Net-Based Applications

Chapter 9: XQuery

Holger Schwarz
Universität Stuttgart

Winter Term 2016/2017

Overview

- Motivation and introduction
- Node construction
- FLWOR expressions
  - Syntax and Clauses
  - Joins in XQuery
  - Grouping and Aggregation
- User-defined functions
- Updates

Processing XML Data

- Processing XML data is needed for:
  - Querying XML data
  - Translation of information from one XML schema to another
- Standard XML querying/translation languages:
  - **XPath**
    - Simple language consisting of path expressions.
  - **XSLT**
    - Simple language designed for translation from XML to XML and XML to other text-based languages.
  - **XQuery**
    - An XML query language with a rich set of features.
    - XQuery builds on experience with existing query languages: XPath, Quilt, XQL, XML-QL, Lorel, YATL, SQL, OQL, ...

Processing XML Data: XQuery

- XQuery is a general purpose query language for XML data.
- Standardized by the World Wide Web Consortium (W3C):
  XQuery 1.0: W3C Recommendation 23 January 2007
- XQuery is derived from the **Quilt** query language, which itself borrows from:
  (“Quilt” refers both to the origin of the language and to its use in “knitting ” together heterogeneous data sources)
  - **XPath**: a concise language for navigating in trees
  - **XML-QL**: a powerful language for generating new structures
  - **SQL**: a database language based on a series of keyword-clauses: SELECT - FROM – WHERE
  - **OQL**: a functional language in which many kinds of expressions can be nested with full generality

Why using a query language?

- XQuery is a domain-specific query language (domain: XML)
- Why learn XQuery when you can use Java and some XML API (like DOM, SAX, StAX, ...)?
  2 reasons:
- Ease of use
  - work with domain-specific concepts directly (instead of API)
  - express the same thing with fewer lines of code
- Perfomance
  - optimized for tasks common to domain
  - no overhead because of API
  - less constraints: declare, what the result should be, not how it is obtained → potential for automatic optimization

XQuery: Language Requirements

- XQuery should be applicable to XML documents
  - without type information
  - with some type information (DTD)
  - with detailed type information (XML schema)
- An XML query language must be able to:
  - Query deeply nested and heterogeneous structures
  - Query metadata as well as user data
  - Search for objects by absolute and relative order
  - Preserve order of objects in input documents
  - Impose new ordering at multiple levels of output
  - Handle missing data and sparse data
  - Preserve or transform the structure of a document
  - Exploit references to unknown or heterogeneous types
  - Easily define recursive functions
  - Provide a very flexible data definition facility

Overview

- Motivation and introduction
- Node construction
- FLWOR expressions
  - Syntax and Clauses
  - Joins in XQuery
  - Grouping and Aggregation
- User-defined functions
- Updates

Document Node

- Represents an XML document
- No single root element required (as in XML 1.0)
- Constructor: document {Expr}
- Example

```
document {<course>Advanced Information Management</course>}
```

Element Nodes

- Represents an XML element
- Direct element construction

```
{<course>Advanced Information Management</course>}
```

- Element construction using expressions

```
<x y="6*7 = {6*7}">
It is { true() or false() }!
</x>
```

➔

```
<x y="6*7 = 42">It is true !</x>
```

```
<add>
  {{ 1 + 1  = { 1+1 }}}
</add>
```

➔

```
<add>{ 1 + 1 = 2 }</add>
```

escaped curly braces

Attribute Nodes

- Provide attribute value directly

```
{<course id="728">Advanced Information Management</course>}
```

- Attribute values and expressions
  - (part of an) attribute value may be provided as an expression
  - evaluated expression is turned into a sequence of atomic values (atomization)
  - elements of the sequence are casted to xs:string
  - all strings are concatenated to provide the attribute value

```
<employee salary="$
{ <calculate>
  12*4000
  </calculate>
} per year"/>
```

➔

```
<employee salary="$ 48000 per year" />
```

Element and Attribute Names

- May be taken from an expression as well
- Constructors:
  - element *name* { *expr* }
  - element { *expr* } { *expr* }
  - attribute *name* { *expr* }
  - attribute { *expr* } { *expr* }
- Examples

```
element {concat("a", "b")} {"c", 3} }
```

➔

```
<ab>c 3</ab>
```

```
<x>{ attribute {"ab"}{"c", "d", 3} }</x>
```

➔

```
<x ab="c d 3" />
```

Other Node Types

- Text

```
text {"Content of the text node."}
```

```
<![CDATA[Content of the text node.]]>
```

- Comment

```
comment {"My comment!"}
```

```
<!-- My comment! -->
```

- Namespace

```
namespace foo {urn:bar}
```

→

```
xmlns:foo="urn:bar"
```

- Processing instructions

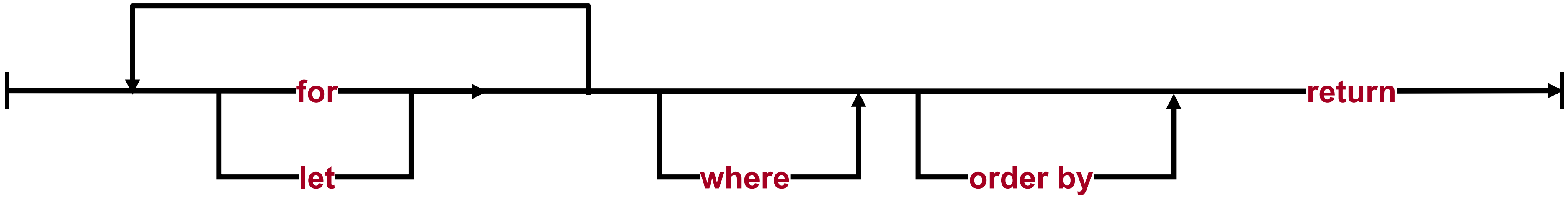
```
<?target content ?>
```

```
processing-instruction {"target"} {"content"}
```

Overview

- Motivation and introduction
- Node construction
- FLWOR expressions
  - Syntax and Clauses
  - Joins in XQuery
  - Grouping and Aggregation
- User-defined functions
- Updates

FLWOR Syntax



- for clause:
  iterates over a set of nodes (possibly specified by an XPath expression), binding a variable to the individual nodes in the set
- let clause:
  binds a variable to the result of an expression
- where clause:
  applies a predicate to filter the variables bound by FOR and LET
- order by clause:
  allows ordering on multiple levels of nesting
- return clause:
  constructs the output

XQuery Syntax

- Associations to SQL query expressions
  for ⇔ SQL from
  where ⇔ SQL where
  order by ⇔ SQL order by
  return ⇔ SQL select
  let allows temporary variables, and has no equivalent in SQL
- XQuery is a functional language
- Each query is an expression
- Expressions can be nested with full generality
  - XPath expressions
  - Element constructors
  - FLWOR expressions
- Path expressions can be used in various places
  - in the For clause to bind variables
  - in the Let clause to bind variables to results of path expressions

Syntax

FLWORExpr	::= (ForClause | LetClause)+ WhereClause? OrderByClause? **return** ExprSingle
ForClause	::= **for** $VarName TypeDeclaration? PositionalVar? **in** ExprSingle (",**"** $VarName TypeDeclaration? PositionalVar? **in** ExprSingle)*
LetClause	::= **let** $VarName TypeDeclaration? **:=** ExprSingle ("**,**" $VarName TypeDeclaration? **:=** ExprSingle)*
TypeDeclaration	::= **as** SequenceType
PositionalVar	::= **at $** VarName
WhereClause	::= **where** Expr
OrderByClause	::= (**order by** | **stable order by**) OrderSpecList
OrderSpecList	::= OrderSpec ("**,**" OrderSpec)*
OrderSpec	::= ExprSingle OrderModifier
OrderModifier	::= (**ascending** | **descending**)? ((**empty greatest**) | (**empty least**))? (**collation** StringLiteral)?

Let Clause

- Allows to bind the result of an expression, e.g., an XPath expression, to a variable
- The variable contains the sequence of atomic values and/or nodes provided by the expression
- The remaining query is evaluated once with this variable binding
- Examples

```
let $v := (<university><student /><professor /></university>)
return $v
```

➔

```
<university>
   <student />
   <professor />
</university>
```

```
let $w := fn:doc("university.xml")
…
```

For Clause

- Allows to bind the result of an expression, e.g., an XPath expression, to a variable
- The variable contains one item of the sequence provided by the expression
- The remaining query is evaluated for each item of the sequence
- Example

```
for $i in  (1, 2), $j in (3, 4, 5), $k in (6, 7)
return ($i ,$j, $k)
```

➔

```
(1, 3, 6, 1, 3, 7, 1, 4, 6, 1, 4, 7, 1, 5, 6, 1, 5, 7,
 2, 3, 6, 2, 3, 7, 2, 4, 6, 2, 4, 7, 2, 5, 6, 2, 5, 7)
```

cartesian product of
input sequences

Comparing Let Clause and For Clause

- Let Clause

```
let $i := ("x", "y", "z")
return fn:count($i)
```

➔

```
3
```

```
let $i := (<x />,<y />, <z />)
return <xyz>{$i}</xyz>
```

➔

```
<xyz><x /><y /><z /></xyz>
```

- For Clause

```
for $i in ("x", "y", "z")
return fn:count($i)
```

➔

```
(1, 1, 1)
```

```
for $i in (<x />,<y />, <z />)
return <xyz>{$i}</xyz>
```

➔

```
<xyz><x /></xyz>
<xyz><y /></xyz>
<xyz><z /></xyz>
```

Position Variables

- Allows to refer to the position of items in a sequence
- Example:

```
<student_list>
{
for $x at $i in (<student1 />, <student2 />, <student3 />)
return (<id>{$i}</id>,$x)
}
</student_list>
```

➔

```
<student_list>
   <id>1</id><student1 />
   <id>2</id><student2 />
   <id>3</id><student3 />
</student_list>
```

Let/For Clause and Types

- Check the type of items of the given sequence
- Runtime error if not
- Example:

```
for $x as xs:integer in ("John", "Tom", "Max")
return $x * 10
```

Where Clause

- Provides predicates that are evaluated on current variable binding
- Variable binding is kept if where clause evaluates to true
- Example

```
<selected_students>
{
for $x at $i in (<student1 yearOfBirth="1989" />,
                 <student2 yearOfBirth="1990" />,
                 <student3 yearOfBirth="1989" />)
where $x/@yearOfBirth = 1989
return (<id>{$i}</id>,$x)
}
</selected_students>
```

➔

```
<selected_students>
<id>1</id><student1 yearOfBirth="1989" />
<id>3</id><student3 yearOfBirth="1989" />
</selected_students>
```

Order By Clause

- Defines order in which variable bindings are processed by return clause
- Use ascending or descending to describe sort order
- Use empty greatest or empty least to describe how to treat missing values
- Use collation to define sort order for strings
- How to order variable bindings that are equal according to the given sort order?
  - stable order by: use document order
  - order by: order undefined
- Use fn:unordered to ignore document order
(opens up optimization opportunities)

Order By Clause

- Example: ascending/descending

```
for $x at $i in
   (<student1 age="20" />,
    <student2 age="19" />,
    <student3 age="20" />)
order by $x/@age ascending
return $x
```

↓

```
<student2 age="19" />
<student3 age="20" />
<student1 age="20" />
```

1

```
for $x at $i in
   (<student1 age="20" />,
    <student2 age="19" />,
    <student3 age="20" />)
order by $x/@age descending
return $x
```

↓

```
<student3 age="20" />
<student1 age="20" />
<student2 age="19" />
```

1

1 implementation-specific order

Order By Clause

- Example: stable order by

```
for $x at $i in
   (<student1 age="20" />,
    <student2 age="19" />,
    <student3 age="20" />)
stable order by $x/@age ascending
return $x
```

↓

```
<student2 age="19" />
<student1 age="20" />
<student3 age="20" />
```

2

```
for $x at $i in
   (<student1 age="20" />,
    <student2 age="19" />,
    <student3 age="20" />)
stable order by $x/@age descending
return $x
```

↓

```
<student1 age="20" />
<student3 age="20" />
<student2 age="19" />
```

2

2 document order

Order By Clause

- Example: fn:unordered

```
for $x at $i in
   fn:unordered((<student1 age="20" />,
                 <student2 age="19" />,
                 <student3 age="20" />,
                 <student4 age="21" />))
return $x
```

```
<student1 age="20" />
<student4 age="21" />
<student3 age="20" />
<student2 age="19" />
```

implementation-specific order

Return Clause

- Provides model for query output
- Use element constructors, attribute constructors, variable references and nested FLWOR expressions
- References to variables need an evaluation context marked by { }
- Example

```
for $x in (<student1 />,<student2 />)
return <data>$x</data>
```

➔

```
<data>$x</data>
<data>$x</data>
```

```
for $x in (<student1 />,<student2 />)
return <data>{$x}</data>
```

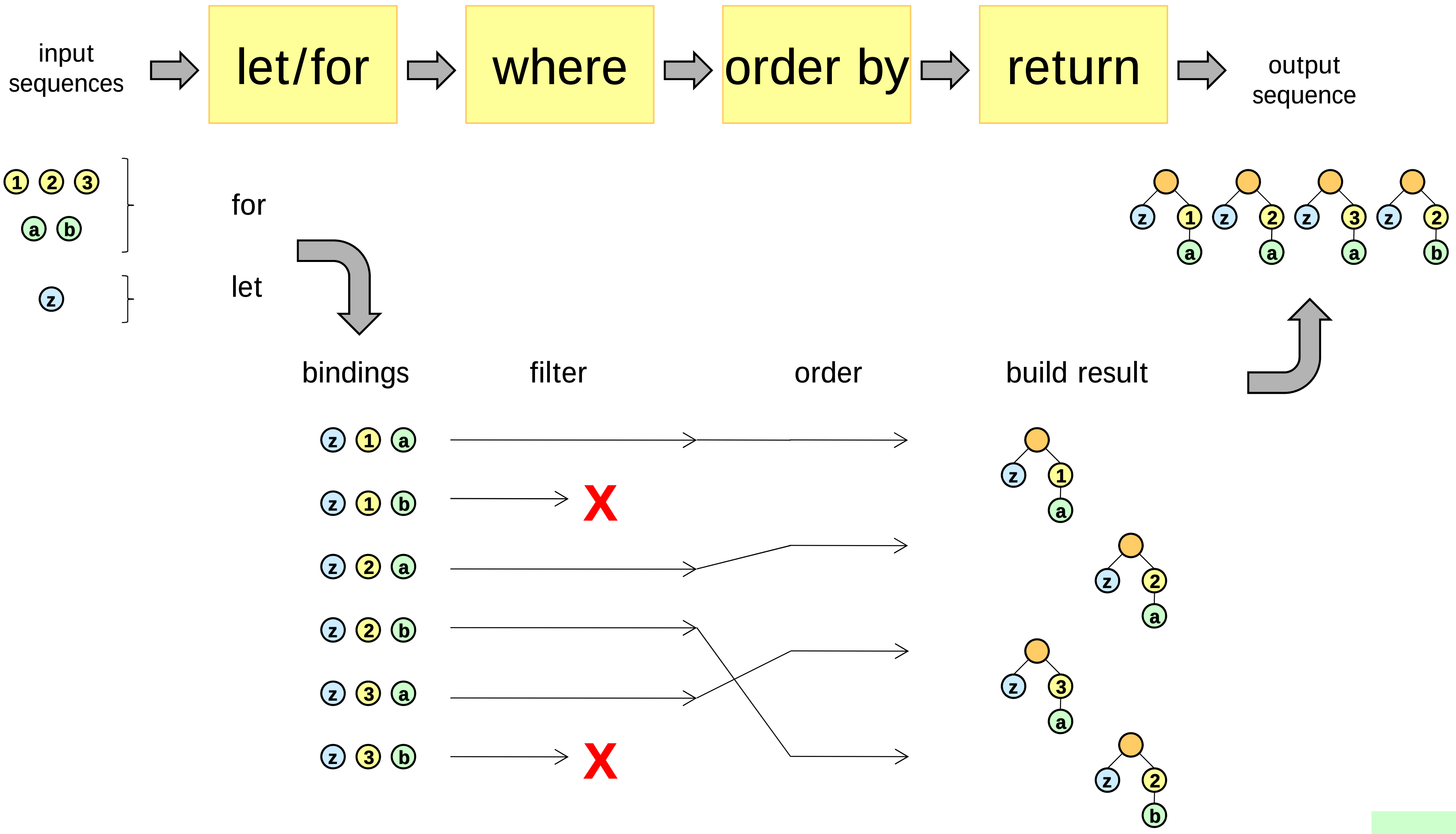
➔

```
<data><student1 /></data>
<data><student2 /></data>
```

```
for $x in fn:doc("students.xml")
order by $x/student/dateOfBirth
return element anonymousStudent
   { $x/student/* except $x/student/name }
```

result covers all information
on students except their name

Evaluating FLWOR Expressions



Overview

- Motivation and introduction
- Node construction
- FLWOR expressions
  - Syntax and Clauses
  - Joins in XQuery
  - Grouping and Aggregation
- User-defined functions
- Updates

Joining Sequences of Values

- For clauses define sequences to be combined
- cross-product or join possible

```
for $x in (1, 2, 3)
for $y in (3, 4, 5)
return ($i, $j)
```

↓

```
(1, 3, 1, 4, 1, 5, 2, 3, 2,
4, 2, 5, 3, 3, 3, 4, 3, 5)
```

```
for $x in (1, 2, 3)
for $y in (3, 4, 5)
where $i = $j
return ($i, $j)
```

↓

```
(3, 3)
```

Sample Documents

projects.xml

```
<?xml version='1.0' ?>
<Projects>
	<Project id="X1" owner="E2">
		<Name>Enter the Tuple Space</Name>
		<Category>Video Games</Category>
	</Project>
	<Project id="X2" owner="E1">
		<Name>Cryptic Codes</Name>
		<Category>Puzzles</Category>
	</Project>
	<Project id="X3" owner="E5">
		<Name>XQuery Bandit</Name>
		<Category>Video Games</Category>
	</Project>
	<Project id="X4" owner="E3">
		<Name>Micropoly</Name>
		<Category>Board Games</Category>
	</Project>
</Projects>
```

team.xml

```
<?xml version='1.0' ?>
<Team name="Project 42">
	<Employee id="E6" years=4.3">
		<Name>Chaz Hoover</Name>
		<Title>Architect</Title>
		<Expertise>Puzzles</Expertise>
		<Expertise>Games</Expertise>
		<Employee id="E2" years="6.1">
			<Name>Carl Yates</Name>
			<Title>Dev Lead</Title>
			<Expertise>Video Games</Expertise>
			<Employee id="E4" years=1.2">
				<Name>Panda Serai</Name>
				<Title>Developer</Titel>
				<Expertise>Hardware</Expertise>
				<Expertise>Entertainment</Expertise>
			</Employee>
			<Employee id="E5" years="0.6">
				<Name> Jason Abedora</Name>
				<Title>Developer</Title>
				<Expertise>Puzzles</Expertise>
			</Employee>
		</Employee>
		<Employee id="E1" years="8.2">
			<Name>Kandy Konrad</NameY
			<Title>QA Lead</Titel>
			<Expertise>Movies</Expertise>
			<Expertise>Sports</Expertise>
			<Employee id="E0" years="8.5"
				<Name>Wanda Wilson</Name>
				<Title>QA Engineer</Title>
				<Expertise>Home Theater</Expertise>
				<Expertise>Board Games</Expertise>
				<Expertise>Puzzles</Expertise>
			</Employee>
		</Employee>
		<Employee id="E3" years="2.8">
			<Name>Jim Barry</Name>
			<Title>QA Engineer</Title>
			<Expertise>Video Games</Expertise>
		</Employee>
	</Employee>
</Team>
```

Joining Several Documents (1:1, n:1)

- Query: Find all projects and the names of their owners

```
for $proj in fn:doc("projects.xml")/Projects/Project
for $emp in fn:doc("team.xml")//Employee
where $proj/@owner = $emp/@id
return $proj/Name, $emp/Name
```

↓

```
<Name>Enter the Tuple Space</Name>
<Name>Carl Yates</Name>
<Name>Cryptic Code</Name>
<Name>Kandy Konrad</Name>
<Name>XQuery Bandit</Name>
<Name>Jason Abedora</Name>
<Name>Micropoly</Name>
<Name>Jim Barry</Name>
```

one-to-one relationship
between project
and owner

Joining Several Documents

- Join predicate as path expression

```
for $proj in fn:doc("projects.xml")/Projects/Project
for $emp in fn:doc("team.xml")//Employee[@id = $proj/@owner]
return $proj/Name, $emp/Name
```

↓

```
<Name>Enter the Tuple Space</Name>
<Name>Carl Yates</Name>
<Name>Cryptic Code</Name>
<Name>Kandy Konrad</Name>
<Name>XQuery Bandit</Name>
<Name>Jason Abedora</Name>
<Name>Micropoly</Name>
<Name>Jim Barry</Name>
```

Joining Several Documents

- Group result by sub elements

```
for $proj in fn:doc("projects.xml")/Projects/Project
for $emp in fn:doc("team.xml")//Employee[@id = $proj/@owner]
return <Assignment>{$proj/Name, $emp/Name}</Assignment>
```

↓

```
<Assignment>
   <Name>Enter the Tuple Space</Name>
   <Name>Carl Yates</Name>
</Assignment>
<Assignment>
   <Name>Cryptic Code</Name>
   <Name>Kandy Konrad</Name>
</Assignment>
<Assignment>
   <Name>XQuery Bandit</Name>
   <Name>Jason Abedora</Name>
</Assignment>
<Assignment>
   <Name>Micropoly</Name>
   <Name>Jim Barry</Name>
</Assignment>
```

Joining Several Documents

- Group result by elements and attributes

```
for $proj in fn:doc("projects.xml")/Projects/Project
for $emp in fn:doc("team.xml")//Employee[@id = $proj/@owner]
return <Assignment proj="{$proj/Name}" emp="{$emp/Name}" />
```

↓

```
<Assignment proj="Enter the Tuple Space" emp="Carl Yates" />
<Assignment proj="Cryptic Code" emp="Kandy Konrad" />
<Assignment proj="XQuery Bandit" emp="Jason Abedora" />
<Assignment proj="Micropoly" emp="Jim Barry" />
```

Joining Several Documents (n:m)

- Query: Find for each project all employees that have the appropriate expertise.

```
for $proj in fn:doc("projects.xml")/Projects/Project
for $emp in fn:doc("team.xml")//Employee
where $proj/Category = $emp/Expertise
return <Assignment proj="{$proj/Name}" emp="{$emp/Name}" />
```

↓

```
<Assignment proj="Enter the Tuple Space" emp="Carl Yates" />
<Assignment proj="Enter the Tuple Space" emp="Jim Barry" />
<Assignment proj="Cryptic Code" emp="Chaz Hoover" />
<Assignment proj="Cryptic Code" emp="Jason Abedora" />
<Assignment proj="Cryptic Code" emp="Wanda Wilson" />
<Assignment proj="XQuery Bandit" emp="Carl Yates" />
<Assignment proj="XQuery Bandit" emp="Jim Barry" />
<Assignment proj="Micropoly" emp="Wanda Wilson" />
```

many-to-many relationship bettween project and employee

projects with no appropriate employee are missing

Outer Join

- Query: Find all projects and for each of them all employees (possibly none) that have the appropriate expertise.

```
for $proj in fn:doc("projects.xml")/Projects/Project
let $emp := fn:doc("team.xml")//Employee[Expertise = $proj/Category]
return <Assignment proj="{$proj/Name}">{$emp/Name}</Assignment>
```

```
<Assignment proj="Enter the Tuple Space">
   <Name>Carl Yates</Name>
   <Name>Jim Barry</Name>
</Assignment>
<Assignment proj="Cryptic Code">
   <Name>Chaz Hoover</Name>
   <Name>Jason Abedora</Name>
   <Name>Wanda Wilson</Name>
<Assignment proj="XQuery Bandit">
   <Name>Carl Yates</Name>
   <Name>Jim Barry</Name>
</Assignment>
<Assignment proj="Micropoly">
   <Name>Wanda Wilson</Name>
</Assignment>
```

Self-Join

- Join a sequence with itself
- Query: Find all employees having the same job title as employee "E0" (Wanda Wilson).

```
let $emp := fn:doc("team.xml")//Employee
for $i in $emp, $j in emp
where $i/Title = $j/Title and $i/@id = "E0"
return $j/Name
```

↓

```
<Name>Wanda Wilson</Name>
<Name>Jim Barry</Name>
```

Joins Based on ID Attributes

- Joins on a single document may use ID attributes and references to them
- Functions: fn:id
  - fn:id delivers all element nodes having one ID from a list of IDs
  - fn:idref delivers all nodes referring to certain IDs
  - The first argument specifies a series of string values (IDs) to be looked up.
- Assume combined sample document projectsAndTeam.xml
  - owner references ID attribute of employees

projectsAndTeam.xml

```
<?xml version='1.0' ?>
<Projects>
        <Project id="X1" owner="E2">
                <Name>Enter the Tuple Space</Name>
                <Category>Video Games</Category>
        </Project>
        <Project id="X2" owner="E1">
                <Name>Cryptic Codes</Name>
                <Category>Puzzles</Category>
        </Project>
        <Project id="X3" owner="E5">
                <Name>XQuery Bandit</Name>
                <Category>Video Games</Category>
        </Project>
        <Project id="X4" owner="E3">
                <Name>Micropoly</Name>
                <Category>Board Games</Category>
        </Project>
</Projects>
<Team name="Project 42">
        <Employee id="E6" years=4.3">
                <Name>Chaz Hoover</Name>
                <Title>Architect</Title>
                <Expertise>Puzzles</Expertise>
                <Expertise>Games</Expertise>
                ...
        </Employee>
</Team>
```

Joins Based on ID Attributes

- Query: Find all projects and the names of their owners

```
for $proj in fn:doc("projectsAndTeam.xml")/Projects/Project
let $emp := fn:id($proj/@owner)
return <Assignment>{$proj/Name, $emp/Name}</Assignment>
```

find the owner
for a project

find the projects
for which the employee
acts as owner

```
for $emp in fn:doc("projectsAndTeam.xml")//Employee
let $proj := (   for $x in fn:idref($emp/@ID)/..
                 where $x instance of element(Project)
                 return $x)
return <Assignment>{$proj/Name, $emp/Name}</Assignment>
```

Overview

- Motivation and introduction
- Node construction
- FLWOR expressions
  - Syntax and Clauses
  - Joins in XQuery
  - Grouping and Aggregation
- User-defined functions
- Updates

Aggregate Functions

signature	description
fn:count($seq as item()*) as xs:integer	returns the number of items in the sequence
fn:avg($seq as xs:anyAtomicType*) as xs:anyAtomicType?	returns the average of the values in the sequence
fn:min / fn:max($seq as xs:anyAtomicType*) as xs:anyAtomicType?	returns the item having the minimum / maximum value from the sequence
fn:sum($seq as xs:anyAtomicType*) as xs:anyAtomicType	returns the sum of all values in the sequence

Grouping by Structure

```
<?xml version='1.0' ?>
<University>
    <Students>
        <Student><Name>Naumann</Name>     <Age>32</Age>         </Student>
        <Student><Name>Shore</Name>       <Age>27</Age>         </Student>
        <Student><Name>Meier</Name>       <Age>25</Age>         </Student>
    </Students>
    <Professors>
        <Professor><Name>Guldenstern</Name><Age>41</Age>        </Professor>
        <Professor><Name>Murawitz</Name>  <Age>65</Age>         </Professor>
    </Professors>
</University>
```

```
<University> {
   for $u in fn:doc("…")//University/*
   let $x := $u/*/Age
   return
      element {fn:node-name($u)} {<Age>{fn:avg($x)}</Age>}
} </University>
```

group by subnodes of university

calculates average
on sequence of age nodes

Grouping by Element Value

```
<?xml version='1.0' ?>
<University>
      <Person> <Position>Student</Position><Name>Naumann</Name><Age>32</Age>                </Person>
      <Person> <Position>Student</Position><Name>Shore</Name><Age>27</Age>                  </Person>
      <Person> <Position>Student</Position><Name>Meier</Name><Age>25</Age>                  </Person>
      <Person> <Position>Professor</Position><Name>Guldenstern</Name><Age>41</Age>          </Person>
      <Person> <Position>Professor</Position><Name>Murawitz</Name><Age>65</Age>             </Person>
</University>
```

group by value
of Position element

```
<University> {
   for $p in fn:distinct-values(fn:doc("…")//Position/text())
   let $x := fn:doc("…")//Age[../Position/text() = $p]
   return
      element {$p} {<Age>{fn:avg($x)}</Age>}
} </University>
```

Grouping by Attribute Values

```
<?xml version='1.0' ?>
<University>
    <Person Position="Student">   <Name>Naumann</Name><Age>32</Age>       </Person>
    <Person Position="Student">   <Name>Shore</Name><Age>27</Age>         </Person>
    <Person Position="Student">   <Name>Meier</Name><Age>25</Age>         </Person>
    <Person Position="Professor"> <Name>Guldenstern</Name><Age>41</Age>   </Person>
    <Person Position="Professor"> <Name>Murawitz</Name><Age>65</Age>      </Person>
</University>
```

group by value
of Position attribute

```
<University> {
   for $p in fn:distinct-values(fn:doc("…")//Person/@Position)
   let $x := fn:doc("…")//Age[../Person/@Position = $p]
   return
      element {$p} {<Age>{fn:avg($x)}</Age>}
} </University>
```

Grouping by Element Names

```
<?xml version='1.0' ?>
<University>
    <Person> <Student /> <Name>Naumann</Name><Age>32</Age>     </Person>
    <Person> <Student /> <Name>Shore</Name><Age>27</Age>       </Person>
    <Person> <Student /> <Name>Meier</Name><Age>25</Age>       </Person>
    <Person> <Professor /> <Name>Guldenstern</Name><Age>41</Age></Person>
    <Person> <Professor /> <Name>Murawitz</Name><Age>65</Age>  </Person>
</University>
```

group by element name

```
<University> {
   for $p in fn:distinct-values(
      for $i in fn:doc("…")//Person/*[fn:node-name(.)='Student' or
                                 fn:node-name(.)='Professor']
      return fn:node-name($i))
   let $x := fn:doc("…")//Age[../fn:node-name(.) = $p]
   return
      element {$p} {<Age>{fn:avg($x)}</Age>}
} </University>
```

Grouping by Several Dimensions

```
<?xml version='1.0' ?>
<University>
      <Students>
            <Student><Name>Naumann</Name>   <Age>32</Age> <Sex>F</Sex>    </Student>
            <Student><Name>Shore</Name>     <Age>27</Age> <Sex>M</Sex>    </Student>
            <Student><Name>Meier</Name>     <Age>25</Age> <Sex>M</Sex>    </Student>
      </Students>
      <Professors>
            <Professor><Name>Guldenstern</Name><Age>41</Age> <Sex>F</Sex> </Professor>
            <Professor><Name>Murawitz</Name> <Age>65</Age> <Sex>F</Sex>   </Professor>
      </Professors>
</University>
```

```
<University> {
   for $p in fn:distinct-values(
      for $i in fn:doc("…")//University/*/*
      return fn:name($i))
   for $s in fn:distinct-values(fn:doc("…")//Sex)
   let $x := fn:doc("…")//[fn:name(.)=$p and Sex=$s]
   return
   (<Position>{$p}</Position><Sex>{$s}</Sex><Age>{fn:avg($x)}</Age>)
} </University>
```

Grouping by Several Dimensions

- The following query removes empty groups from the result:

```
<University> {
   for $p in fn:distinct-values(
      for $i in fn:doc("…")//University/*/*
      return fn:name($i))
   for $s in fn:distinct-values(fn:doc("…")//Sex)
   let $x := fn:doc("…")//[fn:name(.)=$p and Sex=$s]
   where fn:exists($x)
   return
   (<Position>{$p}</Position><Sex>{$s}</Sex><Age>{fn:avg($x)}</Age>)
} </University>
```

Overview

- Motivation and introduction
- Node construction
- FLWOR expressions
  - Syntax and Clauses
  - Joins in XQuery
  - Grouping and Aggregation
- User-defined functions
- Updates

User-Defined Functions (UDF)

- Declared after the query prolog but before the main part of the query
- Function body is either
  - an XQuery expression
  - externally defined
- No overloading
- Use namespace prefix local for functions in current module
- Examples:

```
declare function local:empty-sequence() as empty() {
   ()
}
```

```
declare function abs($i as xs:integer) as
xs:integer {
   if ($i<0) then -$i else $i
}
```

Syntax

```
FunctionDecl      ::= declare function QName "(" ParamList? ")"
                        ( as SequenceType )?
                        ( EnclosedExpr | external )
ParamList         ::= Param ("," Param)*
Param             ::= $ VarName TypeDeclaration?
TypeDeclaration   ::= as SequenceType
EnclosedExpr      ::= "{" Expr "}"
Expr              ::= ExprSingle ("," ExprSingle)*
ExprSingle        ::= FLWORExpr | QuantifiedExpr | TypeswitchExpr | IfExpr | OrExpr
```

Prologue

- XQuery expressions can contain a prologue before the query body.
- The prologue can contain
  - Declarations of UDFs
  - Namespace declarations
  - Order declarations
  - Schema imports
  - ...
- Declarations are terminated by ;

```
declare namespace p = "http://dyomedea.com/ns/people";
fn:doc("author.xml")/p:author/p:name
```

Conditional Expressions

- Conditional expressions: **if … then … else …**
  - condition may be any expression
  - IfExpr may be used wherever an expression is expected
  - else part is mandatory
- Usage:
  - Expression evaluation depending on value

```
for $m in fn:doc("…")
return
    if ($m/Prize*0.1 < 5.0)
    then 5.0
    else if ($m/Prize > 100.0)
        then 10.0
        else $m/Prize*0.1
```

- Check existence

```
if ($m/Prize)
    then …
    else 10.0
```

```
if (fn:exists($m/Prize))
    then …
    else 10.0
```

Quantified Expressions

- Existential quantification: **some**
  - evaluates to true or false
  - empty sequence: evaluates to false

```
some $x in ("Mitschang", "Schwarz") satisfies fn:string-length($x) < 8
```

- Universal quantification: **every**
  - evaluates to true or false
  - empty sequence: evaluates to true
  - Short-circuit evaluation:
    - evaluation stops as soon as the expression evaluates to false for one of the sequence elements
    - evaluation order depends on implementation

```
every $x in ("Mitschang", "Schwarz") satisfies fn:string-length($x) > 6
```

```
every $x in ("Mitschang", "Schwarz", 0.815) satisfies fn:string-length($x) = 9
```

false or runtime error

Typeswitch Expressions

- Use **instance of** to check type of atomic values and sequences

```
<Text>some text</Text> instance of element(*, xs:string)
```

- Use **typeswitch** to concatenate type checks

```
typeswitch ($p)
   case element(*, Professor_T) return 50
   case element(*, Assistant_T) return 40
   case element(*, Student_T) return 30
   default return 10
```

```
typeswitch ($p)
   case $x as element(*, Professor_T) return <Professor>$x</Professor>
   case $x as element(*, Assistant_T) return <Assistant>$x</Assistant>
   case $y as element(*, Student_T) return <Student>$y</Student>
   default return <Others />
```

variable referring to the
result of the switch expression

Syntax

```
QuantifiedExpr      ::=  (some "$" VarName | every "$" VarName ) TypeDeclaration? in ExprSingle
                         ("," "$" VarName TypeDeclaration? in ExprSingle)*
                         satisfies ExprSingle
TypeSwitchExpr      ::=  typeswitch "(" Expr ")"
                              CaseClause+
                              default ("$" VarName)? return ExprSingle
CaseClause          ::=  case ("$" VarName as)? SequenceType return ExprSingle
IfExpr              ::=  if "(" Expr ")" then
                              ExprSingle
                         else
                              ExprSingle
OrExpr              ::=  AndExpr ( or AndExpr)*
AndExpr             ::= ... ( and ... )*
```

... stands for several types of expressions containing **instance of**, **treat as**, **castable as**, **cast as**, arithmetic expressions and path expressions

Overview

- Motivation and introduction
- Node construction
- FLWOR expressions
  - Syntax and Clauses
  - Joins in XQuery
  - Grouping and Aggregation
- User-defined functions
- Updates

XML DML Languages

- So far, XQuery does not specify insert, update or delete
- Several languages have been proposed – some without any connection to XQuery
  - SiXDML, XUpdate, ...
- XQuery Update Facility 1.0: W3C Recommendation 17 March 2011
  - Latest version: http://www.w3.org/TR/xquery-update-10

```
insert node <year>2005</year> after fn:doc("bib.xml")/books/book[1]/publisher
```

```
delete nodes /email/message [fn:currentDate() - date >
                                           xs:dayTimeDuration("P365D")]
```

```
replace node fn:doc("bib.xml")/books/book[1]/publisher with
fn:doc("bib.xml")/books/book[2]/publisher
```

```
rename node fn:doc("bib.xml")/books/book[1]/author[1] as "principal-author"
```

Literature & Information

[Bru04] Michael Brundage: XQuery - The XML Query Language. Addison Wesley, 2004.

[LS04] Wolfgang Lehner, Harald Schöning: XQuery – Grundlagen und fortgeschrittene Methoden. dpunkt.verlag, 2004.