

Lecture 3

XML Databases: XQuery

Martin Svoboda

martin.svoboda@fit.cvut.cz

17. 10. 2017

Lecture Outline

Native XML databases

- General introduction

XQuery and XPath

- Data model
- Query expressions
 - Path expressions
 - FLWOR expressions
 - Constructors, conditions, quantifiers, comparisons, ...

XQuery and XPath

XML Query Language

XML Path Language

Introduction

XPath = *XML Path Language*

- **Navigation in an XML tree, selection of nodes by a variety of criteria**
- Versions: 1.0 (1999), 2.0, 3.0, **3.1** (March 2017)
- W3C recommendation
 - <https://www.w3.org/TR/xpath-31/>

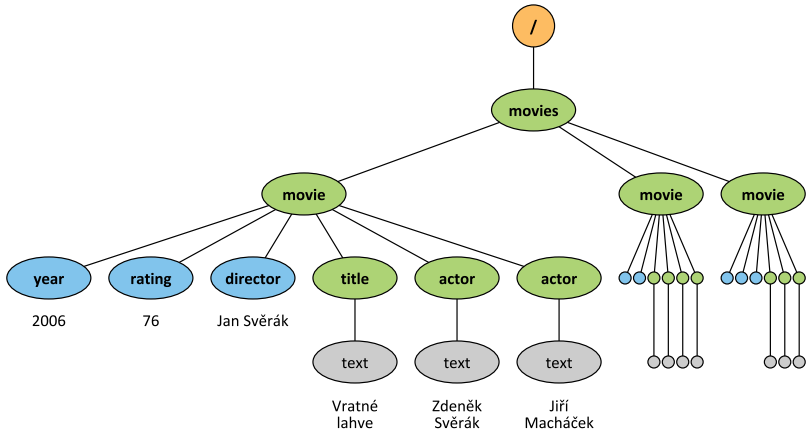
XQuery = *XML Query Language*

- **Complex functional query language**
- Contains XPath
- Versions: 1.0 (2007), 3.0 (2014), **3.1** (March 2017)
- W3C recommendation
 - <https://www.w3.org/TR/xquery-31/>

Sample Data

```
<?xml version="1.1" encoding="UTF-8"?>
<movies>
  <movie year="2006" rating="76" director="Jan Svěrák">
    <title>Vratné lahve</title>
    <actor>Zdeněk Svěrák</actor>
    <actor>Jiří Macháček</actor>
  </movie>
  <movie year="2000" rating="84">
    <title>Samotáři</title>
    <actor>Jitka Schneiderová</actor>
    <actor>Ivan Trojan</actor>
    <actor>Jiří Macháček</actor>
  </movie>
  <movie year="2007" rating="53" director="Jan Hřebejk">
    <title>Medvídek</title>
    <actor>Jiří Macháček</actor>
    <actor>Ivan Trojan</actor>
  </movie>
</movies>
```

Sample Data



Data Model

XDM = *XQuery and XPath Data Model*

- **XML tree** consisting of **nodes** of different kinds
 - Document, element, attribute, text, ...
- **Document order** / reverse document order
 - The order in which nodes appear in the XML file
 - I.e. nodes are numbered using a **pre-order depth-first traversal**

Query result

- Each query expression is evaluated to a **sequence**

Data Model

Sequence = ordered collection of **nodes** and/or **atomic values**

- Automatically **flattened**
 - E.g.: $(1, (), (2, 3), (4)) \Leftrightarrow (1, 2, 3, 4)$
- Standalone items are treated as singleton sequences
 - E.g.: $1 \Leftrightarrow (1)$
- Can be **mixed**
 - But usually just nodes, or just atomic values
- **Duplicate items** are allowed

Expressions

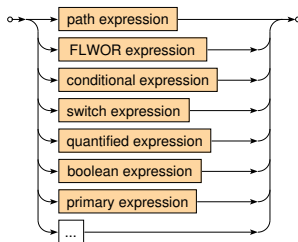
XQuery expressions

- **Path** expressions (traditional XPath)
 - Selection of nodes of an XML tree
- **FLWOR** expressions
 - `for ... let ... where ... order by ... return ...`
- **Conditional** expressions
 - `if ... then ... else ...`
- **Switch** expressions
 - `switch ... case ... default ...`
- **Quantified** expressions
 - `some|every ... satisfies ...`

Expressions

XQuery expressions

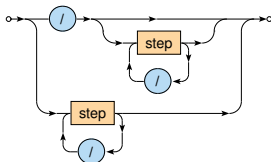
- **Boolean** expressions
 - `and`, `or`, `not` logical connectives
- **Primary** expressions
 - Literals, variable references, function calls, **constructors**, ...
- ...



Path Expressions

Path expression

- Describes navigation within an XML tree
- Consists of individual navigational steps



- **Absolute** paths = path expressions starting with /
 - Navigation starts at the document node
- **Relative** paths
 - Navigation starts at an explicitly specified node / nodes

Path Expressions

Examples

Absolute paths

```
/
```

```
/movies
```

```
/movies/movie
```

```
/movies/movie/title/text()
```

```
/movies/movie/@year
```

Relative paths

```
actor/text()
```

```
@director
```

Path Expressions

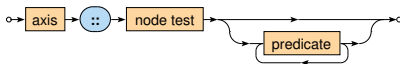
Evaluation of path expressions

- Let P be a **path expression**
- Let C be an initial **context set**
 - If P is **absolute**, then C contains just the document node
 - Otherwise (i.e. P is **relative**) C is given by the user or context
- If P does not contain any step
 - Then C is the **final result**
- Otherwise (i.e. when P contains **at least one step**)
 - Let S be the **first step**, P' the **remaining steps** (if any)
 - Let $C' = \{\}$
 - For each node $u \in C$:
evaluate S with respect to u and add the result to C'
 - Evaluate P' with respect to C'

Path Expressions

Step

- Each step consists of (up to) 3 components



- Axis**
 - Specifies the **relation of nodes** to be selected for a given node u
- Node test**
 - Basic condition** the selected nodes must further satisfy
- Predicates**
 - Advanced conditions** the selected nodes must further satisfy

Path Expressions: Axes

Axis

- Specifies the relation of nodes to be selected for a given node

Forward axes

- `self`, `child`, `descendant(-or-self)`, `following(-sibling)`
- The order of the nodes corresponds to the document order

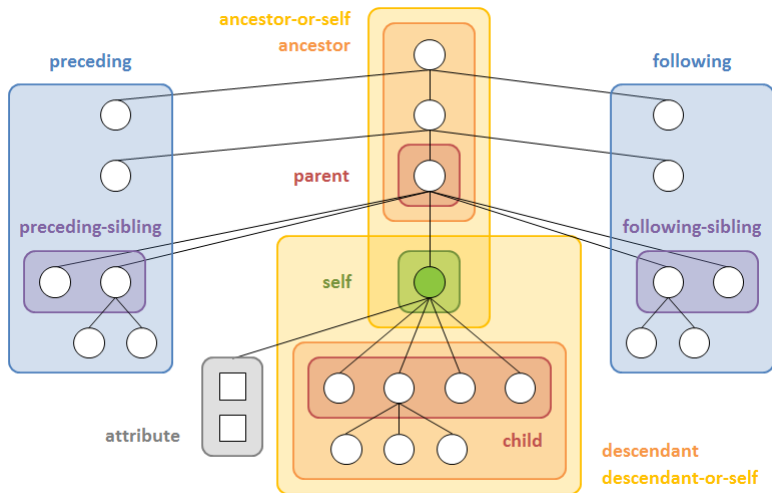
Reverse axes

- `parent`, `ancestor(-or-self)`, `preceding(-sibling)`
- The order of the nodes is reversed

Attribute axis

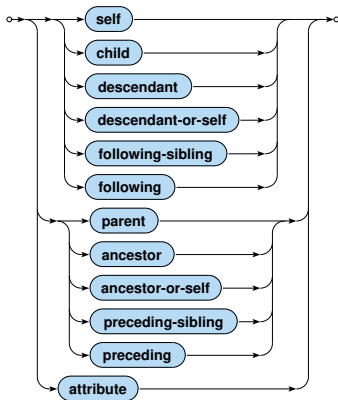
- `attribute` – the only axis that selects attributes

Path Expressions: Axes



Path Expressions: Axes

Available **axes**



Path Expressions

Examples

Axes

```
/child::movies
```

```
/child::movies/child::movie/child::title/child::text()
```

```
/child::movies/child::movie/attribute::year
```

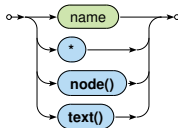
```
/descendant::movie/child::title
```

```
/descendant::movie/child::title/following-sibling::actor
```

Path Expressions: Node Tests

Node test

- Filters the nodes selected by the axis using basic tests



Available node tests

- name** – all elements / attributes with a given name
- *** – all elements / attributes
- node()** – all nodes (i.e. no filtering takes place)
- text()** – all text nodes

Path Expressions

Examples

Node tests

```
/movies
```

```
/child::movies
```

```
/descendant::movie/title/text()
```

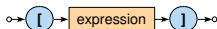
```
/movies/*
```

```
/movies/movie/attribute::*
```

Path Expressions: Predicates

Predicate

- Further filters the nodes using advanced conditions



Commonly used conditions

- Comparisons
- Path expressions
 - Handled as `true` when evaluated to a non-empty sequence
- Position testing
 - Based on the order as defined by the axis, starting with 1
- Boolean expressions, ...

When **multiple predicates** are provided, they must all be satisfied

Path Expressions

Examples

Predicates

```
/movies/movie[actor]
```

```
/movies/movie[actor]/title/text()
```

```
/descendant::movie[count(actor) >= 3]/title
```

```
/descendant::movie[@year > 2000 and @director]
```

```
/descendant::movie[@director][@year > 2000]
```

```
/descendant::movie/actor[position() = last()]
```

Path Expressions: Abbreviations

Multiple (mostly syntax) **abbreviations** are provided

- `.../...` (i.e. no axis is specified) $\Leftrightarrow$ `.../child::...`
- `.../@...` $\Leftrightarrow$ `.../attribute::...`
- `.../. ...` $\Leftrightarrow$ `.../self::node()...`
- `.../.. ...` $\Leftrightarrow$ `.../parent::node()...`
- `...//...` $\Leftrightarrow$ `.../descendant-or-self::node()/...`
- `.../[number]...` $\Leftrightarrow$ `.../[position() = number]...`

Path Expressions

Examples

Abbreviations

```
/movie/title
```

```
/child::movie/child::title
```

```
/movie/@year
```

```
/child::movie/attribute::year
```

```
/movie/actor[2]
```

```
/child::movie/child::actor[position() = 2]
```

```
//actor
```

```
/descendant-or-self::node()/child::actor
```

Path Expressions: Conclusion

Path expressions

- Absolute / relative

Step components

- Axis
- Node test
- Predicates

Path expression result

- **Result of the entire path expression** is the result of its last step
- Nodes are ordered in the **document order**
- **Duplicate nodes** are removed (based on the identity of nodes)

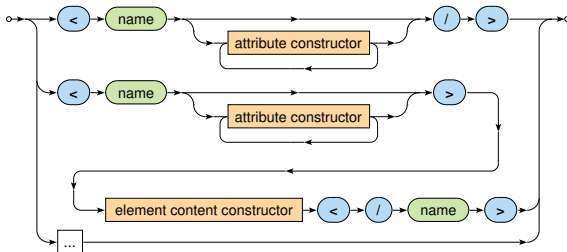
Constructors

Constructors

- Allow us to **create new nodes for elements, attributes, ...**
- **Direct constructor**
 - Well-formed XML fragment with **nested query expressions**
 - E.g.: `<movies>{ count(//movie) }</movies>`
 - **Names of elements and attributes must be fixed,**
their content can be dynamic
- **Computed constructor**
 - Special syntax
 - E.g.: `element movies { count(//movie) }`
 - **Both names and content can be dynamic**

Constructors

Direct constructor

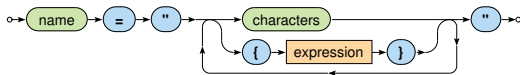


- Both **attribute value** and **element content** may contain an arbitrary number of **nested query expressions**
 - Enclosed by curly braces `{}`
 - Escaping sequences: `{ {` and `} }`

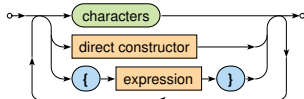
Constructors

Direct constructor

- Attribute



- Element content



Constructors

Example: Direct Constructor

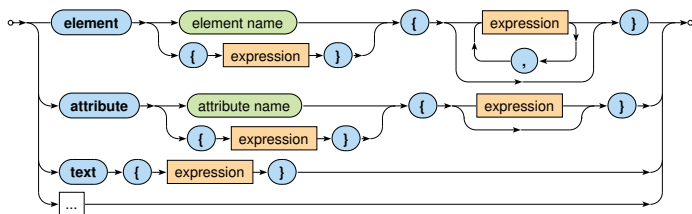
Create a summary of all movies

```
<movies>
  <count>{ count(//movie) }</count>
  {
    for $m in //movie
    return
      <movie year="{ data($m/@year) }">{ $m/title/text() }</movie>
  }
</movies>
```

```
<movies>
  <count>3</count>
  <movie year="2006">Vratné lahve</movie>
  <movie year="2000">Samotáři</movie>
  <movie year="2007">Medvídek</movie>
</movies>
```

Constructors

Computed constructor



Constructors

Example: Computed Constructor

Create a summary of all movies

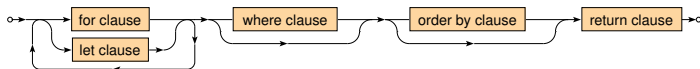
```
element movies {  
  element count { count(//movie) },  
  for $m in //movie  
  return  
    element movie {  
      attribute year { data($m/@year) },  
      text { $m/title/text() }  
    }  
}
```

```
<movies>  
  <count>3</count>  
  <movie year="2006">Vratné lahve</movie>  
  <movie year="2000">Samotáři</movie>  
  <movie year="2007">Medvídek</movie>  
</movies>
```

FLWOR Expressions

FLWOR expression

- Versatile construct allowing for **iterations over sequences**



Clauses

- `for` – selection of items to be iterated over
- `let` – bindings of auxiliary variables
- `where` – conditions to be satisfied (by a given item)
- `order by` – order in which the items are processed
- `return` – result to be constructed (for a given item)

FLWOR Expressions

Example

Find titles of movies with rating 75 and more

```
for $m in //movie
let $r := $m/@rating
where $r >= 75
order by $m/@year
return $m/title/text()
```

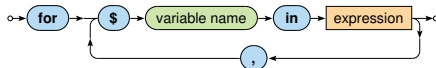
Samotáři

Vratné lahve

FLWOR Expressions: Clauses

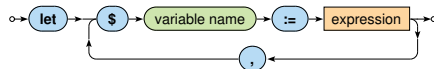
For clause

- Specifies a **sequence of values or nodes to be iterated over**
- Multiple sequences can be specified at once
 - Then the behavior is identical as when more single-variable for clauses would be provided



Let clause

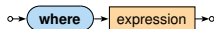
- Defines one or more auxiliary **variable assignments**



FLWOR Expressions: Clauses

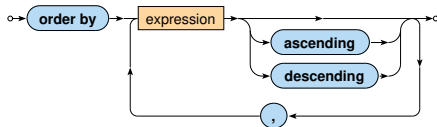
Where clause

- Allows to describe complex **filtering conditions**
- Items not satisfying the conditions are skipped



Order by clause

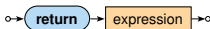
- Defines the **order in which the items are processed**



FLWOR Expressions: Clauses

Return clause

- **Defines how the result sequence is constructed**
- Evaluated once for each suitable item



Various supported **use cases**

- Querying, joining, grouping, aggregation, integration, transformation, validation, ...

FLWOR Expressions

Examples

Find titles of movies filmed in *2000* or later such that they have at most 3 actors and a rating above the overall average

```
let $r := avg(//movie/@rating)
for $m in //movie[@rating >= $r]
let $a := count($m/actor)
where ($a <= 3) and ($m/@year >= 2000)
order by $a ascending, $m/title descending
return $m/title
```

```
<title>Vratné lahve</title>
<title>Samotáři</title>
```

FLWOR Expressions

Examples

Find movies in which each individual actor starred

```
for $a in distinct-values(//actor)
return <actor name="{ $a }">
    {
        for $m in //movie[actor[text() = $a]]
        return <movie>{ $m/title/text() }</movie>
    }
</actor>
```

```
<actor name="Zdeněk Svěrák">
  <movie>Vratné lahve</movie>
</actor>
<actor name="Jiří Macháček">
  <movie>Vratné lahve</movie>
  <movie>Samotáři</movie>
  <movie>Medvídek</movie>
</actor>
...
```

FLWOR Expressions

Examples

Construct an HTML table with data about movies

```
<table>
  <tr><th>Title</th><th>Year</th><th>Actors</th></tr>
  {
    for $m in //movie
    return
      <tr>
        <td>{ $m/title/text() }</td>
        <td>{ data($m/@year) }</td>
        <td>{ count($m/actor) }</td>
      </tr>
  }
</table>
```

FLWOR Expressions

Examples

Construct an HTML table with data about movies

```
<table>
  <tr><th>Title</th><th>Year</th><th>Actors</th></tr>
  <tr><td>Vratné lahve</td><td>2006</td><td>2</td></tr>
  <tr><td>Samotáři</td><td>2000</td><td>3</td></tr>
  <tr><td>Medvídek</td><td>2007</td><td>2</td></tr>
</table>
```

Conditional Expressions

Conditional expression

- Note that the else branch is compulsory
 - Empty sequence () can be returned if needed



Example

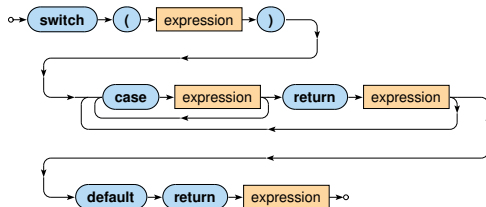
```
if (count(//movie) > 0)
then <movies>{ string-join(//movie/title, ", ") }</movies>
else ()
```

```
<movies>Vratné lahve, Samotáři, Medvídek</movies>
```

Switch Expressions

Switch

- **The first matching branch is chosen,** its return clause is evaluated and the result returned



- The default branch is compulsory and must be provided as the last option

Switch Expressions

Example

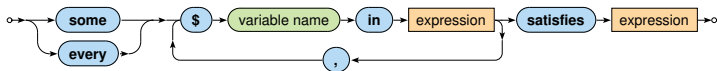
Return movies with aggregated information about their actors

```
xquery version "3.0";
for $m in //movie
return
  <movie>
    { $m/title }
    {
      switch (count($m/actor))
      case 0 return <no-actors/>
      case 1 return <actor>{ $m/actor/text() }</actor>
      default return <actors>{ string-join($m/actor, ", ") }</actors>
    }
  </movie>
```

Quantified Expressions

Quantifier

- Returns true if and only if...
 - in case of some **at least one item**
 - in case of every **all the items**
- ... of a given sequence/s **satisfy the provided condition**



Quantified Expressions

Examples

Find titles of movies in which *Ivan Trojan* played

```
for $m in //movie
where
  some $a in $m/actor satisfies $a = "Ivan Trojan"
return $m/title/text()
```

Samotáři
Medvídek

Find names of actors who played in all movies

```
for $a in distinct-values(//actor)
where
  every $m in //movie satisfies $m/actor[text() = $a]
return $a
```

Jiří Macháček

Comparison Expressions

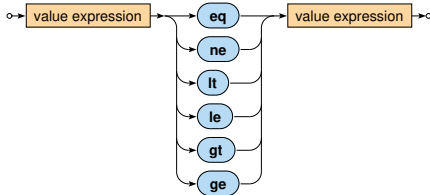
Comparisons

- **Value** comparisons
 - Two standalone values (singleton sequences) are expected to be compared
 - eq, ne, lt, le, ge, gt
- **General** comparisons
 - Two sequences of values are expected to be compared
 - =, !=, <, <=, >=, >
- **Node** comparisons
 - is – tests identity of nodes
 - <<, >> – test positions of nodes
 - Similar behavior as in case of value comparisons

Comparison Expressions

Value comparison

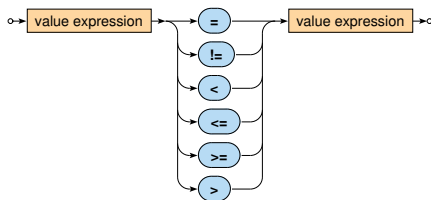
- **Both the operands are expected to be evaluated to singleton sequences**
 - Then these values are mutually compared in a standard way
- Empty sequence () is returned...
 - when at least one operand is evaluated to an empty sequence
- Type error is raised...
 - when at least one operand is evaluated to a longer sequence



Comparison Expressions

General comparison (existentially quantified comparisons)

- Both the operands can be evaluated to sequences of values of any length
- The result is true if and only if there exists at least one pair of individual values satisfying the given relationship



Comparison Expressions

Value and general comparisons

- **Atomization of values** – takes place automatically
 - Atomic values are preserved untouched
 - **Nodes are first transformed into strings with concatenated text values** they contain (even indirectly)
 - E.g.: `<movie year="2006">Vratné lahve</movie>` is atomized to a string `Vratné lahve`
 - Note that attribute values are not included!

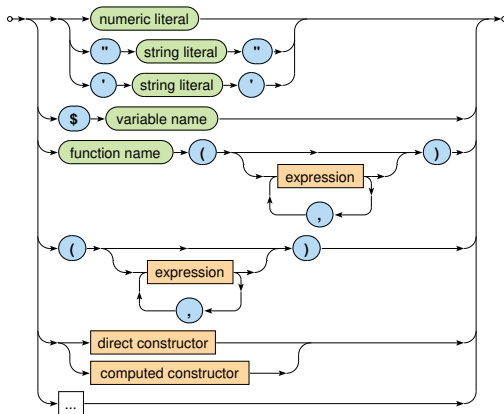
Comparison Expressions

Examples

- `1 le 2` $\Rightarrow$ `true`
- `(1) le (2)` $\Rightarrow$ `true`
- `(1) le (1,2)` $\Rightarrow$ `error`
- `(1) le ()` $\Rightarrow$ `()`
- `<a>5</a> eq <b>5</b>` $\Rightarrow$ `true`
- `1 < 2` $\Rightarrow$ `true`
- `(1) < (1,2)` $\Rightarrow$ `true`
- `(1) < ()` $\Rightarrow$ `false`
- `(0,1) = (1,2)` $\Rightarrow$ `true`
- `(0,1) != (1,2)` $\Rightarrow$ `true`

Primary Expressions

Primary expression



Lecture Conclusion

XPath expressions

- Absolute / relative paths
- Axes, node tests, predicates

XQuery expressions

- Constructors: direct, computed
- FLWOR expressions
- Conditional, quantified, comparison, ...