Lecture Notes

Course Number:  
CSC 513

Instructor:  
Dr. Singh

Lecture Number:  
6
XPath Queries (Selection Conditions)

- Attributes: //Song[@genre="jazz"]
- Text: //Song[starts-with(./group, "Led")]
- Existence of attribute: //Song[@genre]
- Existence of subelement: //Song[group]
- Boolean operators: and, not, or
- Set operator: union (|), analogous to choice
- Arithmetic operators: >, <, ...
- String functions: contains(), concat(), length(), starts-with(), ends-with()
- distinct-values()
- Aggregates: sum(), count()
THINK

songlist

Sons

Sgr

Son

Find siblings (or self)

Leaves: \$y / / x \ [\text{count}(=) = 0]

\$y1.../Sgr

\$y1.../x

\$y211
XPath Axes: 1

Axes are addressable node sets based on the document tree and the current node
- Axes facilitate navigation of a tree
- Several are defined
- Mostly straightforward but some of them order the nodes as the reverse of others
- Some captured via special notation
  - current, child, parent, attribute, ...
XPath Axes: 2

- **preceding:** nodes that precede the start of the context node (not ancestors, attributes, namespace nodes)

- **following:** nodes that follow the end of the context node (not descendants, attributes, namespace nodes)

- **preceding-sibling:** preceding nodes that are children of the same parent, in reverse document order

- **following-sibling:** following nodes that are children of the same parent
XPath Axes: 3

- **ancestor**: proper ancestors, i.e., element nodes (other than the context node) that contain the context node, in reverse document order
- **descendant**: proper descendants
- **ancestor-or-self**: ancestors, including self (if it matches the next condition)
- **descendant-or-self**: descendants, including self (if it matches the next condition)
XPath Axes: 4

- Longer syntax: `child::Song`  
- Some captured via special notation
  - `self::*`:
  - `child::node()`: `node()` matches all nodes
  - `preceding::*`
  - `descendant::*text()`
  - `ancestor::*Song`
  - `/descendant-or-self::*node()`, which abbreviates to `//`
  - Compare `/descendant-or-self::*Song[1]` (first descendant Song) and `//Song[1]` (first Songs (children of their parents))
XPath Axes: 5

- Each axis has a **principal node kind**
  - `attribute`: attribute
  - `namespace`: namespace
  - All other axes: element

- `*` matches whatever is the principal node kind of the current axis

- `node()` matches all nodes
XPointer

Enables pointing to specific parts of documents
- Combines XPath with URLs
- URL to get to a document; XPath to walk down the document
- Can be used to formulate queries, e.g.,
  - Song-URL#xpointer(/Song[@genre="jazz"])
  - The part after # is a fragment identifier
- Fine-grained addressability enhances the Web architecture

High-level “conceptual” identification of node sets
XQuery

- The official query language for XML, now a W3C recommendation, as version 1.0
- Given a non-XML syntax, easier on the human eye than XML
- An XML rendition, XqueryX, is in the works
XQuery Basic Paradigm

The basic paradigm mimics the SQL (SELECT–FROM–WHERE) clause

```xquery
for $x in doc('q2.xml')//Song
where $x/@lg = 'en'
return <English-Sgr name='{$x/Sgr/@name}' ti='{$x/@ti}' />
```
FLWOR Expressions

Pronounced “flower”

- **For**: iterative binding of variables over range of values
- **Let**: one shot binding of variables over vector of values
- **Where** (optional)
- **Order by** (sort: optional)
- **Return** (required)

Need at least one of **for** or **let**
XQuery For Clause

The `for` clause

- Introduces one or more variables
- Generates possible bindings for each variable
- Acts as a mapping functor or iterator
  - In essence, all possible combinations of bindings are generated: like a Cartesian product in relational algebra
  - The bindings form an ordered list
XQuery Where Clause

The **where** clause

- Selects the combinations of bindings that are desired
- Behaves like the **where** clause in SQL, in essence producing a join based on the Cartesian product
XQuery Return Clause

The `return` clause

- Specifies what node-sets are returned based on the selected combinations of bindings
XQuery Let Clause

The `let` clause

- Like `for`, introduces one or more variables
- Like `for`, generates possible bindings for each variable
- Unlike `for`, generates the bindings as a list in one shot (no iteration)
XQuery Order By Clause

The **order by** clause

- Specifies how the vector of variable bindings is to be sorted before the return clause
- Sorting expressions can be nested by separating them with commas
- Variants allow specifying
  - *descending* or *ascending* (default)
  - *empty greatest* or *empty least* to accommodate empty elements
  - stable sorts: **stable order by**
  - collations: **order by $t collation**
    - collation-URI: (obscure, so skip)
XQuery Positional Variables

The **for** clause can be enhanced with a positional variable

- A positional variable captures the position of the main variable in the given **for** clause with respect to the expression from which the main variable is generated

- Introduce a positional variable via the **at $var** construct

```
for $x in (5, 9, 8, 7, 6) at $i

x = 9  i = 1
8  2
```
```python
for $x$ in doc(\ldots) // item[1]
for $y$ in doc(\ldots) // item[2]
for $x$ at $pos$ in doc(\ldots) // item
order by $pos$ descending
return $x$
```

```
SELECT $t$.id
FROM student $t$
```