guides ers, a guide for each rule.

Why are such transformations needed? (I)

Standards in Health Care and Life Sciences



ALL HL7 STANDARDS

Version 2.x Messaging Standard

V2 Messages, formally published as "Application Protoco ments" is an interoperability specification for transactions produced and received by computer systems. These spe hat describe the transaction interactions by domain.

Version 3 Messaging Standard

V3 Messages is an interoperability specification for trans models and vocabulary and define communications produced and received by computer systems. V3 Messag iential interactions, and model-based message payloads. These specifications are published as a collection of topi iin.

Version 3 Rules/GELLO

GELLO is a standard expression language for decision su on the Object Constraint Language (OCL). OCL was developed by the Object Management Group (OMG) as a s. Given that the HL7 Version 3 Reference Information Model (RIM) and associated Refined Message Information Mode ed to leverage the semantics of these HL7 models, in combination with HL7 Vocabulary and Data Types, for clir

Arden Syntax

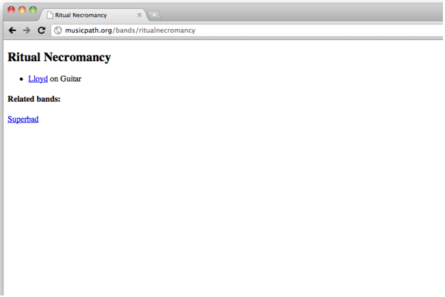
Arden is a "rules syntax" specification that allows rules tc er system and subsequently imported into computer systems for healthcare use. Arden implementation guides ers, a guide for each rule.

Possible solution for the heterogeneous message formats

- Store your data in RDF
- Convert it to the required XML format when necessary

Why are such transformations needed? (II)

Creating (X)HTML from an RDF backend (under development):
http://musicpath.org/



Ritual Necromancy

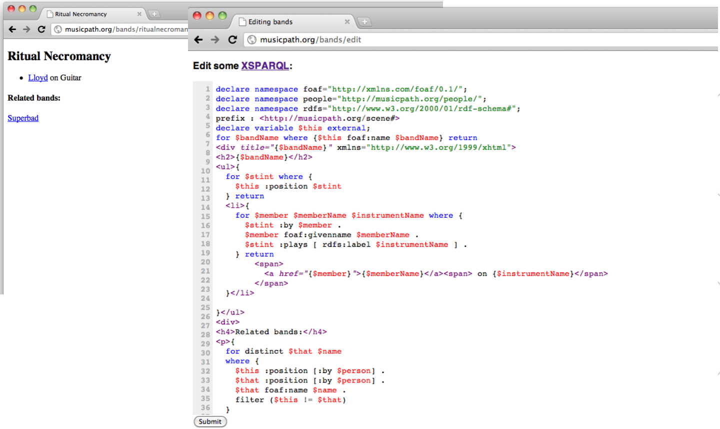
- Lloyd on Guitar

Related bands:

Superbad

Why are such transformations needed? (II)

Creating (X)HTML from an RDF backend (under development):
http://musicpath.org/



Ritual Necromancy

- Lloyd on Guitar

Related bands:

Superbad

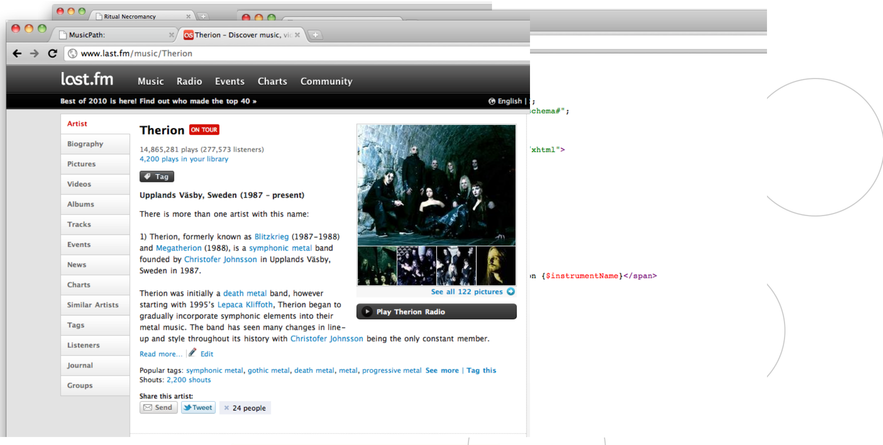
Edit some XSPARQL:

```
declare namespace foaf="http://xmlns.com/foaf/0.1/";
declare namespace people="http://musicpath.org/people/";
declare namespace rdfs="http://www.w3.org/2000/01/rdf-schema#";
prefix : <http://musicpath.org/scene#>
declare variable $this external;
for $bandName where {$this foaf:name $bandName} return
<div title="{$bandName}" xmlns="http://www.w3.org/1999/xhtml">
<h2>{$bandName}</h2>
<ul>{
  for $stint where {
    $this :position $stint
  } return
  <li>{
    for $member $memberName $instrumentName where {
      $stint :by $member .
      $member foaf:givenname $memberName .
      $stint :plays [ rdfs:label $instrumentName ] .
    } return
       <span>
         <a href="{$member}">{$memberName}</a><span> on {$instrumentName}</span>
       </span>
  }</li>

}</ul>
<div>
<h4>Related bands:</h4>
<p>{
  for distinct $that $name
  where {
    $this :position [:by $person] .
    $that :position [:by $person] .
    $that foaf:name $name .
    filter ($this != $that)
  }
```

Why are such transformations needed? (II)

Creating (X)HTML from an RDF backend (under development): http://musicpath.org/



Best of 2010 is here! Find out who made the top 40 »

Therion ON TOUR

14,865,281 plays (277,573 listeners)
4,200 plays in your library

Upplands Väsby, Sweden (1987 – present)

There is more than one artist with this name:

1) Therion, formerly known as Blitzkrieg (1987–1988) and Megatherion (1988), is a symphonic metal band founded by Christofer Johnsson in Upplands Väsby, Sweden in 1987.

Therion was initially a death metal band, however starting with 1995's Lepaca Kliffoth, Therion began to gradually incorporate symphonic elements into their metal music. The band has seen many changes in line-up and style throughout its history with Christofer Johnsson being the only constant member.

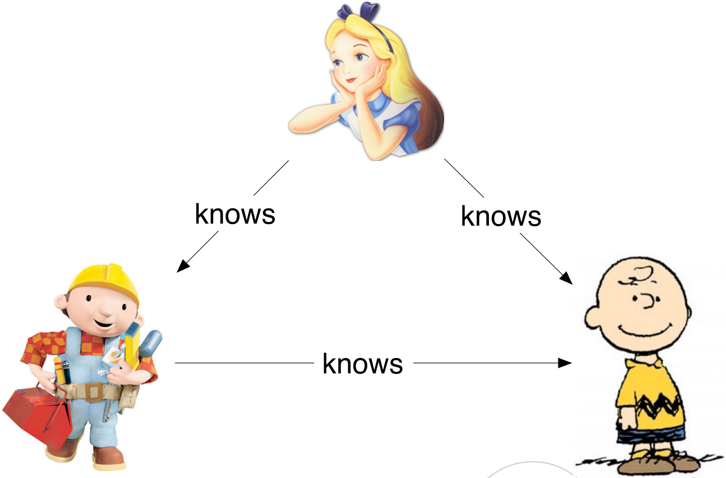
Read more... Edit

Popular tags: symphonic metal, gothic metal, death metal, metal, progressive metal See more | Tag this
Shouts: 2,200 shouts

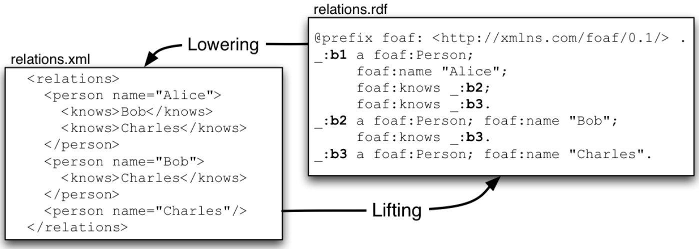
Share this artist:
24 people

```
;
chema#";
'xhtml">
n {$instrumentName}</span>
```

Toy example for the tutorial



Toy example for the tutorial



relations.rdf

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:b1 a foaf:Person;
     foaf:name "Alice";
     foaf:knows _:b2;
     foaf:knows _:b3.
_:b2 a foaf:Person; foaf:name "Bob";
     foaf:knows _:b3.
_:b3 a foaf:Person; foaf:name "Charles".
```

Lowering

relations.xml

```
<relations>
  <person name="Alice">
    <knows>Bob</knows>
    <knows>Charles</knows>
  </person>
  <person name="Bob">
    <knows>Charles</knows>
  </person>
  <person name="Charles"/>
</relations>
```

Lifting

knows

Why XQuery/XSLT is not enough:

- Different syntaxes and serialisations for the same RDF graph:

```
@prefix alice: <alice/> .
@prefix foaf: <...foaf/0.1/> .

_:b1 rdf:type foaf:Person;
     foaf:knows _:b2.
_:b2 rdf:type foaf:Person;
     foaf:name "Bob".
```

```
<rdf:RDF xmlns:foaf="...foaf/0.1/">
  <foaf:Person>
   <foaf:knows>
     <foaf:Person foaf:name="Bob"/>
   </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
  xmlns:rdf="...rdf-syntax-ns#">
  <rdf:Description rdf:nodeID="b1">
    <rdf:type
      rdf:resource=".../Person"/>
    <foaf:knows rdf:nodeID="b2"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="b2">
    <rdf:type
        rdf:resource=".../Person"/>
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
</rdf:RDF>
```

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
       xmlns:rdf="...rdf-syntax-ns#">
    <rdf:Description rdf:nodeID="x">
      <foaf:knows rdf:nodeID="y"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="x">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
</rdf:RDF>
```

Why XQuery/XSLT is not enough:

- Different syntaxes and serialisations for the same RDF graph:

Turtle

```
@prefix alice: <alice/> .
@prefix foaf: <...foaf/0.1/> .

_:b1 rdf:type foaf:Person;
     foaf:knows _:b2.
_:b2 rdf:type foaf:Person;
     foaf:name "Bob".
```

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
  xmlns:rdf="...rdf-syntax-ns#">
  <rdf:Description rdf:nodeID="b1">
    <rdf:type
      rdf:resource=".../Person"/>
    <foaf:knows rdf:nodeID="b2"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="b2">
    <rdf:type
        rdf:resource=".../Person"/>
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
</rdf:RDF>
```

```
<rdf:RDF xmlns:foaf="...foaf/0.1/">
  <foaf:Person>
   <foaf:knows>
     <foaf:Person foaf:name="Bob"/>
   </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
       xmlns:rdf="...rdf-syntax-ns#">
    <rdf:Description rdf:nodeID="x">
      <foaf:knows rdf:nodeID="y"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="x">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
</rdf:RDF>
```

Why XQuery/XSLT is not enough:

- Different syntaxes and serialisations for the same RDF graph:

```
@prefix alice: <.../alice/> .
@prefix foaf: <...foaf/0.1/> .

_:b1 rdf:type foaf:Person;
     foaf:knows _:b2.
_:b2 rdf:type foaf:Person;
     foaf:name "Bob".
```

Turtle

Triplestore

```
<rdf:RDF xmlns:foaf="...foaf/0.1/">
  <foaf:Person>
   <foaf:knows>
     <foaf:Person foaf:name="Bob"/>
   </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
  xmlns:rdf="...rdf-syntax-ns#">
  <rdf:Description rdf:nodeID="b1">
    <rdf:type
      rdf:resource=".../Person"/>
    <foaf:knows rdf:nodeID="b2"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="b2">
    <rdf:type
        rdf:resource=".../Person"/>
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
</rdf:RDF>
```

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
       xmlns:rdf="...rdf-syntax-ns#">
    <rdf:Description rdf:nodeID="x">
      <foaf:knows rdf:nodeID="y"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="x">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
</rdf:RDF>
```

Why XQuery/XSLT is not enough:

- Different syntaxes and serialisations for the same RDF graph:

```
@prefix alice: <alice/> .
@prefix foaf: <.../foaf/0.1/> .

_:b1 rdf:type foaf:Person;
     foaf:knows _:b2.
_:b2 rdf:type foaf:Person;
     foaf:name "Bob".
```

Triplestore

Turtle

```
<rdf:RDF xmlns:foaf="...foaf/0.1/">
  <foaf:Person>
   <foaf:knows>
     <foaf:Person foaf:name="Bob"/>
   </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

RDF/**XML**

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
  xmlns:rdf="...rdf-syntax-ns#">
  <rdf:Description rdf:nodeID="b1">
    <rdf:type
      rdf:resource=".../Person"/>
    <foaf:knows rdf:nodeID="b2"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="b2">
    <rdf:type
        rdf:resource=".../Person"/>
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
</rdf:RDF>
```

RDF/**XML**

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
       xmlns:rdf="...rdf-syntax-ns#">
    <rdf:Description rdf:nodeID="x">
      <foaf:knows rdf:nodeID="y"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="x">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
</rdf:RDF>
```

RDF/**XML**

Why XQuery/XSLT is not enough:

- Different syntaxes and serialisations for the same RDF graph:

Turtle

Triplestore

```
@prefix alice: <alice/> .
@prefix foaf: <...foaf/0.1/> .

_:b1 rdf:type foaf:Person;
     foaf:knows _:b2.
_:b2 rdf:type foaf:Person;
     foaf:name "Bob".
```

RDF/**XML**

```
<rdf:RDF xmlns:foaf="...foaf/0.1/">
  <foaf:Person>
   <foaf:knows>
     <foaf:Person foaf:name="Bob"/>
   </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

RDF/**XML**

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
  xmlns:rdf="...rdf-syntax-ns#">
  <rdf:Description rdf:nodeID="b1">
    <rdf:type
      rdf:resource=".../Person"/>
    <foaf:knows rdf:nodeID="b2"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="b2">
    <rdf:type
      rdf:resource=".../Person"/>
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
</rdf:RDF>
```

RDF/**XML**

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
       xmlns:rdf="...rdf-syntax-ns#">
    <rdf:Description rdf:nodeID="x">
      <foaf:knows rdf:nodeID="y"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="x">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
</rdf:RDF>
```

Any transformation needs to take into account the different RDF/XML representations

Why XQuery/XSLT is not enough:

- Different syntaxes and serialisations for the same RDF graph:

```
@prefix alice: <alice/> .
@prefix foaf: <.../foaf/0.1/> .

_:b1 rdf:type foaf:Person;
     foaf:knows _:b2.
_:b2 rdf:type foaf:Person;
     foaf:name "Bob".
```

Turtle

Triplestore

```
<rdf:RDF xmlns:foaf="...foaf/0.1/">
  <foaf:Person>
   <foaf:knows>
     <foaf:Person foaf:name="Bob"/>
   </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

RDF/XML

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
  xmlns:rdf="...rdf-syntax-ns#">
  <rdf:Description rdf:nodeID="b1">
    <rdf:type
      rdf:resource=".../Person"/>
    <foaf:knows rdf:nodeID="b2"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="b2">
    <rdf:type
        rdf:resource=".../Person"/>
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
</rdf:RDF>
```

RDF/XML

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
       xmlns:rdf="...rdf-syntax-ns#">
    <rdf:Description rdf:nodeID="x">
      <foaf:knows rdf:nodeID="y"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="x">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
</rdf:RDF>
```

RDF/XML

Or: end up with different transformations for the same RDF data

Why SPARQL is not enough:

Great for querying RDF! Easy to output Turtle or SPARQL XML results format …

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

construct { $X foaf:name $FN.}
from <vCard.rdf>
where { $X vc:FN $FN .}
```

… but

Why SPARQL is not enough:

Great for querying RDF! Easy to output Turtle or SPARQL XML results format ...

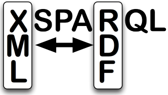
```
prefix vc: <http://www.w3.org/2001/vcard-rdf/3.0#>
prefix foaf: <http://xmlns.com/foaf/0.1/>

construct { $X foaf:name $FN.}
from <vCard.rdf>
where { $X vc:FN $FN .}
```

... but

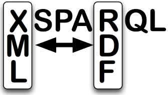
How to produce arbitrary XML???

XSPARQL

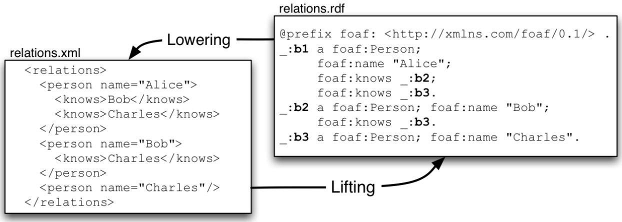


- Transformation language
- XML and RDF formats (based on XQuery and SPARQL)

XSPARQL



- Transformation language
- XML and RDF formats (based on XQuery and SPARQL)
- *Lifting* and *Lowering* in a single language



relations.xml

```
<relations>
  <person name="Alice">
    <knows>Bob</knows>
    <knows>Charles</knows>
  </person>
  <person name="Bob">
    <knows>Charles</knows>
  </person>
  <person name="Charles"/>
</relations>
```

relations.rdf

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:b1 a foaf:Person;
     foaf:name "Alice";
     foaf:knows _:b2;
     foaf:knows _:b3.
_:b2 a foaf:Person; foaf:name "Bob";
     foaf:knows _:b3.
_:b3 a foaf:Person; foaf:name "Charles".
```

Outline

Overview

Outline

Overview

XPath & XQuery

- XPath
- XQuery
- XQuery Semantics

Outline

Overview

XPath & XQuery

- XPath
- XQuery
- XQuery Semantics

SPARQL

Outline

Overview

XPath & XQuery

- XPath
- XQuery
- XQuery Semantics

SPARQL

XSPARQL

- Syntax
- Semantics
- Implementation
- XSPARQL Features
- Query examples

Outline

Overview

XPath & XQuery

- XPath
- XQuery
- XQuery Semantics

SPARQL

XSPARQL

- Syntax
- Semantics
- Implementation
- XSPARQL Features
- Query examples

XPath

- XPath is used to locate nodes in XML trees
- An XPath expression is a sequence of *steps* separated by /.
- Each *step* evaluates to a sequence of nodes.

relations.xml

```
<relations>
    <person name="Alice">
      <knows>Bob</knows>
      <knows>Charles</knows>
    </person>
    <person name="Bob">
      <knows>Charles</knows>
    </person>
    <person name="Charles"/>
</relations>
```

Full spec at http://www.w3.org/TR/xpath20/.
Tutorial at http://www.w3schools.com/xpath/default.asp.

XPath

- XPath is used to locate nodes in XML trees
- An XPath expression is a sequence of *steps* separated by /.
- Each *step* evaluates to a sequence of nodes.

relations.xml

```
<relations>
    <person name="Alice">
      <knows>Bob</knows>
      <knows>Charles</knows>
    </person>
    <person name="Bob">
      <knows>Charles</knows>
    </person>
    <person name="Charles"/>
</relations>
```

`relations` node is the root element

Full spec at http://www.w3.org/TR/xpath20/.
Tutorial at http://www.w3schools.com/xpath/default.asp.

XPath

- XPath is used to locate nodes in XML trees
- An XPath expression is a sequence of *steps* separated by /.
- Each *step* evaluates to a sequence of nodes.

relations.xml

```
<relations>
    <person name="Alice">
      <knows>Bob</knows>
      <knows>Charles</knows>
    </person>
    <person name="Bob">
      <knows>Charles</knows>
    </person>
    <person name="Charles"/>
</relations>
```

3 child elements person with attribute name

Full spec at http://www.w3.org/TR/xpath20/.
Tutorial at http://www.w3schools.com/xpath/default.asp.

XPath

- XPath is used to locate nodes in XML trees
- An XPath expression is a sequence of *steps* separated by /.
- Each *step* evaluates to a sequence of nodes.

relations.xml

```
<relations>
    <person name="Alice">
      <knows>Bob</knows>
      <knows>Charles</knows>
    </person>
    <person name="Bob">
      <knows>Charles</knows>
    </person>
    <person name="Charles"/>
</relations>
```

Each person element can have knows childs

Full spec at http://www.w3.org/TR/xpath20/.
Tutorial at http://www.w3schools.com/xpath/default.asp.

XPath Steps

Step examples

/relations	Selects the root element relations

relations.xml

```
<relations>
    <person name="Alice">
      <knows>Bob</knows>
      <knows>Charles</knows>
    </person>
    <person name="Bob">
      <knows>Charles</knows>
    </person>
    <person name="Charles"/>
</relations>
```

XPath Steps

Step examples

/relations	Selects the root element relations
/relations/person	Selects all person elements that are children of relations

relations.xml

```
<relations>
    <person name="Alice">
      <knows>Bob</knows>
      <knows>Charles</knows>
    </person>
    <person name="Bob">
      <knows>Charles</knows>
    </person>
    <person name="Charles"/>
</relations>
```

XPath Steps

Step examples

/relations	Selects the root element relations
/relations/person	Selects all person elements that are children of relations
//knows	Selects all knows elements (in all the document)

relations.xml

```
<relations>
    <person name="Alice">
      <knows>Bob</knows>
      <knows>Charles</knows>
    </person>
    <person name="Bob">
      <knows>Charles</knows>
    </person>
    <person name="Charles"/>
</relations>
```

XPath Predicates

Predicate examples

/relations/person[3] Selects the third `person` child of `relations`

relations.xml

```
<relations>
    <person name="Alice">
      <knows>Bob</knows>
      <knows>Charles</knows>
    </person>
    <person name="Bob">
      <knows>Charles</knows>
    </person>
    <person name="Charles"/>
</relations>
```

XPath Predicates

Predicate examples

/relations/person[3]	Selects the third person child of relations
/relations/person[position()<3]	Selects the first two person children of relations

relations.xml

```
<relations>
    <person name="Alice">
      <knows>Bob</knows>
      <knows>Charles</knows>
    </person>
    <person name="Bob">
      <knows>Charles</knows>
    </person>
    <person name="Charles"/>
</relations>
```

XPath Predicates

Predicate examples

/relations/person[3]	Selects the third person child of relations
/relations/person[position()<3]	Selects the first two person children of relations
//person[@name='Alice']	Selects all person elements which the value of the name attribute is 'Alice'

relations.xml

```
<relations>
    <person name="Alice">
      <knows>Bob</knows>
      <knows>Charles</knows>
    </person>
    <person name="Bob">
      <knows>Charles</knows>
    </person>
    <person name="Charles"/>
</relations>
```

XQuery

XQuery

- Query language for XML (different requirements than XSLT)
  - functional language
  - typed language
- Superset of XPath
- Overview of the formal semantics (Normalisation rules, Static Typing and Dynamic Evaluation)

XQuery spec: http://www.w3.org/TR/xquery/.
XQuery and XPath Formal Semantics: http://www.w3.org/TR/xquery-semantics/.

Schematic view on XQuery

Prolog:	**P**	`declare namespace` *prefix*="*namespace-URI*"
Body:	**F**	`for` *var* `in` *XPath-expression*
	L	`let` *var* `:=` *XPath-expression*
	W	`where` *XPath-expression*
	O	`order by` *XPath-expression*
Head:	**R**	`return` *XML* + *nested XQuery*

Query Example: *Convert our relations data into RDF/XML*

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
  $nameA in $person/@name
where $nameA = "Alice"
return  <foaf:Person>{$nameA,
  for $nameB in $person/knows
  let $friend := <foaf:Person name="{$nameB}"/>
  order by $nameB
  return <foaf:knows>{$friend}</foaf:knows>
}</foaf:Person>
```

Schematic view on XQuery

Prolog:	**P**	`declare namespace` *prefix*=`"`*namespace-URI*`"`
Body:	**F** **L** **W** **O**	`for` *var* `in` *XPath-expression* `let` *var* `:=` *XPath-expression* `where` *XPath-expression* `order by` *XPath-expression*
Head:	**R**	`return` *XML* + *nested XQuery*

Query Example: *Convert our relations data into RDF/XML*

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
  $nameA in $person/@name
where $nameA = "Alice"
return  <foaf:Person>{$nameA,
  for $nameB in $person/knows
  let $friend := <foaf:Person name="{$nameB}"/>
  order by $nameB
  return <foaf:knows>{$friend}</foaf:knows>
}</foaf:Person>
```

Schematic view on XQuery

Prolog:	**P**	declare namespace *prefix*="*namespace-URI*"
Body:	**F**	for *var* in *XPath-expression*
	L	let *var* := *XPath-expression*
	W	where *XPath-expression*
	O	order by *XPath-expression*
Head:	**R**	return *XML* + *nested XQuery*

Query Example: *Convert our relations data into RDF/XML*

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
  $nameA in $person/@name
where $nameA = "Alice"
return  <foaf:Person>{$nameA,
  for $nameB in $person/knows
  let $friend := <foaf:Person name="{$nameB}"/>
  order by $nameB
  return <foaf:knows>{$friend}</foaf:knows>
}</foaf:Person>
```

Schematic view on XQuery

Prolog:	**P**	`declare namespace` *prefix*=`"`*namespace-URI*`"`
Body:	**F**	`for` *var* `in` *XPath-expression*
	L	`let` *var* `:=` *XPath-expression*
	W	`where` *XPath-expression*
	O	`order by` *XPath-expression*
Head:	**R**	`return` *XML + nested XQuery*

Query Example: *Convert our relations data into RDF/XML*

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
  $nameA in $person/@name
where $nameA = "Alice"
return  <foaf:Person>{$nameA,
  for $nameB in $person/knows
  let $friend := <foaf:Person name="{$nameB}"/>
  order by $nameB
  return <foaf:knows>{$friend}</foaf:knows>
}</foaf:Person>
```

Schematic view on XQuery

Prolog:	**P**	`declare namespace` *prefix*=`"`*namespace-URI*`"`
Body:	**F**	`for` *var* `in` *XPath-expression*
	L	`let` *var* `:=` *XPath-expression*
	W	`where` *XPath-expression*
	O	`order by` *XPath-expression*
Head:	**R**	`return` *XML + nested XQuery*

Query Example: *Convert our relations data into RDF/XML*

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
  $nameA in $person/@name
where $nameA = "Alice"
return  <foaf:Person>{$nameA,
  for $nameB in $person/knows
  let $friend := <foaf:Person name="{$nameB}"/>
  order by $nameB
  return <foaf:knows>{$friend}</foaf:knows>
}</foaf:Person>
```

Schematic view on XQuery

Prolog:	**P**	`declare namespace` *prefix*=`"`*namespace-URI*`"`
Body:	**F**	`for` *var* `in` *XPath-expression*
	L	`let` *var* `:=` *XPath-expression*
	W	`where` *XPath-expression*
	O	`order by` *XPath-expression*
Head:	**R**	`return` *XML* + *nested XQuery*

Query Example: *Convert our relations data into RDF/XML*

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
  $nameA in $person/@name
where $nameA = "Alice"
return  <foaf:Person>{$nameA,
  for $nameB in $person/knows
  let $friend := <foaf:Person name="{$nameB}"/>
  order by $nameB
  return <foaf:knows>{$friend}</foaf:knows>
}</foaf:Person>
```

Schematic view on XQuery

Prolog:	**P**	`declare namespace` *prefix*="*namespace-URI*"
Body:	**F**	`for` *var* `in` *XPath-expression*
	L	`let` *var* `:=` *XPath-expression*
	W	`where` *XPath-expression*
	O	`order by` *XPath-expression*
Head:	**R**	`return` *XML* + *nested XQuery*

Query Example: *Convert our relations data into RDF/XML*

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
  $nameA in $person/@name
where $nameA = "Alice"
return  <foaf:Person>{$nameA,
  for $nameB in $person/knows
  let $friend := <foaf:Person name="{$nameB}"/>
  order by $nameB
  return <foaf:knows>{$friend}</foaf:knows>
}</foaf:Person>
```

Schematic view on XQuery

Prolog:	**P**	declare namespace *prefix*="*namespace-URI*"
Body:	**F**	for *var* in *XPath-expression*
	L	let *var* := *XPath-expression*
	W	where *XPath-expression*
	O	order by *XPath-expression*
Head:	**R**	return *XML* + *nested XQuery*

Query Example: *Convert our relations data into RDF/XML*

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
  $nameA in $person/@name
where $nameA = "Alice"
return  <foaf:Person>{$nameA,
  for $nameB in $person/knows
  let $friend := <foaf:Person name="{$nameB}"/>
  order by $nameB
  return <foaf:knows>{$friend}</foaf:knows>
}</foaf:Person>
```

Schematic view on XQuery

Prolog:	**P**	`declare namespace` *prefix*="*namespace-URI*"
Body:	**F**	`for` *var* `in` *XPath-expression*
	L	`let` *var* `:=` *XPath-expression*
	W	`where` *XPath-expression*
	O	`order by` *XPath-expression*
Head:	**R**	`return` *XML + nested XQuery*

Query Example: *Convert our relations data into RDF/XML*

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
  $nameA in $person/@name
where $nameA = "Alice"
return  <foaf:Person>{$nameA,
  for $nameB in $person/knows
  let $friend := <foaf:Person name="{$nameB}"/>
  order by $nameB
  return <foaf:knows>{$friend}</foaf:knows>
}</foaf:Person>
```

Schematic view on XQuery

Prolog:	**P**	declare namespace *prefix*="*namespace-URI*"
Body:	**F**	for *var* in *XPath-expression*
	L	let *var* := *XPath-expression*
	W	where *XPath-expression*
	O	order by *XPath-expression*
Head:	**R**	return *XML* + *nested XQuery*

Query result

```
<foaf:Person xmlns:foaf="http://xmlns.com/foaf/0.1/" name="Alice">
   <foaf:knows>
      <foaf:Person name="Bob"/>
   </foaf:knows>
   <foaf:knows>
      <foaf:Person name="Charles"/>
   </foaf:knows>
</foaf:Person>
```

Positional variable at

Query example: *add an* `id` *attribute to the relations data*

```
for $person at $pos in doc("relations.xml")//person
return <person id="{$pos}">
          {$person/@*, $person/*}
       </person>
```

`$pos` refers to the position of `$person` in the `for` expression

Positional variable at

Query example: *add an id attribute to the relations data*

```
for $person at $pos in doc("relations.xml")//person
return <person id="{$pos}">
          {$person/@*, $person/*}
       </person>
```

$pos refers to the position of $person in the for expression

Query result

```
<person id="1" name="Alice">
   <knows>Bob</knows>
   <knows>Charles</knows>
</person>
<person id="2" name="Bob">
   <knows>Charles</knows>
</person>
<person id="3" name="Charles"/>
```

XQuery Formal semantics

Normalisation rules

Normalisation rules are rewriting rules that translate XQuery into a simplified version (XQuery Core).

XQuery Formal semantics

Normalisation rules

Normalisation rules are rewriting rules that translate XQuery into a simplified version (XQuery Core).

Rule application

1. Static Analysis: Apply normalisation rules and static type analysis
2. Dynamic Evaluation Rules: evaluate expressions

XQuery Formal semantics

Normalisation rules

Normalisation rules are rewriting rules that translate XQuery into a simplified version (XQuery Core).

Rule application

1. Static Analysis: Apply normalisation rules and static type analysis
2. Dynamic Evaluation Rules: evaluate expressions

Environments

statEnv contains information needed for performing static type analysis. E.g. `varType`, `funcType`, . . .

XQuery Formal semantics

Normalisation rules

Normalisation rules are rewriting rules that translate XQuery into a simplified version (XQuery Core).

Rule application

1. Static Analysis: Apply normalisation rules and static type analysis
2. Dynamic Evaluation Rules: evaluate expressions

Environments

statEnv contains information needed for performing static type analysis. E.g. `varType`, `funcType`, . . .

dynEnv contains information needed for the evaluation of expressions. E.g. `varValue`, . . .

Semantics - Normalisation rules example

for Example

```
for $i in (1, 2),
    $j in (3, 4)
return
  <pair>{ ($i,$j) }</pair>
```

Semantics - Normalisation rules example

for Example

```
for $i in (1, 2),
    $j in (3, 4)
return
  <pair>{ ($i,$j) }</pair>
```

for Normalised example

```
for $i in (1, 2) return
  for $j in (3, 4) return
      <pair>{ ($i,$j) }</pair>
```

Semantics - Normalisation rules example

For Normalisation

$$\left[\!\!\left[\begin{array}{l}\texttt{for } \$\mathit{VarName}_1 \texttt{ in } \mathit{Expr}_1, \cdots, \\ \quad\quad \$\mathit{VarName}_n \texttt{ in } \mathit{Expr}_n \\ \mathit{ReturnClause}\end{array}\right]\!\!\right]_{\mathit{Expr}}$$

$$==$$

$$\begin{array}{l}\texttt{for } \$\mathit{VarName}_1 \texttt{ in } [\mathit{Expr}_1]_{\mathit{Expr}} \texttt{ return} \\ \ldots \\ \texttt{for } \$\mathit{VarName}_n \texttt{ in } [\mathit{Expr}_n]_{\mathit{Expr}}\ [\mathit{ReturnClause}]_{\mathit{Expr}}\end{array}$$

Semantics - Static typing example

For Static Type Analysis

$$\frac{\begin{array}{c}\mathrm{statEnv} \vdash \mathit{Expr}_1 : \mathit{Type}_1 \\ \mathrm{statEnv} + \mathrm{varType}(\mathit{Variable} \Rightarrow \mathit{Type}_1) \vdash \mathit{Expr}_2 : \mathit{Type}_2\end{array}}{\mathrm{statEnv} \vdash \begin{array}{l}\texttt{for } \$\mathit{Variable} \texttt{ in } \mathit{Expr}_1 \\ \texttt{return } \mathit{Expr}_2 : \mathit{Type}_2 \cdot \mathrm{quantifier}(\mathit{Type}_1)\end{array}}$$

Semantics - Static typing example

: means Expr_1 is of type Type_1

For Static Type Analysis

$$\frac{\begin{array}{c}\mathrm{statEnv} \vdash \mathit{Expr}_1 : \mathit{Type}_1 \\ \mathrm{statEnv} + \mathrm{varType}(\mathit{Variable} \Rightarrow \mathit{Type}_1) \vdash \mathit{Expr}_2 : \mathit{Type}_2\end{array}}{\mathrm{statEnv} \vdash \begin{array}{l}\texttt{for}\ \$\mathit{Variable}\ \texttt{in}\ \mathit{Expr}_1 \\ \texttt{return}\ \mathit{Expr}_2 : \mathit{Type}_2 \cdot \mathrm{quantifier}(\mathit{Type}_1)\end{array}}$$

Semantics - Static typing example

+ means extend the environment

: means $Expr_1$ is of type $Type_1$

For Static Type Analysis

$$\frac{\begin{array}{c}\mathrm{statEnv} \vdash \mathit{Expr}_1 : \mathit{Type}_1 \\ \mathrm{statEnv} + \mathrm{varType}(\mathit{Variable} \Rightarrow \mathit{Type}_1) \vdash \mathit{Expr}_2 : \mathit{Type}_2\end{array}}{\mathrm{statEnv} \vdash \begin{array}{l}\texttt{for } \$\mathit{Variable} \texttt{ in } \mathit{Expr}_1 \\ \texttt{return } \mathit{Expr}_2 : \mathit{Type}_2 \cdot \mathrm{quantifier}(\mathit{Type}_1)\end{array}}$$

Semantics - Static typing example

+ means extend the environment

: means Expr_1 is of type Type_1

For Static Type Analysis

$$\frac{\begin{array}{c}\text{statEnv} \vdash \mathit{Expr}_1 : \mathit{Type}_1 \\ \text{statEnv} + \text{varType}(\mathit{Variable} \Rightarrow \mathit{Type}_1) \vdash \mathit{Expr}_2 : \mathit{Type}_2\end{array}}{\text{statEnv} \vdash \begin{array}{l}\texttt{for } \$\mathit{Variable} \texttt{ in } \mathit{Expr}_1 \\ \texttt{return } \mathit{Expr}_2 : \mathit{Type}_2 \cdot \text{quantifier}(\mathit{Type}_1)\end{array}}$$

`quantifier` estimates the number of solutions: `*`, `+` or `?`

Semantics - Dynamic evaluation rules

Simple `for` example

```
for $i in (1, 2) return $i+1
```

Semantics - Dynamic evaluation rules

For each result in the expression

Simple for example

```
for $i in (1, 2) return $i+1
```

Semantics - Dynamic evaluation rules

For each result in the expression

Simple for example

```
for $i in (1, 2) return $i+1
```

Variable $i is assigned the corresponding value

Semantics - Dynamic evaluation rules

Simple for example

```
for $i in (1, 2) return $i+1
```

For each result in the expression

Variable $i is assigned the corresponding value

Return expresion is evaluated

Semantics - Dynamic evaluation rules

Simple for example

```
for $i in (1, 2) return $i+1
```

For Dynamic Evaluation (Simplified)

$$\frac{\begin{array}{c}\mathrm{dynEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Item}_1, \ldots, \mathit{Item}_n \\ \mathrm{dynEnv} + \mathrm{varValue}(\mathit{Variable} \Rightarrow \mathit{Item}_1) \vdash \mathit{Expr}_2 \Rightarrow \mathit{Value}_1 \\ \cdots \\ \mathrm{dynEnv} + \mathrm{varValue}(\mathit{Variable} \Rightarrow \mathit{Item}_n) \vdash \mathit{Expr}_2 \Rightarrow \mathit{Value}_n\end{array}}{\mathrm{dynEnv} \vdash \begin{array}{l}\texttt{for } \$\mathit{Variable} \texttt{ in } \mathit{Expr}_1 \\ \texttt{return } \mathit{Expr}_2 \Rightarrow \mathit{Value}_1, \cdots, \mathit{Value}_n\end{array}}$$

Semantics - Dynamic evaluation rules

Simple for example

```
for $i in (1, 2) return $i+1
```

$\Rightarrow$ means $Expr_1$ evaluates to the sequence $Item_1, \ldots, Item_n$

For Dynamic Evaluation (Simplified)

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Item}_1, \ldots, \mathit{Item}_n \\ \text{dynEnv} + \text{varValue}(\mathit{Variable} \Rightarrow \mathit{Item}_1) \vdash \mathit{Expr}_2 \Rightarrow \mathit{Value}_1 \\ \ldots \\ \text{dynEnv} + \text{varValue}(\mathit{Variable} \Rightarrow \mathit{Item}_n) \vdash \mathit{Expr}_2 \Rightarrow \mathit{Value}_n\end{array}}{\text{dynEnv} \vdash \begin{array}{l}\texttt{for } \$\mathit{Variable} \texttt{ in } \mathit{Expr}_1 \\ \texttt{return } \mathit{Expr}_2 \Rightarrow \mathit{Value}_1, \cdots, \mathit{Value}_n\end{array}}$$

Semantics - Dynamic evaluation rules

Simple `for` example

```
for $i in (1, 2) return $i+1
```

For each *Item*$_i$, add *Variable* ⇒ *Item*$_i$ to dynEnv and evaluate *Expr*$_2$

⇒ means *Expr*$_1$ evaluates to the sequence *Item*$_1$, . . . , *Item*$_n$

For Dynamic Evaluation (Simplified)

$$\frac{\begin{array}{c}\mathrm{dynEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Item}_1, \ldots, \mathit{Item}_n \\ \mathrm{dynEnv} + \mathrm{varValue}(\mathit{Variable} \Rightarrow \mathit{Item}_1) \vdash \mathit{Expr}_2 \Rightarrow \mathit{Value}_1 \\ \cdots \\ \mathrm{dynEnv} + \mathrm{varValue}(\mathit{Variable} \Rightarrow \mathit{Item}_n) \vdash \mathit{Expr}_2 \Rightarrow \mathit{Value}_n\end{array}}{\mathrm{dynEnv} \vdash \begin{array}{l}\texttt{for } \$\mathit{Variable} \texttt{ in } \mathit{Expr}_1 \\ \texttt{return } \mathit{Expr}_2 \Rightarrow \mathit{Value}_1, \cdots, \mathit{Value}_n\end{array}}$$

Outline

Overview

XPath & XQuery
- XPath
- XQuery
- XQuery Semantics

SPARQL

XSPARQL
- Syntax
- Semantics
- Implementation
- XSPARQL Features
- Query examples

SPARQL (in 3 slides)

SPARQL

- Query language for RDF

SPARQL spec: http://www.w3.org/TR/rdf-sparql-query/.
SPARQL 1.1 Tutorial:
http://polleres.net/presentations/20101019SPARQL1.1Tutorial.pdf.

SPARQL (in 3 slides)

SPARQL

- Query language for RDF
- RDF represents data as *triples*: Subject, Predicate, Object. An *RDF Graph* is a set of triples.

SPARQL spec: http://www.w3.org/TR/rdf-sparql-query/.
SPARQL 1.1 Tutorial:
http://polleres.net/presentations/20101019SPARQL1.1Tutorial.pdf.

SPARQL (in 3 slides)

SPARQL

- Query language for RDF
- RDF represents data as *triples*: Subject, Predicate, Object. An *RDF Graph* is a set of triples.
- SPARQL queries RDF data by *pattern matching*: given a set of triple patterns, finds the corresponding triples in the input graph

SPARQL spec: http://www.w3.org/TR/rdf-sparql-query/.
SPARQL 1.1 Tutorial:
http://polleres.net/presentations/20101019SPARQL1.1Tutorial.pdf.

SPARQL (in 3 slides)

SPARQL

- Query language for RDF
- RDF represents data as *triples*: Subject, Predicate, Object. An *RDF Graph* is a set of triples.
- SPARQL queries RDF data by *pattern matching*: given a set of triple patterns, finds the corresponding triples in the input graph
- Actually, matched against the *scoping graph*: a graph equivalent to the input graph but does not share any blank nodes with it or the query.

SPARQL spec: http://www.w3.org/TR/rdf-sparql-query/.
SPARQL 1.1 Tutorial:
http://polleres.net/presentations/20101019SPARQL1.1Tutorial.pdf.

Schematic view on SPARQL

Prolog:	**P**	`prefix` *prefix*: <*namespace-URI*>
Head:	**C**	`construct` { *template* } `select` *variableList*
Body:	**D** **W** **M**	`from` / `from named` <*dataset-URI*> `where` { *pattern* } `order by` *expression* `limit` *integer* > 0 `offset` *integer* > 0

Query Example: *Convert between different RDF vocabularies*

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

construct { $X foaf:name $FN.}
from <vCard.ttl>
where { $X vc:FN $FN .}
order by $FN
limit 1 offset 1
```

Schematic view on SPARQL

Prolog:	**P**	prefix *prefix*: <*namespace-URI*>
Head:	**C**	construct { *template* }
		select *variableList*
Body:	**D**	from / from named <*dataset-URI*>
	W	where { *pattern* }
	M	order by *expression*
		limit *integer* > 0
		offset *integer* > 0

Query Example: *Convert between different RDF vocabularies*

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

construct { $X foaf:name $FN.}
from <vCard.ttl>
where { $X vc:FN $FN .}
order by $FN
limit 1 offset 1
```

Schematic view on SPARQL

Prolog:	P	prefix *prefix*: <*namespace-URI*>
Head:	C	construct { *template* }
		select *variableList*
Body:	D	from / from named <*dataset-URI*>
	W	where { *pattern* }
	M	order by *expression*
		limit *integer* > 0
		offset *integer* > 0

Query Example: *Convert between different RDF vocabularies*

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

construct { $X foaf:name $FN.}
from <vCard.ttl>
where { $X vc:FN $FN .}
order by $FN
limit 1 offset 1
```

Schematic view on SPARQL

Prolog:	**P**	`prefix` *prefix*: <*namespace-URI*>
Head:	**C**	`construct` { *template* } `select` *variableList*
Body:	**D** **W** **M**	`from` / `from named` <*dataset-URI*> `where` { *pattern* } `order by` *expression* `limit` *integer* > 0 `offset` *integer* > 0

Query Example: *Convert between different RDF vocabularies*

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

construct { $X foaf:name $FN.}
from <vCard.ttl>
where { $X vc:FN $FN .}
order by $FN
limit 1 offset 1
```

Schematic view on SPARQL

Prolog:	**P**	`prefix` *prefix*: <*namespace-URI*>
Head:	**C**	`construct` { *template* } `select` *variableList*
Body:	**D**	`from` / `from named` <*dataset-URI*>
	W	`where` { *pattern* }
	M	`order by` *expression* `limit` *integer* > 0 `offset` *integer* > 0

Query Example: *Convert between different RDF vocabularies*

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

construct { $X foaf:name $FN.}
from <vCard.ttl>
where { $X vc:FN $FN .}
order by $FN
limit 1 offset 1
```

Schematic view on SPARQL

Prolog:	**P**	prefix *prefix*: <*namespace-URI*>
Head:	**C**	construct { *template* } select *variableList*
Body:	**D** **W** **M**	from / from named <*dataset-URI*> where { *pattern* } order by *expression* limit *integer* > 0 offset *integer* > 0

Query Example: *Convert between different RDF vocabularies*

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

construct { $X foaf:name $FN.}
from <vCard.ttl>
where { $X vc:FN $FN .}
order by $FN
limit 1 offset 1
```

Schematic view on SPARQL

Prolog:	**P**	`prefix` *prefix*: <*namespace-URI*>
Head:	**C**	`construct` { *template* } `select` *variableList*
Body:	**D** **W** **M**	`from` / `from named` <*dataset-URI*> `where` { *pattern* } `order by` *expression* `limit` *integer* > 0 `offset` *integer* > 0

Query Example: *Convert between different RDF vocabularies*

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

construct { $X foaf:name $FN.}
from <vCard.ttl>
where { $X vc:FN $FN .}
order by $FN
limit 1 offset 1
```

Schematic view on SPARQL

Prolog:	**P**	`prefix` *prefix*: <*namespace-URI*>
Head:	**C**	`construct` { *template* } `select` *variableList*
Body:	**D** **W** **M**	`from` / `from named` <*dataset-URI*> `where` { *pattern* } `order by` *expression* `limit` *integer* > 0 `offset` *integer* > 0

Query Example: *Convert between different RDF vocabularies*

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

construct { $X foaf:name $FN.}
from <vCard.ttl>
where { $X vc:FN $FN .}
order by $FN
limit 1 offset 1
```

Schematic view on SPARQL

Prolog:	**P**	prefix *prefix*: <*namespace-URI*>
Head:	**C**	construct { *template* } select *variableList*
Body:	**D** **W** **M**	from / from named <*dataset-URI*> where { *pattern* } order by *expression* limit *integer* > 0 offset *integer* > 0

Query Example: *Convert between different RDF vocabularies*

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

select $X $FN
from <vCard.ttl>
where { $X vc:FN $FN .}
order by $FN
limit 1 offset 1
```

SPARQL `select` solutions

- Solutions for SPARQL `select` queries are substitutions for the variables present in the head (*variableList*)

SPARQL `select` solutions

- Solutions for SPARQL `select` queries are substitutions for the variables present in the head (*variableList*)
- Can be represented as XML

SPARQL XML Results Format (previous query)

```
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="X"/>
    <variable name="FN"/>
  </head>
  <results>
    <result>
      <binding name="X"><bnode>b0</bnode></binding>
      <binding name="FN"><literal>Nuno Lopes</literal></binding>
    </result>
  </results>
</sparql>
```

Outline

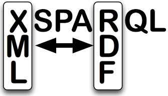
Overview

XPath & XQuery
- XPath
- XQuery
- XQuery Semantics

SPARQL

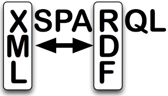
XSPARQL
- Syntax
- Semantics
- Implementation
- XSPARQL Features
- Query examples

XSPARQL



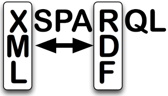
- Transformation language
- Consume and generate XML and RDF

XSPARQL



- Transformation language
- Consume and generate XML and RDF
- Syntactic extension of XQuery

XSPARQL



- Transformation language
- Consume and generate XML and RDF
- Syntactic extension of XQuery
- With a formally defined semantics (based on the XQuery semantics)

Outline

Overview

XPath & XQuery

- XPath
- XQuery
- XQuery Semantics

SPARQL

XSPARQL

- Syntax
- Semantics
- Implementation
- XSPARQL Features
- Query examples

XSPARQL: Combining XQuery with SPARQL

Prolog:	**P**	declare namespace *prefix*="*namespace-URI*" or prefix *prefix*: <*namespace-URI*>	
Body:	**F**	for *var* in *XPath-expression*	
	L	let *var* := *XPath-expression*	
	W	where *XPath-expression*	
	O	order by *expression*	**or**
	F'	for *varlist*	
	D	from / from named <*dataset-URI*>	
	W	where {*pattern* }	
	M	order by *expression* limit *integer* > 0 offset *integer* > 0	
Head:	**C**	construct { *template (with nested XSPARQL)* }	**or**
	R	return *XML + nested XSPARQL*	

XSPARQL: Combining XQuery with SPARQL

prefix declarations

Prolog:	**P**	declare namespace *prefix*="*namespace-URI*" or prefix *prefix*: <*namespace-URI*>	
Body:	**F**	for *var* in *XPath-expression*	
	L	let *var* := *XPath-expression*	
	W	where *XPath-expression*	
	O	order by *expression*	**or**
	F'	for *varlist*	
	D	from / from named <*dataset-URI*>	
	W	where { *pattern* }	
	M	order by *expression*	
		limit *integer* > 0	
		offset *integer* > 0	
Head:	**C**	construct { *template (with nested XSPARQL)* }	**or**
	R	return *XML* + *nested XSPARQL*	

XSPARQL: Combining XQuery with SPARQL

Data input (XML or RDF)

Prolog:	**P**	declare namespace *prefix*="*namespace-URI*" or prefix *prefix*: <*namespace-URI*>	
Body:	**F** **L** **W** **O**	for *var* in *XPath-expression* let *var* := *XPath-expression* where *XPath-expression* order by *expression*	**or**

	F'
	D
	W
	M
		for *varlist*
		from / from named <*dataset-URI*>
		where {*pattern* }
		order by *expression*
		limit *integer* > 0
		offset *integer* > 0	

Head:	**C**	construct { *template (with nested XSPARQL)* }	**or**
	R	return *XML* + *nested XSPARQL*	

XSPARQL: Combining XQuery with SPARQL

Prolog:	**P**	`declare namespace` *prefix*="*namespace-URI*" or `prefix` *prefix*: <*namespace-URI*>	
Body:	**F** **L** **W** **O**	`for` *var* `in` *XPath-expression* `let` *var* := *XPath-expression* `where` *XPath-expression* `order by` *expression*	**or**
	F' **D** **W** **M**	`for` *varlist* `from` / `from named` <*dataset-URI*> `where` { *pattern* } `order by` *expression* `limit` *integer* > 0 `offset` *integer* > 0	

Head:	**C**	`construct`
		{ *template (with nested XSPARQL)* }
			or

	R	`return` *XML + nested XSPARQL*	

Data output (XML or RDF)

XSPARQL: Combining XQuery with SPARQL

XQuery or SPARQL `prefix` declarations

Prolog:	**P**	`declare namespace` *prefix*="*namespace-URI*" or `prefix` *prefix*: <*namespace-URI*>	
Body:	**F**	`for` *var* `in` *XPath-expression*	
	L	`let` *var* := *XPath-expression*	
	W	`where` *XPath-expression*	
	O	`order by` *expression*	**or**
	F'	`for` *varlist*	
	D	`from` / `from named` <*dataset-URI*>	
	W	`where` {*pattern* }	
	M	`order by` *expression*	
		`limit` *integer* > 0	
		`offset` *integer* > 0	

Head:	**C**	`construct`
		{ *template (with nested XSPARQL)* }
			or

	R	`return` *XML* + *nested XSPARQL*	

XSPARQL: Combining XQuery with SPARQL

Any XQuery query

Prolog:	P	`declare namespace` *prefix*="*namespace-URI*" or `prefix` *prefix*: <*namespace-URI*>	
Body:	F	`for` *var* `in` *XPath-expression*	
	L	`let` *var* := *XPath-expression*	
	W	`where` *XPath-expression*	
	O	`order by` *expression*	or
	F'	`for` *varlist*	
	D	`from` / `from named` <*dataset-URI*>	
	W	`where` {*pattern* }	
	M	`order by` *expression*	
		`limit` *integer* > 0	
		`offset` *integer* > 0	

Head:	C	`construct`
		{ *template (with nested XSPARQL)* }
			or

	R	`return` *XML* + *nested XSPARQL*	

XSPARQL: Combining XQuery with SPARQL

SPARQLForClause represents a SPARQL query

Prolog:	**P**	declare namespace *prefix*="*namespace-URI*" or prefix *prefix*: <*namespace-URI*>	
Body:	**F**	for *var* in *XPath-expression*	
	L	let *var* := *XPath-expression*	
	W	where *XPath-expression*	
	O	order by *expression*	**or**
	F'	for *varlist*	
	D	from / from named <*dataset-URI*>	
	W	where {*pattern* }	
	M	order by *expression*	
		limit *integer* > 0	
		offset *integer* > 0	
Head:	**C**	construct { *template (with nested XSPARQL)* }	**or**
	R	return *XML + nested XSPARQL*	

XSPARQL: Combining XQuery with SPARQL

Prolog:	**P**	`declare namespace` *prefix*="*namespace-URI*" or `prefix` *prefix*: <*namespace-URI*>	
Body:	**F**	`for` *var* `in` *XPath-expression*	
	L	`let` *var* := *XPath-expression*	
	W	`where` *XPath-expression*	
	O	`order by` *expression*	**or**
	F'	`for` *varlist*	
	D	`from` / `from named` <*dataset-URI*>	
	W	`where` {*pattern* }	
	M	`order by` *expression*	
		`limit` *integer* > 0	
		`offset` *integer* > 0	

Head:	**C**	`construct`
		{ *template (with nested XSPARQL)* }
			or

	R	`return` *XML + nested XSPARQL*	

`construct` creates RDF output

Query Example - Lifting

Convert our relations data into RDF

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
    $nameA in $person/@name,
    $nameB in $person/knows
construct { [ foaf:name {data($nameA)}; a foaf:Person ] foaf:knows
            [ foaf:name {data($nameB)}; a foaf:Person ]. }
```

Query Example - Lifting

Convert our relations data into RDF

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
    $nameA in $person/@name,
    $nameB in $person/knows
construct { [ foaf:name {data($nameA)}; a foaf:Person ] foaf:knows
            [ foaf:name {data($nameB)}; a foaf:Person ]. }
```

XQuery for data selection

Query Example - Lifting

Convert our relations data into RDF

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
    $nameA in $person/@name,
    $nameB in $person/knows
construct { [ foaf:name {data($nameA)}; a foaf:Person ] foaf:knows
            [ foaf:name {data($nameB)}; a foaf:Person ]. }
```

construct clause generates RDF

Query Example - Lifting

Convert our relations data into RDF

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
    $nameA in $person/@name,
    $nameB in $person/knows
construct { [ foaf:name {data($nameA)}; a foaf:Person ] foaf:knows
            [ foaf:name {data($nameB)}; a foaf:Person ]. }
```

Query Example - Lifting

Convert our relations data into RDF

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
    $nameA in $person/@name,
    $nameB in $person/knows
construct { [ foaf:name {data($nameA)}; a foaf:Person ] foaf:knows
            [ foaf:name {data($nameB)}; a foaf:Person ]. }
```

Nesting produces an RDF literal

Query Example - Lifting

Convert our relations data into RDF

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";

for $person in doc("relations.xml")//person,
    $nameA in $person/@name,
    $nameB in $person/knows
construct { [ foaf:name {data($nameA)}; a foaf:Person ] foaf:knows
            [ foaf:name {data($nameB)}; a foaf:Person ]. }
```

Query result

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
[ foaf:name "Alice"; a foaf:Person; foaf:knows
  [ foaf:name "Bob"; a foaf:Person ] ].
[ foaf:name "Alice"; a foaf:Person; foaf:knows
  [ foaf:name "Charles"; a foaf:Person ] ].
[ foaf:name "Bob"; a foaf:Person; foaf:knows
  [ foaf:name "Charles"; a foaf:Person ] ].
```

Nesting in `construct` clauses

Nesting operators

`{Expr}` The result of evaluating `Expr` will be an RDF literal

Nesting in construct clauses

Nesting operators

{Expr}	The result of evaluating Expr will be an RDF literal
_:{Expr}	Same but for RDF blank nodes
<{Expr}>	and IRIs

Query Example - Lifting (II)

Convert our relations data into RDF

```
declare namespace foaf="http://xmlns.com/foaf/0.1/";
let $persons := doc("relations.xml")//person
let $ids := data($persons/@name)
for $p in $persons
  let $id := fn:index-of($ids, $p/@name)
  construct { _:b{$id} a foaf:Person; foaf:name {data($p/@name)}.
              { for $k in $p/knows
                let $kid := fn:index-of($ids, $k)
                construct { _:b{$id}  foaf:knows _:b{$kid} } } }
```

Query Example - Lifting (II)

Convert our relations data into RDF

```
declare namespace foaf="http://xmlns.com/foaf/0.1/";
let $persons := doc("relations.xml")//person
let $ids := data($persons/@name)
for $p in $persons
  let $id := fn:index-of($ids, $p/@name)
  construct { _:b{$id} a foaf:Person; foaf:name {data($p/@name)}.
              { for $k in $p/knows
                let $kid := fn:index-of($ids, $k)
                construct { _:b{$id}  foaf:knows _:b{$kid} } } }
```

Keep person identifiers

Query Example - Lifting (II)

Convert our relations data into RDF

```
declare namespace foaf="http://xmlns.com/foaf/0.1/";
let $persons := doc("relations.xml")//person
let $ids := data($persons/@name)
for $p in $persons
  let $id := fn:index-of($ids, $p/@name)
  construct { _:b{$id} a foaf:Person; foaf:name {data($p/@name)}.
              { for $k in $p/knows
                let $kid := fn:index-of($ids, $k)
                construct { _:b{$id}  foaf:knows _:b{$kid} } } }
```

For each person lookup their id

Query Example - Lifting (II)

Convert our relations data into RDF

```
declare namespace foaf="http://xmlns.com/foaf/0.1/";
let $persons := doc("relations.xml")//person
let $ids := data($persons/@name)
for $p in $persons
  let $id := fn:index-of($ids, $p/@name)
  construct { _:b{$id} a foaf:Person; foaf:name {data($p/@name)}.
                 { for $k in $p/knows
                   let $kid := fn:index-of($ids, $k)
                   construct { _:b{$id}  foaf:knows _:b{$kid} } } }
```

The same for each person then know

Query Example - Lifting (II)

Convert our relations data into RDF

```
declare namespace foaf="http://xmlns.com/foaf/0.1/";
let $persons := doc("relations.xml")//person
let $ids := data($persons/@name)
for $p in $persons
  let $id := fn:index-of($ids, $p/@name)
  construct { _:b{$id} a foaf:Person; foaf:name {data($p/@name)}.
              { for $k in $p/knows
                let $kid := fn:index-of($ids, $k)
                construct { _:b{$id}  foaf:knows _:b{$kid} } } }
```

Query result (reformatted output)

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:b1 a foaf:Person; foaf:name "Alice"; foaf:knows _:b2, _:b3 .
_:b2 a foaf:Person; foaf:name "Bob"; foaf:knows _:b3 .
_:b3 a foaf:Person; foaf:name "Charles" .
```

Query Example - Lowering

Convert FOAF RDF data into XML

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
         { for $FName from <relations.rdf>
           where { $Person foaf:knows $Friend .
                   $Person foaf:name $Name.
                   $Friend foaf:name $FName. }
           return <knows> { $FName }</knows>}
         </person>}
</relations>
```

Query Example - Lowering

Convert FOAF RDF data into XML

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
       { for $FName from <relations.rdf>
         where { $Person foaf:knows $Friend .
                 $Person foaf:name $Name.
                 $Friend foaf:name $FName. }
         return <knows> { $FName }</knows>}
       </person>}
</relations>
```

XML construction

Query Example - Lowering

Convert FOAF RDF data into XML

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
       { for $FName from <relations.rdf>
         where { $Person foaf:knows $Friend .
                 $Person foaf:name $Name.
                 $Friend foaf:name $FName. }
         return <knows> { $FName }</knows>}
       </person>}
</relations>
```

SPARQL for query: *"Give me persons and their names"*

Query Example - Lowering

Convert FOAF RDF data into XML

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
       { for $FName from <relations.rdf>
         where { $Person foaf:knows $Friend .
                 $Person foaf:name $Name.
                 $Friend foaf:name $FName. }
         return <knows> { $FName }</knows>}
       </person>}
</relations>
```

SPARQL variables are $-prefixed

SPARQL for query: *“Give me persons and their names”*

Query Example - Lowering

Convert FOAF RDF data into XML

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
       { for $FName from <relations.rdf>
         where { $Person foaf:knows $Friend .
                 $Person foaf:name $Name.
                 $Friend foaf:name $FName. }
         return <knows> { $FName }</knows>}
       </person>}
</relations>
```

XML construction

Query Example - Lowering

Convert FOAF RDF data into XML

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
       { for $FName from <relations.rdf>
         where { $Person foaf:knows $Friend .
                 $Person foaf:name $Name.
                 $Friend foaf:name $FName. }
         return <knows> { $FName }</knows>}
       </person>}
</relations>
```

SPARQL for query: *"Give me the persons each one knows"*

Query Example - Lowering

Convert FOAF RDF data into XML

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
       { for $FName from <relations.rdf>
         where { $Person foaf:knows $Friend .
                 $Person foaf:name $Name.
                 $Friend foaf:name $FName. }
         return <knows> { $FName }</knows>}
       </person>}
</relations>
```

SPARQL for query: *"Give me the persons each one knows"*

$Person and $Name instantiated by the outer loop

Query Example - Lowering

Convert FOAF RDF data into XML

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
       { for $FName from <relations.rdf>
         where { $Person foaf:knows $Friend .
                 $Person foaf:name $Name.
                 $Friend foaf:name $FName. }
         return <knows> { $FName }</knows>}
       </person>}
</relations>
```

XML construction

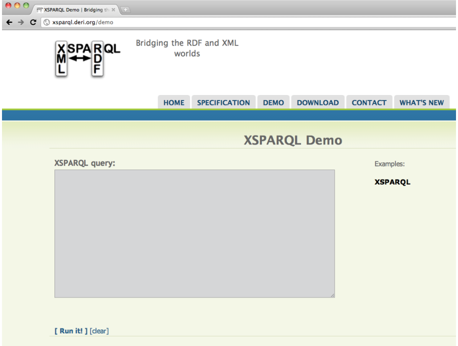
Query Example - Lowering result

Query result

```
<relations>
   <person name="Alice">
      <knows>Charles</knows>
      <knows>Bob</knows>
   </person>
   <person name="Bob">
      <knows>Charles</knows>
   </person>
   <person name="Charles"/>
</relations>
```

Online Demo

Online Demo at: http://xsparql.deri.org/demo/



XSPARQL Demo | Bridging th

xsparql.deri.org/demo

XSPARQL XML RDF

Bridging the RDF and XML worlds

HOME SPECIFICATION DEMO DOWNLOAD CONTACT WHAT'S NEW

XSPARQL Demo

XSPARQL query:

Examples:

XSPARQL

[Run it!] [clear]

Try it for yourself!

Outline

Overview

XPath & XQuery

- XPath
- XQuery
- XQuery Semantics

SPARQL

XSPARQL

- Syntax
- Semantics
- Implementation
- XSPARQL Features
- Query examples

XSPARQL Formal Semantics

- Extend the XQuery semantics
- Adding the normalisation, static type and dynamic evaluation rules for the new expressions:
  - SPARQL `for` clause
  - `construct` clause

Formal semantics types

Newly defined types

`RDFTerm` is the type of SPARQL variables, with the subtypes:

- `uri`
- `bnode`
- `literal`

Formal semantics types

Newly defined types

`RDFTerm` is the type of SPARQL variables, with the subtypes:

- `uri`
- `bnode`
- `literal`

`RDFGraph` will be the type of `construct` expressions

Formal semantics types

Newly defined types

`RDFTerm` is the type of SPARQL variables, with the subtypes:

- `uri`
- `bnode`
- `literal`

`RDFGraph` will be the type of `construct` expressions

`PatternSolution` is a pair (*variableName*, *RDFTerm*) representing SPARQL variable bindings

SPARQL `for` - Static Type Analysis

$$\frac{\begin{array}{c} \text{statEnv} + \text{varType}(\mathit{Variable}_1 \Rightarrow \mathit{RDFTerm}; \\ \cdots; \\ \mathit{Variable}_n \Rightarrow \mathit{RDFTerm} \\ ) \vdash \mathit{ReturnExpr} : \mathit{Type}_2 \end{array}}{\text{statEnv} \vdash \begin{array}{l} \texttt{for}\ \$\mathit{Variable}_1 \cdots \$\mathit{Variable}_n\ \mathit{DatasetClause} \\ \texttt{where}\ \mathit{GroupGraphPattern}\ \mathit{SolutionModifier} \\ \texttt{return}\ \mathit{ReturnExpr} : \mathit{Type}_2 * \end{array}}$$

SPARQL for - Static Type Analysis

$$\frac{\text{statEnv} + \text{varType}(\mathit{Variable}_1 \Rightarrow \mathit{RDFTerm}; \; \cdots ; \; \mathit{Variable}_n \Rightarrow \mathit{RDFTerm}) \vdash \mathit{ReturnExpr} : \mathit{Type}_2}{\text{statEnv} \vdash \begin{array}{l} \texttt{for } \$\mathit{Variable}_1 \cdots \$\mathit{Variable}_n \; \mathit{DatasetClause} \\ \texttt{where } \mathit{GroupGraphPattern} \; \mathit{SolutionModifier} \\ \texttt{return } \mathit{ReturnExpr} : \mathit{Type}_2 * \end{array}}$$

$\$Variable_1 \cdots \$Variable_n$ are of type RDFTerm

SPARQL `for` - Static Type Analysis

$$\frac{\begin{array}{c}\text{statEnv} + \text{varType}(\mathit{Variable}_1 \Rightarrow \mathit{RDFTerm};\\ \cdots;\\ \mathit{Variable}_n \Rightarrow \mathit{RDFTerm}\\ ) \vdash \mathit{ReturnExpr} : \mathit{Type}_2\end{array}}{\text{statEnv} \vdash \begin{array}{l}\texttt{for } \$\mathit{Variable}_1 \cdots \$\mathit{Variable}_n \; \mathit{DatasetClause}\\ \texttt{where } \mathit{GroupGraphPattern} \; \mathit{SolutionModifier}\\ \texttt{return } \mathit{ReturnExpr} : \mathit{Type}_2{*}\end{array}}$$

$*$ comes from the sequence of SPARQL results

Example: Simple SPARQL for

Simple SPARQL for example

```
for $s $p $o
from <foaf.rdf> where { $s $p $o }
return ($s, $p, $o)
```

Example: Simple SPARQL for

For each SPARQL result

Simple SPARQL for example

```
for $s $p $o
from <foaf.rdf> where { $s $p $o }
return ($s, $p, $o)
```

Example: Simple SPARQL for

Simple SPARQL for example

```
for $s $p $o
from <foaf.rdf> where { $s $p $o }
return ($s, $p, $o)
```

For each SPARQL result

Variables are assigned values

Example: Simple SPARQL for

Simple SPARQL for example

```
for $s $p $o
from <foaf.rdf> where { $s $p $o }
return ($s, $p, $o)
```

For each SPARQL result

Variables are assigned values

Return expresion is evaluated

SPARQL `for` - Dynamic Evaluation

$$\frac{\begin{array}{l}
\text{dynEnv} \vdash \mathit{fs}{:}\mathit{sparql}(\mathit{DatasetClause}, \mathit{GroupGraphPattern}, \\
\qquad\qquad\qquad \mathit{SolutionModifier}) \Rightarrow \mathit{PS}_1, \ldots, \mathit{PS}_m \\
\text{dynEnv} + \text{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_1); \\
\qquad\qquad\qquad \ldots; \\
\qquad\qquad\qquad \mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_n) \\
\qquad\qquad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1 \\
\qquad\qquad\qquad \vdots \\
\text{dynEnv} + \text{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_1); \\
\qquad\qquad\qquad \ldots; \\
\qquad\qquad\qquad \mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_n) \\
\qquad\qquad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_m
\end{array}}{\text{dynEnv} \vdash \begin{array}{l}
\texttt{for}\ \$\mathit{Variable}_1 \cdots \$\mathit{Variable}_n \\
\texttt{where}\ \mathit{GroupGraphPattern}\ \mathit{SolutionModifier} \\
\texttt{return}\ \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m
\end{array}}$$

SPARQL `for` - Dynamic Evaluation

The results of the SPARQL query

$$\frac{\begin{array}{r} \text{dynEnv} \vdash \mathit{fs}{:}\mathit{sparql}(\mathit{DatasetClause}, \mathit{GroupGraphPattern}, \\ \mathit{SolutionModifier}) \Rightarrow \mathit{PS}_1, \ldots, \mathit{PS}_m \\ \text{dynEnv} + \text{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_1); \\ \ldots; \\ \mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_n) \\ ) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1 \\ \vdots \\ \text{dynEnv} + \text{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_1); \\ \ldots; \\ \mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_n) \\ ) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_m \end{array}}{\text{dynEnv} \vdash \begin{array}{l} \texttt{for } \$\mathit{Variable}_1 \cdots \$\mathit{Variable}_n \\ \texttt{where } \mathit{GroupGraphPattern}\ \mathit{SolutionModifier} \\ \texttt{return } \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m \end{array}}$$

SPARQL `for` - Dynamic Evaluation

$$\frac{\begin{array}{l} \text{dynEnv} \vdash \mathit{fs}{:}\mathit{sparql}(\mathit{DatasetClause}, \mathit{GroupGraphPattern}, \\ \qquad\qquad \mathit{SolutionModifier}) \Rightarrow \mathit{PS}_1, \ldots, \mathit{PS}_m \\ \text{dynEnv} + \text{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_1); \\ \qquad \ldots; \\ \qquad \mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_n) \\ \quad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1 \\ \qquad \vdots \\ \text{dynEnv} + \text{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_1); \\ \qquad \ldots; \\ \qquad \mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_n) \\ \quad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_m \end{array}}{\text{dynEnv} \vdash \begin{array}{l} \texttt{for } \$\mathit{Variable}_1 \cdots \$\mathit{Variable}_n \\ \texttt{where } \mathit{GroupGraphPattern}\ \mathit{SolutionModifier} \\ \texttt{return } \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m \end{array}}$$

The results of the SPARQL query

fs:sparql evaluates a SPARQL query

SPARQL for - Dynamic Evaluation

For each PS add the values of the variables to dynEnv

$$
\frac{
\begin{array}{c}
\mathrm{dynEnv} \vdash \mathit{fs}{:}\mathit{sparql}(\mathit{DatasetClause}, \mathit{GroupGraphPattern}, \\
\qquad \mathit{SolutionModifier}) \Rightarrow \mathit{PS}_1, \ldots, \mathit{PS}_m \\
\mathrm{dynEnv} + \mathrm{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_1); \\
\ldots; \\
\mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_n) \\
) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1 \\
\vdots \\
\mathrm{dynEnv} + \mathrm{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_1); \\
\ldots; \\
\mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_n) \\
) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_m
\end{array}
}{
\mathrm{dynEnv} \vdash
\begin{array}{l}
\texttt{for } \$\mathit{Variable}_1 \cdots \$\mathit{Variable}_n \\
\texttt{where } \mathit{GroupGraphPattern}\ \mathit{SolutionModifier} \\
\texttt{return } \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m
\end{array}
}
$$

SPARQL `for` - Dynamic Evaluation

fs:*value* selects the value of *Variable$_i$* from the *PS*

For each PS add the values of the variables to dynEnv

$$\frac{\begin{array}{l} \mathrm{dynEnv} \vdash \mathit{fs}{:}\mathit{sparql}(\mathit{DatasetClause}, \mathit{GroupGraphPattern}, \\ \qquad \mathit{SolutionModifier}) \Rightarrow \mathit{PS}_1, \ldots, \mathit{PS}_m \\ \mathrm{dynEnv} + \mathrm{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_1); \\ \qquad \ldots; \\ \qquad \mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_n) \\ \quad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1 \\ \qquad \vdots \\ \mathrm{dynEnv} + \mathrm{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_1); \\ \qquad \ldots; \\ \qquad \mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_n) \\ \quad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_m \end{array}}{\mathrm{dynEnv} \vdash \begin{array}{l} \texttt{for } \$\mathit{Variable}_1 \cdots \$\mathit{Variable}_n \\ \texttt{where } \mathit{GroupGraphPattern}\ \mathit{SolutionModifier} \\ \texttt{return } \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m \end{array}}$$

SPARQL `for` - Dynamic Evaluation

$$
\frac{
\begin{array}{l}
\mathrm{dynEnv} \vdash \mathit{fs}{:}\mathit{sparql}(\mathit{DatasetClause}, \mathit{GroupGraphPattern}, \\
\qquad\qquad\qquad \mathit{SolutionModifier}) \Rightarrow \mathit{PS}_1, \ldots, \mathit{PS}_m \\
\mathrm{dynEnv} + \mathrm{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_1); \\
\qquad\qquad\qquad \ldots; \\
\qquad\qquad\qquad \mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_n) \\
\qquad\qquad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1 \\
\qquad\qquad\qquad \vdots \\
\mathrm{dynEnv} + \mathrm{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_1); \\
\qquad\qquad\qquad \ldots; \\
\qquad\qquad\qquad \mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_n) \\
\qquad\qquad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_m
\end{array}
}{
\mathrm{dynEnv} \vdash
\begin{array}{l}
\texttt{for}\ \$\mathit{Variable}_1 \cdots \$\mathit{Variable}_n \\
\texttt{where}\ \mathit{GroupGraphPattern}\ \mathit{SolutionModifier} \\
\texttt{return}\ \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m
\end{array}
}
$$

For each `PS` add the values of the variables to dynEnv

SPARQL for - Dynamic Evaluation

$$\dfrac{\begin{array}{l} \text{dynEnv} \vdash \mathit{fs}{:}\mathit{sparql}(\mathit{DatasetClause}, \mathit{GroupGraphPattern}, \\ \qquad\qquad \mathit{SolutionModifier}) \Rightarrow \mathit{PS}_1, \ldots, \mathit{PS}_m \\ \text{dynEnv} + \text{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_1); \\ \qquad\qquad \ldots; \\ \qquad\qquad \mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_1, \mathit{Variable}_n) \\ \qquad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1 \\ \qquad\qquad \vdots \\ \text{dynEnv} + \text{varValue}(\mathit{Variable}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_1); \\ \qquad\qquad \ldots; \\ \qquad\qquad \mathit{Variable}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mathit{PS}_m, \mathit{Variable}_n) \\ \qquad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_m \end{array}}{\text{dynEnv} \vdash \begin{array}{l} \texttt{for } \$\mathit{Variable}_1 \cdots \$\mathit{Variable}_n \\ \texttt{where } \mathit{GroupGraphPattern}\ \mathit{SolutionModifier} \\ \texttt{return } \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m \end{array}}$$

The result of the expression is the sequence of computed *Values*

construct expressions

$$\left[\!\left[\ \texttt{construct}\ \mathit{ConstructTemplate}'\ \right]\!\right]_{\mathit{Expr}}$$

$$==$$

$$\texttt{return}\ \mathit{fs}{:}\mathit{evalTemplate}\left([\mathit{ConstructTemplate}']_{\mathit{normaliseTemplate}}\right)$$

construct expressions

$$\left[\!\left[\ \texttt{construct}\ \mathit{ConstructTemplate}\ \right]\!\right]_{\mathit{Expr}}$$

$$==$$

$$\texttt{return}\ \mathit{fs{:}evalTemplate}\left(\left[\mathit{ConstructTemplate'}\right]_{\mathit{normaliseTemplate}}\right)$$

$[\cdot]_{\mathit{normaliseTemplate}}$ expands any Turtle shortcuts in its argument

construct expressions

fs:evalTemplate validates the created triples

$$\left[\!\left[\ \texttt{construct}\ \mathit{ConstructTemplate}'\ \right]\!\right]_{\mathit{Expr}}$$

$$==$$

$$\texttt{return}\ \mathit{fs{:}evalTemplate}\left([\mathit{ConstructTemplate}']_{\mathit{normaliseTemplate}}\right)$$

Outline

Overview

XPath & XQuery

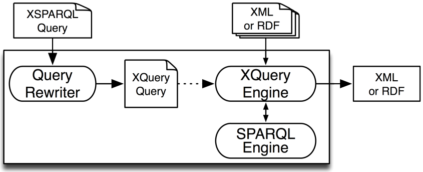
- XPath
- XQuery
- XQuery Semantics

SPARQL

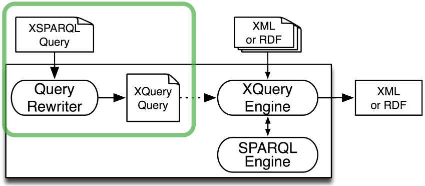
XSPARQL

- Syntax
- Semantics
- Implementation
- XSPARQL Features
- Query examples

Implementation

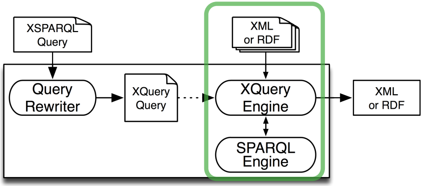


Implementation



- Each XSPARQL query is rewritten into an XQuery

Implementation



- Each XSPARQL query is rewritten into an XQuery
- SPARQLForClauses are translated into SPARQL SELECT queries and executed using a SPARQL engine

Lifting: Rewriting to XQuery

Convert our relations data into RDF

```
declare namespace foaf="http://xmlns.com/foaf/0.1/";
let $persons := doc("relations.xml")//person
let $ids := data($persons/@name)
for $p in $persons
  let $id := fn:index-of($ids, $p/@name)
  construct { _:b{$id} a foaf:Person; foaf:name {data($p/@name)}.
              { for $k in $p/knows
                let $kid := fn:index-of($ids, $k)
                construct { _:b{$id}  foaf:knows _:b{$kid} } } }
```

Lifting: Rewriting to XQuery

Convert our relations data into RDF

```
{ for $k in $p/knows
  let $kid := fn:index-of($ids, $k)
  construct { _:b{$id}  foaf:knows _:b{$kid} } } }
```

Lifting: Rewriting to XQuery

Convert our relations data into RDF (partial)

```
{ for $k in $p/knows
  let $kid := fn:index-of($ids, $k)
  construct { _:b{$id}  foaf:knows _:b{$kid} } } }
```

Rewritten XQuery (and re-rewritten for presentation purposes)

```
for $k at $k_pos in $p/knows
let $kid := fn:index-of( $ids, $k )
return
  let $_rdf8 := _xsparql:_binding_term( "_:b", $id, "", "" )
  let $_rdf9 := _xsparql:_binding_term( "_:b", $kid, "", "" )
  return _xsparql:_serialize((
    if (_xsparql:_validSubject($_rdf8) and _xsparql:_validObject($_rdf9))
    then
      ($_rdf8, " foaf:knows ", _xsparql:_rdf_term( $_rdf9 ), " .&#xA;")
    else "")
 ) ) )
```

Lifting: Rewriting to XQuery

Convert our relations data into RDF (partial)

```
{ for $k in $p/knows
  let $kid := fn:index-of($ids, $k)
  construct { _:b{$id}  foaf:knows _:b{$kid} } } }
```

Rewritten XQuery (and re-rewritten for presentation purposes)

```
for $k at $k_pos in $p/knows
let $kid := fn:index-of( $ids, $k )
return
  let $_rdf8 := _xsparql:_binding_term( "_:b", $id, "", "" )
  let $_rdf9 := _xsparql:_binding_term( "_:b", $kid, "", "" )
  return _xsparql:_serialize((
    if ( _xsparql:_validSubject($_rdf8) and _xsparql:_validObject($_rdf9))
    then
      ($_rdf8, " foaf:knows ", _xsparql:_rdf_term( $_rdf9 ), " .&#xA;")
    else "")
 ) ) )
```

Create variables of type RDFTerm for each constructed term

Lifting: Rewriting to XQuery

Convert our relations data into RDF (partial)

```
{ for $k in $p/knows
  let $kid := fn:index-of($ids, $k)
  construct { _:b{$id}  foaf:knows _:b{$kid} } } }
```

Rewritten XQuery (and re-rewritten for presentation purposes)

```
for $k at $k_pos in $p/knows
let $kid := fn:index-of( $ids, $k )
return
  let $_rdf8 := _xsparql:_binding_term( "_:b", $id, "", "" )
  let $_rdf9 := _xsparql:_binding_term( "_:b", $kid, "", "" )
  return _xsparql:_serialize((
    if (_xsparql:_validSubject($_rdf8) and _xsparql:_validObject($_rdf9))
    then
      ($_rdf8, " foaf:knows ", _xsparql:_rdf_term( $_rdf9 ), " .&#xA;")
    else "")
 ) ) )
```

If all the variables represent valid RDF terms for the given position we output the triple

Handling blank nodes in CONSTRUCT

Construct foaf:Person

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

for $id in ("a","b","c","d")
construct { _:Person a foaf:Person }
```

Handling blank nodes in CONSTRUCT

Construct foaf:Person

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

for $id in ("a","b","c","d")
construct { _:Person a foaf:Person }
```

Blank nodes
should be different
for each solution

Handling blank nodes in CONSTRUCT

Construct foaf:Person

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

for $id in ("a","b","c","d")
construct { _:Person a foaf:Person }
```

Append position variable to the blank node

XQuery rewriting

```
for $id at $id_pos in ("a", "b", "c", "d")
let $_rdf0 := _xsparql:_binding_term(fn:concat("_:Person","_",$id_pos))
return
  _xsparql:_serialize( (
    if (_xsparql:_validSubject( $_rdf0 )) then
      ($_rdf0, " ", "a", " ", "foaf:Person", " .&#xA;")
    else "") )
```

Handling blank nodes in CONSTRUCT

Construct foaf:Person

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

for $id in ("a","b","c","d")
construct { _:Person a foaf:Person }
```

Query result

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:Person_1 a foaf:Person .
_:Person_2 a foaf:Person .
_:Person_3 a foaf:Person .
_:Person_4 a foaf:Person .
```

Lowering: Rewriting to XQuery

Convert FOAF RDF data into XML

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
       { for $FName from <relations.rdf>
         where { $Person foaf:knows $Friend .
                 $Person foaf:name $Name.
                 $Friend foaf:name $FName. }
         return <knows> { $FName }</knows>}
       </person>}
</relations>
```

Lowering: Rewriting to XQuery

Convert FOAF RDF data into XML

```
for $Person $Name from <relations.rdf>
where { $Person foaf:name $Name }
order by $Name
```

Lowering: Rewriting to XQuery

Convert FOAF RDF data into XML (partial)

```
for $Person $Name from <relations.rdf>
where { $Person foaf:name $Name }
order by $Name
```

Rewriting to XQuery

```
let $_aux_results3 := _xsparql:_sparqlQuery(
"PREFIX foaf: <http://xmlns.com/foaf/0.1/>
 SELECT $Person $Name from <relations.rdf>
 where  { $Person foaf:name $Name . } order by $Name " )
for $_aux_result3 at $_aux_result3_pos in  $_aux_results3
  let $Person := _xsparql:_resultNode( $_aux_result3, "Person" )
  let $Name := _xsparql:_resultNode( $_aux_result3, "Name" )
```

Lowering: Rewriting to XQuery

Convert FOAF RDF data into XML (partial)

```
for $Person $Name from <relations.rdf>
where { $Person foaf:name $Name }
order by $Name
```

Rewriting to XQuery

```
let $_aux_results3 := _xsparql:_sparqlQuery(
"PREFIX foaf: <http://xmlns.com/foaf/0.1/>
 SELECT $Person $Name from <relations.rdf>
 where  { $Person foaf:name $Name . } order by $Name " )
for $_aux_result3 at $_aux_result3_pos in  $_aux_results3
  let $Person := _xsparql:_resultNode( $_aux_result3, "Person" )
  let $Name := _xsparql:_resultNode( $_aux_result3, "Name" )
```

Lowering: Rewriting to XQuery

Convert FOAF RDF data into XML (partial)

```
for $Person $Name from <relations.rdf>
where { $Person foaf:name $Name }
order by $Name
```

Rewriting to XQuery

```
let $_aux_results3 := _xsparql:_sparqlQuery(
"PREFIX foaf: <http://xmlns.com/foaf/0.1/>
 SELECT $Person $Name from <relations.rdf>
 where  { $Person foaf:name $Name . } order by $Name " )
for $_aux_result3 at $_aux_result3_pos in  $_aux_results3
  let $Person := _xsparql:_resultNode( $_aux_result3, "Person" )
  let $Name := _xsparql:_resultNode( $_aux_result3, "Name" )
```

Lowering: Rewriting to XQuery

Convert FOAF RDF data into XML (partial)

```
for $Person $Name from <relations.rdf>
where { $Person foaf:name $Name }
order by $Name
```

Rewriting to XQuery

```
let $_aux_results3 := _xsparql:_sparqlQuery(
"PREFIX foaf: <http://xmlns.com/foaf/0.1/>
 SELECT $Person $Name from <relations.rdf>
 where  { $Person foaf:name $Name . } order by $Name " )
for $_aux_result3 at $_aux_result3_pos in  $_aux_results3
  let $Person := _xsparql:_resultNode( $_aux_result3, "Person" )
  let $Name := _xsparql:_resultNode( $_aux_result3, "Name" )
```

Lowering: Rewriting to XQuery

Convert FOAF RDF data into XML (partial)

```
for $Person $Name from <relations.rdf>
where { $Person foaf:name $Name }
order by $Name
```

_sparqlQuery Implementation: HTTP call, Java extension

Rewriting to XQuery

```
let $_aux_results3 := _xsparql:_sparqlQuery(
"PREFIX foaf: <http://xmlns.com/foaf/0.1/>
 SELECT $Person $Name from <relations.rdf>
 where  { $Person foaf:name $Name . } order by $Name " )
for $_aux_result3 at $_aux_result3_pos in  $_aux_results3
  let $Person := _xsparql:_resultNode( $_aux_result3, "Person" )
  let $Name := _xsparql:_resultNode( $_aux_result3, "Name" )
```

Lowering: Rewriting to XQuery

Convert FOAF RDF data into XML

```
for $FName from <relations.rdf>
where { $Person foaf:knows $Friend .
        $Person foaf:name $Name.
        $Friend foaf:name $FName. }
```

Lowering: Rewriting to XQuery

Convert FOAF RDF data into XML (partial)

```
for $FName from <relations.rdf>
where { $Person foaf:knows $Friend .
        $Person foaf:name $Name.
        $Friend foaf:name $FName. }
```

Rewriting to XQuery

```
let $_aux_results5 := _xsparql:_sparqlQuery( fn:concat(
  "PREFIX foaf: <http://xmlns.com/foaf/0.1/>
   SELECT $FName from <relations.rdf>
   where  { ", $Person, " foaf:knows $Friend . ",
   $Person, " foaf:name ", $Name, " .
   $Friend foaf:name $FName . } " ) )
for $_aux_result5 at $_aux_result5_pos in $_aux_results5
let $FName := _xsparql:_resultNode( $_aux_result5, "FName" )
```

Lowering: Rewriting to XQuery

Convert FOAF RDF data into XML (partial)

```
for $FName from <relations.rdf>
where { $Person foaf:knows $Friend .
        $Person foaf:name $Name.
        $Friend foaf:name $FName. }
```

Rewriting to XQuery

```
let $_aux_results5 := _xsparql:_sparqlQuery( fn:concat(
  "PREFIX foaf: <http://xmlns.com/foaf/0.1/>
   SELECT $FName from <relations.rdf>
   where  { ", $Person, " foaf:knows $Friend . ",
   $Person, " foaf:name ", $Name, " .
   $Friend foaf:name $FName . } " ) )
for $_aux_result5 at $_aux_result5_pos in $_aux_results5
let $FName := _xsparql:_resultNode( $_aux_result5, "FName" )
```

$Person and $Name instantiated by the outer loop

Lowering: Rewriting to XQuery

Convert FOAF RDF data into XML (partial)

```
for $FName from <relations.rdf>
where { $Person foaf:knows $Friend .
        $Person foaf:name $Name.
        $Friend foaf:name $FName. }
```

Rewriting to XQuery

```
let $_aux_results5 := _xsparql:_sparqlQuery( fn:concat(
  "PREFIX foaf: <http://xmlns.com/foaf/0.1/>
   SELECT $FName from <relations.rdf>
   where  { ", $Person, " foaf:knows $Friend . ",
   $Person, " foaf:name ", $Name, " .
   $Friend foaf:name $FName . } " ) )
for $_aux_result5 at $_aux_result5_pos in $_aux_results5
let $FName := _xsparql:_resultNode( $_aux_result5, "FName" )
```

Replaced by their value in the query string

Outline

Overview

XPath & XQuery
- XPath
- XQuery
- XQuery Semantics

SPARQL

XSPARQL
- Syntax
- Semantics
- Implementation
- XSPARQL Features
- Query examples

Lowering Example

Convert FOAF RDF data into XML

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
       { for $FName from <relations.rdf>
         where { $Person foaf:knows $Friend .
                 $Person foaf:name $Name.
                 $Friend foaf:name $FName. }
         return <knows> { $FName }</knows>}
       </person>}
</relations>
```

Lowering Example

Convert FOAF RDF data into XML

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
       { for $FName from <relations.rdf>
         where { $Person foaf:knows $Friend .
                 $Person foaf:name $Name.
                 $Friend foaf:name $FName. }
         return <knows> { $FName }</knows>}
       </person>}
</relations>
```

$Person and $Name instantiated by the outer loop

Lowering Example

Convert FOAF RDF data into XML

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
       { for $FName from <relations.rdf>
         where { $Person foaf:knows $Friend .
                 $Person foaf:name $Name.
                 $Friend foaf:name $FName. }
         return <knows> { $FName }</knows>}
       </person>}
</relations>
```

If $Person is a blank node, query joining is done by the $Name variable

Problem with the Lowering example

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:b1 a foaf:Person; foaf:name "Alice"; foaf:knows _:b2.
_:b4 a foaf:Person; foaf:name "Alice"; foaf:knows _:b3.
_:b2 a foaf:Person; foaf:name "Bob"; foaf:knows _:b3.
_:b3 a foaf:Person; foaf:name "Charles".
```

Problem with the Lowering example

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:b1 a foaf:Person; foaf:name "Alice"; foaf:knows _:b2.
_:b4 a foaf:Person; foaf:name "Alice"; foaf:knows _:b3.
_:b2 a foaf:Person; foaf:name "Bob"; foaf:knows _:b3.
_:b3 a foaf:Person; foaf:name "Charles".
```

Problem with the Lowering example

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:b1 a foaf:Person; foaf:name "Alice"; foaf:knows _:b2.
_:b4 a foaf:Person; foaf:name "Alice"; foaf:knows _:b3.
_:b2 a foaf:Person; foaf:name "Bob"; foaf:knows _:b3.
_:b3 a foaf:Person; foaf:name "Charles".
```

```
<relations>
   <person name="Alice">
      <knows>Charles</knows>
      <knows>Bob</knows>
   </person>
   <person name="Alice">
      <knows>Charles</knows>
      <knows>Bob</knows>
   </person>
   <person name="Bob"><knows>Charles</knows></person>
   <person name="Charles"/>
</relations>
```

Problem with the Lowering example

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:b1 a foaf:Person; foaf:name "Alice"; foaf:knows _:b2.
_:b4 a foaf:Person; foaf:name "Alice"; foaf:knows _:b3.
_:b2 a foaf:Person; foaf:name "Bob"; foaf:knows _:b3.
_:b3 a foaf:Person; foaf:name "Charles".
```

```
<relations>
   <person name="Alice">
      <knows>Charles</knows>
      <knows>Bob</knows>
   </person>
   <person name="Alice">
      <knows>Charles</knows>
      <knows>Bob</knows>
   </person>
   <person name="Bob"><knows>Charles</knows></person>
   <person name="Charles"/>
</relations>
```

Scoped Dataset

Scoping Graph

SPARQL query solutions are taken from the *scoping graph*: a graph that is equivalent to the active graph but does not share any blank nodes with it or any graph pattern within the query.

Scoped Dataset

Scoping Graph

SPARQL query solutions are taken from the *scoping graph*: a graph that is equivalent to the active graph but does not share any blank nodes with it or any graph pattern within the query.

Scoped Dataset

The XSPARQL *scoped dataset* allows to make SPARQL queries over the previous *scoping graph*, keeping the same blank node assignments

Scoped Dataset Example

Convert FOAF RDF data into XML (Scoped Dataset)

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
           { for $FName
             where { $Person foaf:knows $Friend .
                     $Friend foaf:name $FName. }
             return <knows> { $FName }</knows>}
           </person>}
</relations>
```

Scoped Dataset Example

Convert FOAF RDF data into XML (Scoped Dataset)

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
         { for $FName
           where { $Person foaf:knows $Friend .
                   $Friend foaf:name $FName. }
           return <knows> { $FName }</knows>}
         </person>}
</relations>
```

Keep the same *active* dataset that was last used

Scoped Dataset Example

Convert FOAF RDF data into XML (Scoped Dataset)

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
         { for $FName
           where { $Person foaf:knows $Friend .
                   $Friend foaf:name $FName. }
           return <knows> { $FName }</knows>}
         </person>}
</relations>
```

Scoped Dataset Example

Convert FOAF RDF data into XML (Scoped Dataset)

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
         { for $FName
           where { $Person foaf:knows $Friend .
                   $Friend foaf:name $FName. }
           return <knows> { $FName }</knows>}
         </person>}
</relations>
```

Can no longer be implemented as rewriting to a SPARQL query

Scoped Dataset Example

Query output

```
<relations>
   <person name="Alice">
      <knows>Charles</knows>
   </person>
   <person name="Alice">
      <knows>Bob</knows>
   </person>
   <person name="Bob">
      <knows>Charles</knows>
   </person>
   <person name="Charles"/>
</relations>
```

Scoped Dataset Example

Query output

```
<relations>
   <person name="Alice">
      <knows>Charles</knows>
   </person>
   <person name="Alice">
      <knows>Bob</knows>
   </person>
   <person name="Bob">
      <knows>Charles</knows>
   </person>
   <person name="Charles"/>
</relations>
```

Constructed Dataset

- Assign the result of a construct query to a variable
- The variable can then be used as the dataset of a SPARQL query

Lifting and Lowering query :)

```
let $ds := for $person in doc("relations.xml")//person,
      $nameA in $person/@name,
      $nameB in $person/knows
  construct { [ foaf:name {data($nameA)}; a foaf:Person ] foaf:knows
              [ foaf:name {data($nameB)}; a foaf:Person ]. }
return <relations>{ for $Person $Name from $ds
  where { $Person foaf:name $Name } order by $Name
  return <person name="{$Name}">{ for $FName
           where { $Person foaf:knows $Friend .
                   $Friend foaf:name $FName. }
           return <knows> { $FName }</knows>}
</person>}</relations>
```

Constructed Dataset

- Assign the result of a construct query to a variable
- The variable can then be used as the dataset of a SPARQL query

Lifting and Lowering query :)

```
let $ds := for $person in doc("relations.xml")//person,
      $nameA in $person/@name,
      $nameB in $person/knows
  construct { [ foaf:name {data($nameA)}; a foaf:Person ] foaf:knows
              [ foaf:name {data($nameB)}; a foaf:Person ]. }
return <relations>{ for $Person $Name from $ds
  where { $Person foaf:name $Name } order by $Name
  return <person name="{$Name}">{ for $FName
           where { $Person foaf:knows $Friend .
                   $Friend foaf:name $FName. }
           return <knows> { $FName }</knows>}
</person>}</relations>
```

Constructed Dataset

- Assign the result of a construct query to a variable
- The variable can then be used as the dataset of a SPARQL query

Lifting and Lowering query :)

```
let $ds := for $person in doc("relations.xml")//person,
      $nameA in $person/@name,
      $nameB in $person/knows
  construct { [ foaf:name {data($nameA)}; a foaf:Person ] foaf:knows
              [ foaf:name {data($nameB)}; a foaf:Person ]. }
return <relations>{ for $Person $Name from $ds
  where { $Person foaf:name $Name } order by $Name
  return <person name="{$Name}">{ for $FName
           where { $Person foaf:knows $Friend .
                   $Friend foaf:name $FName. }
           return <knows> { $FName }</knows>}
</person>}</relations>
```

Constructed Dataset

- Assign the result of a construct query to a variable
- The variable can then be used as the dataset of a SPARQL query

Lifting and Lowering query :)

```
let $ds := for $person in doc("relations.xml")//person,
      $nameA in $person/@name,
      $nameB in $person/knows
  construct { [ foaf:name {data($nameA)}; a foaf:Person ] foaf:knows
              [ foaf:name {data($nameB)}; a foaf:Person ]. }
return <relations>{ for $Person $Name from $ds
  where { $Person foaf:name $Name } order by $Name
  return <person name="{$Name}">{ for $FName
           where { $Person foaf:knows $Friend .
                   $Friend foaf:name $FName. }
           return <knows> { $FName }</knows>}
</person>}</relations>
```

$ds contains the RDF Dataset

Constructed Dataset

- Assign the result of a construct query to a variable
- The variable can then be used as the dataset of a SPARQL query

Lifting and Lowering query :)

```
let $ds := for $person in doc("relations.xml")//person,
      $nameA in $person/@name,
      $nameB in $person/knows
  construct { [ foaf:name {data($nameA)}; a foaf:Person ] foaf:knows
              [ foaf:name {data($nameB)}; a foaf:Person ]. }
return <relations>{ for $Person $Name from $ds
  where { $Person foaf:name $Name } order by $Name
  return <person name="{$Name}">{ for $FName
           where { $Person foaf:knows $Friend .
                   $Friend foaf:name $FName. }
           return <knows>{ $FName }</knows>}
</person>}</relations>
```

Later used in a from clause

Outline

Overview

XPath & XQuery

- XPath
- XQuery
- XQuery Semantics

SPARQL

XSPARQL

- Syntax
- Semantics
- Implementation
- XSPARQL Features
- Query examples

More expressive SPARQL

Convert between different RDF vocabularies (SPARQL)

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

construct { $X foaf:name $FN.}
from <vCard.ttl>
where { $X vc:FN $FN .}
order by $FN
limit 1 offset 1
```

More expressive SPARQL

Convert between different RDF vocabularies

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

construct { _:b foaf:name {fn:concat($N," ", $F)}.}
from <vCard.rdf>
where { $P vc:Given $N. $P vc:Family $F. }
```

More expressive SPARQL

Convert between different RDF vocabularies

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

construct { _:b foaf:name {fn:concat($N," ", $F)}.}
from <vCard.rdf>
where { $P vc:Given $N. $P vc:Family $F. }
```

Construction of new values not available in SPARQL 1.0

More expressive SPARQL

Convert between different RDF vocabularies

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

construct { _:b foaf:name {fn:concat($N," ", $F)}.}
from <vCard.rdf>
where { $P vc:Given $N. $P vc:Family $F. }
```

XSPARQL provides you with all the XQuery functions

Query Example - KML Lowering

Create a KML file from RDF Geolocation data

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>

<kml xmlns="http://www.opengis.net/kml/2.2">{
 for $person $name $long $lat
 from <http://nunolopes.org/foaf.rdf>
 where { $person a foaf:Person; foaf:name $name;
         foaf:based_near [ a geo:Point; geo:long $long;
                           geo:lat $lat ] }
return <Placemark>
         <name>{fn:concat("Location of ", $name)}</name>
         <Point>
           <coordinates>{fn:concat($long, ",", $lat, ",0")}
           </coordinates>
         </Point>
       </Placemark>
}</kml>
```

Query Example - KML Lowering

Create a KML file from RDF Geolocation data

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>

<kml xmlns="http://www.opengis.net/kml/2.2">{
 for $person $name $long $lat
 from <http://nunolopes.org/foaf.rdf>
 where { $person a foaf:Person; foaf:name $name;
         foaf:based_near [ a geo:Point; geo:long $long;
                           geo:lat $lat ] }
return <Placemark>
         <name>{fn:concat("Location of ", $name)}</name>
         <Point>
           <coordinates>{fn:concat($long, ",", $lat, ",0")}
           </coordinates>
         </Point>
       </Placemark>
}</kml>
```

SPARQL: *Persons and their geographic locations*

Query Example - KML Lowering

Create a KML file from RDF Geolocation data

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>

<kml xmlns="http://www.opengis.net/kml/2.2">{
 for $person $name $long $lat
 from <http://nunolopes.org/foaf.rdf>
 where { $person a foaf:Person; foaf:name $name;
         foaf:based_near [ a geo:Point; geo:long $long;
                            geo:lat $lat ] }
return <Placemark>
         <name>{fn:concat("Location of ", $name)}</name>
         <Point>
           <coordinates>{fn:concat($long, ",", $lat, ",0")}
           </coordinates>
         </Point>
       </Placemark>
}</kml>
```

XML representing the specific KML file structure

Query Example - KML Lowering

Create a KML file from RDF Geolocation data

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>

<kml xmlns="http://www.opengis.net/kml/2.2">{
 for $person $name $long $lat
 from <http://nunolopes.org/foaf.rdf>
 where { $person a foaf:Person; foaf:name $name;
         foaf:based_near [ a geo:Point; geo:long
                                 geo:lat $lat ] }
return <Placemark>
         <name>{fn:concat("Location of ", $name)
         <Point>
           <coordinates>{fn:concat($long, ",", $
           </coordinates>
         </Point>
       </Placemark>
}</kml>
```

Query Example - KML Lifting

Create RDF Geolocation data from a KML file

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
prefix kml: <http://earth.google.com/kml/2.0>

let $loc := "Departamento de Ingeneria Matematica, U. de Chile"
for $place in doc(fn:concat("http://maps.google.com/?q=",
                  fn:encode-for-uri($loc),"&amp;num=1&amp;output=kml"))
let $geo := fn:tokenize($place//kml:coordinates, ",")
construct { [] foaf:based_near [ a geo:Point; geo:long {$geo[1]};
                                geo:lat {$geo[2]} ] }
```

Query Example - KML Lifting

Create RDF Geolocation data from a KML file

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
prefix kml: <http://earth.google.com/kml/2.0>

let $loc := "Departamento de Ingeneria Matematica, U. de Chile"
for $place in doc(fn:concat("http://maps.google.com/?q=",
                  fn:encode-for-uri($loc),"&amp;num=1&amp;output=kml"))
let $geo := fn:tokenize($place//kml:coordinates, ",")
construct { [] foaf:based_near [ a geo:Point; geo:long {$geo[1]};
                                 geo:lat {$geo[2]} ] }
```

Google doesn't know Departamento de Ciencias de la Computacion??

Query Example - KML Lifting

Create RDF Geolocation data from a KML file

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
prefix kml: <http://earth.google.com/kml/2.0>

let $loc := "Departamento de Ingeneria Matematica, U. de Chile"
for $place in doc(fn:concat("http://maps.google.com/?q=",
                  fn:encode-for-uri($loc),"&amp;num=1&amp;output=kml"))
let $geo := fn:tokenize($place//kml:coordinates, ",")
construct { [] foaf:based_near [ a geo:Point; geo:long {$geo[1]};
                                 geo:lat {$geo[2]} ] }
```

Ask Google Maps for the KML description

Query Example - KML Lifting

Create RDF Geolocation data from a KML file

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
prefix kml: <http://earth.google.com/kml/2.0>

let $loc := "Departamento de Ingeneria Matematica, U. de Chile"
for $place in doc(fn:concat("http://maps.google.com/?q=",
                  fn:encode-for-uri($loc),"&amp;num=1&amp;output=kml"))
let $geo := fn:tokenize($place//kml:coordinates, ",")
construct { [] foaf:based_near [ a geo:Point; geo:long {$geo[1]};
                                  geo:lat {$geo[2]} ] }
```

Construct the updated RDF graph

Query Example - KML Lifting

Create RDF Geolocation data from a KML file

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
prefix kml: <http://earth.google.com/kml/2.0>

let $loc := "Departamento de Ingeneria Matematica, U. de Chile"
for $place in doc(fn:concat("http://maps.google.com/?q=",
                  fn:encode-for-uri($loc),"&amp;num=1&amp;output=kml"))
let $geo := fn:tokenize($place//kml:coordinates, ",")
construct { [] foaf:based_near [ a geo:Point; geo:long {$geo[1]};
                                geo:lat {$geo[2]} ] }
```

Output RDF data

```
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
[ foaf:based_near [ a geo:Point ; geo:long "-70.664448" ;
                                  geo:lat "-33.457286" ] ] .
```

Queries over Linked Data

Give me all pairs of co-authors and their joint publications.

```
let $ds :=
  for * from <http://dblp.l3s.de/d2r/resource/authors/Axel_Polleres>
  where { $pub dc:creator [] }
  construct { { for * from $pub where { $p dc:creator $o . }
               construct { $p dc:creator $o } } }
let $allauthors := distinct-values(
                      for $o from $ds where {$p dc:creator $o}
                      order by $o return $o)
for $auth at $auth_pos in $allauthors
 for $coauth in $allauthors[position() > $auth_pos]
    let $commonPubs := count(
     { for $pub from $ds
       where { $pub dc:creator $auth, $coauth }
       return $pub } )
where ($commonPubs > 0)
construct { [ :author $auth; :author $coauth; :commonPubs $commonPubs ] }
```

Queries over Linked Data

Give me all pairs of co-authors and their joint publications.

```
let $ds :=
  for * from <http://dblp.l3s.de/d2r/resource/authors/Axel_Polleres>
  where { $pub dc:creator [] }
  construct { { for * from $pub where { $p dc:creator $o . }
                construct { $p dc:creator $o } } }
let $allauthors := distinct-values(
                      for $o from $ds where {$p dc:creator $o}
                      order by $o return $o)
for $auth at $auth_pos in $allauthors
 for $coauth in $allauthors[position() > $auth_pos]
    let $commonPubs := count(
     { for $pub from $ds
        where { $pub dc:creator $auth, $coauth }
        return $pub } )
where ($commonPubs > 0)
construct { [ :author $auth; :author $coauth; :commonPubs $commonPubs ] }
```

Find all the co-authors of Axel Polleres

Queries over Linked Data

Give me all pairs of co-authors and their joint publications.

```
let $ds :=
  for * from <http://dblp.l3s.de/d2r/resource/authors/Axel_Polleres>
  where { $pub dc:creator [] }
  construct { { for * from $pub where { $p dc:creator $o . }
               construct { $p dc:creator $o } } }
let $allauthors := distinct-values(
                      for $o from $ds where {$p dc:creator $o}
                      order by $o return $o)
for $auth at $auth_pos in $allauthors
 for $coauth in $allauthors[position() > $auth_pos]
    let $commonPubs := count(
     { for $pub from $ds
       where { $pub dc:creator $auth, $coauth }
       return $pub } )
where ($commonPubs > 0)
construct { [ :author $auth; :author $coauth; :commonPubs $commonPubs ] }
```

And the distinct names

Queries over Linked Data

Give me all pairs of co-authors and their joint publications.

```
let $ds :=
  for * from <http://dblp.l3s.de/d2r/resource/authors/Axel_Polleres>
  where { $pub dc:creator [] }
  construct { { for * from $pub where { $p dc:creator $o . }
               construct { $p dc:creator $o } } }
let $allauthors := distinct-values(
                      for $o from $ds where {$p dc:creator $o}
                      order by $o return $o)
for $auth at $auth_pos in $allauthors
 for $coauth in $allauthors[position() > $auth_pos]
    let $commonPubs := count(
     { for $pub from $ds
       where { $pub dc:creator $auth, $coauth }
       return $pub } )
where ($commonPubs > 0)
construct { [ :author $auth; :author $coauth; :commonPubs $commonPubs ] }
```

For each distinct pair of authors

Queries over Linked Data

Give me all pairs of co-authors and their joint publications.

```
let $ds :=
  for * from <http://dblp.l3s.de/d2r/resource/authors/Axel_Polleres>
  where { $pub dc:creator [] }
  construct { { for * from $pub where { $p dc:creator $o . }
                construct { $p dc:creator $o } } }
let $allauthors := distinct-values(
                      for $o from $ds where {$p dc:creator $o}
                      order by $o return $o)
for $auth at $auth_pos in $allauthors
 for $coauth in $allauthors[position() > $auth_pos]
    let $commonPubs := count(
     { for $pub from $ds
       where { $pub dc:creator $auth, $coauth }
       return $pub } )
where ($commonPubs > 0)
construct { [ :author $auth; :author $coauth; :commonPubs $commonPubs ] }
```

Count the number of their shared publications

Queries over Linked Data

Give me all pairs of co-authors and their joint publications.

```
let $ds :=
  for * from <http://dblp.l3s.de/d2r/resource/authors/Axel_Polleres>
  where { $pub dc:creator [] }
  construct { { for * from $pub where { $p dc:creator $o . }
               construct { $p dc:creator $o } } }
let $allauthors := distinct-values(
                      for $o from $ds where {$p dc:creator $o}
                      order by $o return $o)
for $auth at $auth_pos in $allauthors
 for $coauth in $allauthors[position() > $auth_pos]
    let $commonPubs := count(
     { for $pub from $ds
       where { $pub dc:creator $auth, $coauth }
       return $pub } )
where ($commonPubs > 0)
construct { [ :author $auth; :author $coauth; :commonPubs $commonPubs ] }
```

Create the RDF output

The end...

http://xsparql.deri.org/

Downloads: http://sourceforge.net/projects/xsparql/

new version coming very soon!

Mailing List: xsparql-discussion@lists.sourceforge.net

This tutorial: http://nunolopes.org/presentations/2010.12.17-XSPARQLTutorial.pdf

Outline

Optimisations

Optimisations & Benchmarks

Data

- Benchmark suite for XML
  - http://www.xml-benchmark.org/
  - Provides data generator and 20 benchmark queries
  - Data simulates an auction website, containing people, items and auctions

Optimisations & Benchmarks

Data

- Benchmark suite for XML
  - http://www.xml-benchmark.org/
  - Provides data generator and 20 benchmark queries
  - Data simulates an auction website, containing people, items and auctions
- Converted XML data to RDF

Optimisations & Benchmarks

Data

- Benchmark suite for XML
  - http://www.xml-benchmark.org/
  - Provides data generator and 20 benchmark queries
  - Data simulates an auction website, containing people, items and auctions
- Converted XML data to RDF
- Queries written using XSPARQL

Optimisations & Benchmarks

Data

- Benchmark suite for XML
  - http://www.xml-benchmark.org/
  - Provides data generator and 20 benchmark queries
  - Data simulates an auction website, containing people, items and auctions
- Converted XML data to RDF
- Queries written using XSPARQL

Query example

List the names of persons and the number of items they bought

Benchmark Query example

XQuery

```
let $auction := doc("input.xml") return
for $p in $auction/site/people/person
let $a := for $t in $auction/site/closed_auctions/closed_auction
          where $t/buyer/@person = $p/@id
          return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

Benchmark Query example

XQuery

```
let $auction := doc("input.xml") return
for $p in $auction/site/people/person
let $a := for $t in $auction/site/closed_auctions/closed_auction
          where $t/buyer/@person = $p/@id
          return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

Translation to XSPARQL

```
for $id $name from <input.rdf>
where { $person a foaf:Person ; :id $id ; foaf:name $name . }
return <item person="{$name}">{
  let $x := for * from $rdf
            where { $ca a :ClosedAuction ; :buyer [ :id $id ] . }
            return $ca
  return count($x)
}</item>
```

Rewriting to XQuery

Unoptimised Version

```
let $_aux_results0 := _xsparql:_sparqlQuery( fn:concat(
 "SELECT $id $name from <input.rdf>
  WHERE { $person  a  foaf:Person ; :id $id ; foaf:name $name .}"))
for $_aux_result0 at $_aux_result0_pos in $_aux_results0
 let $id := _xsparql:_resultNode( $_aux_result0, "id" )
 let $name := _xsparql:_resultNode( $_aux_result0, "name" )
 return <item person="{$name}">{
 let $x := let $_aux_results2 := _xsparql:_sparqlQuery( fn:concat(
           "SELECT $ca FROM <input.rdf>
            WHERE { $ca a :ClosedAuction ; :buyer [:id ", $id, "] .}"))
   for $_aux_result2 at $_aux_result2_pos in $_aux_results2
   let $ca := _xsparql:_resultNode( $_aux_result2, "ca" )
   return  $ca
return count( $x )
}</item>
```

Rewriting to XQuery

Unoptimised Version

```
let $_aux_results0 := _xsparql:_sparqlQuery( fn:concat(
 "SELECT $id $name from <input.rdf>
  WHERE { $person  a  foaf:Person ; :id $id ; foaf:name $name .}"))
for $_aux_result0 at $_aux_result0_pos in $_aux_results0
 let $id := _xsparql:_resultNode( $_aux_result0, "id" )
 let $name := _xsparql:_resultNode( $_aux_result0, "name" )
 return <item person="{$name}">{
 let $x := let $_aux_results2 := _xsparql:_sparqlQuery( fn:concat(
           "SELECT $ca FROM <input.rdf>
            WHERE { $ca a :ClosedAuction ; :buyer [:id ", $id, "] .}"))
   for $_aux_result2 at $_aux_result2_pos in $_aux_results2
   let $ca := _xsparql:_resultNode( $_aux_result2, "ca" )
   return  $ca
return count( $x )
}</item>
```

Outer SPARQL query

Rewriting to XQuery

Unoptimised Version

```
let $_aux_results0 := _xsparql:_sparqlQuery( fn:concat(
 "SELECT $id $name from <input.rdf>
  WHERE { $person  a  foaf:Person ; :id $id ; foaf:name $name .}"))
for $_aux_result0 at $_aux_result0_pos in $_aux_results0
 let $id := _xsparql:_resultNode( $_aux_result0, "id" )
 let $name := _xsparql:_resultNode( $_aux_result0, "name" )
 return <item person="{$name}">{
 let $x := let $_aux_results2 := _xsparql:_sparqlQuery( fn:concat(
           "SELECT $ca FROM <input.rdf>
            WHERE { $ca a :ClosedAuction ; :buyer [:id ", $id, "] .}"))
   for $_aux_result2 at $_aux_result2_pos in $_aux_results2
   let $ca := _xsparql:_resultNode( $_aux_result2, "ca" )
   return  $ca
return count( $x )
}</item>
```

Inner SPARQL query

Rewriting to XQuery

Optimised Version (nested loop join)

```
let $_aux_results4 := _xsparql:_sparqlQuery( fn:concat(
 "SELECT $ca $id from <input.rdf>
  WHERE { $ca a :ClosedAuction; :buyer [:id $id ] .}") )
let $_aux_results0 := _xsparql:_sparqlQuery( fn:concat(
 "SELECT $id $name from <input.rdf>
  WHERE { $person a foaf:Person; :id $id; foaf:name $name .}"))
for $_aux_result0 at $_aux_result0_pos in $_aux_results0
 let $id := _xsparql:_resultNode( $_aux_result0, "id" )
 let $name := _xsparql:_resultNode( $_aux_result0, "name" )
return <item person="{$name}">{
 let $x :=
  for $_aux_result4 at $_aux_result4_pos in $_aux_results4
  where $id = _xsparql:_resultNode( $_aux_result4, "id" )
  return _xsparql:_resultNode( $_aux_result4, "ca" )
 return count( $x )
}</item>
```

Rewriting to XQuery

Optimised Version (nested loop join)

```
let $_aux_results4 := _xsparql:_sparqlQuery( fn:concat(
 "SELECT $ca $id from <input.rdf>
  WHERE { $ca a :ClosedAuction; :buyer [:id $id ] .}") )
let $_aux_results0 := _xsparql:_sparqlQuery( fn:concat(
 "SELECT $id $name from <input.rdf>
  WHERE { $person a foaf:Person; :id $id; foaf:name $name .}"))
for $_aux_result0 at $_aux_result0_pos in $_aux_results0
 let $id := _xsparql:_resultNode( $_aux_result0, "id" )
 let $name := _xsparql:_resultNode( $_aux_result0, "name" )
return <item person="{$name}">{
 let $x :=
  for $_aux_result4 at $_aux_result4_pos in $_aux_results4
  where $id = _xsparql:_resultNode( $_aux_result4, "id" )
  return _xsparql:_resultNode( $_aux_result4, "ca" )
 return count( $x )
}</item>
```

Inner SPARQL query

Rewriting to XQuery

Optimised Version (nested loop join)

```
let $_aux_results4 := _xsparql:_sparqlQuery( fn:concat(
 "SELECT $ca $id from <input.rdf>
  WHERE { $ca a :ClosedAuction; :buyer [:id $id ] .}") )
let $_aux_results0 := _xsparql:_sparqlQuery( fn:concat(
 "SELECT $id $name from <input.rdf>
  WHERE { $person a foaf:Person; :id $id; foaf:name $name .}"))
for $_aux_result0 at $_aux_result0_pos in $_aux_results0
 let $id := _xsparql:_resultNode( $_aux_result0, "id" )
 let $name := _xsparql:_resultNode( $_aux_result0, "name" )
return <item person="{$name}">{
 let $x :=
  for $_aux_result4 at $_aux_result4_pos in $_aux_results4
  where $id = _xsparql:_resultNode( $_aux_result4, "id" )
  return _xsparql:_resultNode( $_aux_result4, "ca" )
 return count( $x )
}</item>
```

Outer SPARQL query

Rewriting to XQuery

Optimised Version (nested loop join)

```
let $_aux_results4 := _xsparql:_sparqlQuery( fn:concat(
 "SELECT $ca $id from <input.rdf>
  WHERE { $ca a :ClosedAuction; :buyer [:id $id ] .}") )
let $_aux_results0 := _xsparql:_sparqlQuery( fn:concat(
 "SELECT $id $name from <input.rdf>
  WHERE { $person a foaf:Person; :id $id; foaf:name $name .}"))
for $_aux_result0 at $_aux_result0_pos in $_aux_results0
 let $id := _xsparql:_resultNode( $_aux_result0, "id" )
 let $name := _xsparql:_resultNode( $_aux_result0, "name" )
return <item person="{$name}">{
 let $x :=
  for $_aux_result4 at $_aux_result4_pos in $_aux_results4
  where $id = _xsparql:_resultNode( $_aux_result4, "id" )
  return _xsparql:_resultNode( $_aux_result4, "ca" )
 return count( $x )
}</item>
```

Preliminary results look promising...

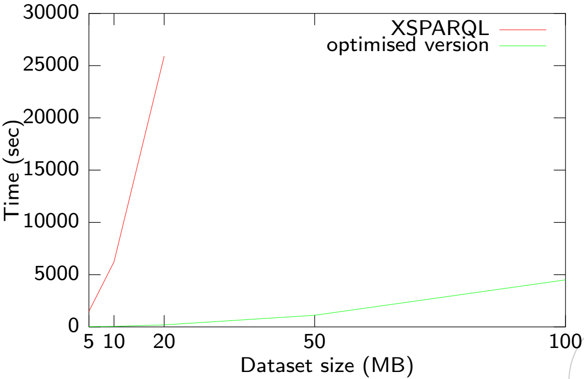


Figure: optimisation query 08