XML and Databases

Lecture 6
Node Selecting Queries: XPath 1.0

Sebastian Maneth
NICTA and UNSW

Outline

1. XPath Data Model: 7 types of nodes
2. Simple Examples
3. Location Steps and Paths
4. Value Comparison, and Other Functions

Outline

1. XPath Data Model: 7 types of nodes

XPath

- Query language to select (a sequence of) nodes of an XML document
- W3C Standard
- Most important XML query language: used in many other standards such as XQuery, XSLT, XPointer, XLink, ...
- Supported by every modern web browser for JavaScript processing!
- Cave: version 2.0 is considerably more expressive than 1.0

We study XPath 1.0

Terminology: Instead of XPath "query" we often say XPath expression.
(An expression is the primary construction of the XPath grammar; it matches the production Expr of the XPath grammar.)

Outline - Assignments

2. SAX Parse into memory structure: Tree and DAG
3. Map XML into RDBMS ➔ 29. April
4. XPath evaluation ➔ 17. May
5. XPath into SQL Translation ➔ 31. May

Outline - Lectures

1. Introduction to XML, Encodings, Parsers
2. Memory Representations for XML: Space vs Access Speed
3. RDBMS Representation of XML
4. DTDs, Schemas, Regular Expressions, Ambiguity
5. XML Validation using Automata
6. Node Selecting Queries: XPath
7. Tree Automata for Efficient XPath Evaluation, Parallel Evaluation
8. XPath Properties: backward axes, containment test
9. Streaming Evaluation: how much memory do you need?
10. XPath Evaluation using RDBMS
11. XSLT – stylesheets and transform
12. XQuery – XML query language
13. Wrap up, Exam Preparation, Open questions, etc

XPath Data Model

Evaluate Q on D (in XPath data model) XPath Query Q XML document D sequence of result nodes

Document D is modeled as a tree.

 THERE ARE SEVEN TYPES OF NODES in the XPath Data Model:

- root nodes
- element nodes
- text nodes
- attribute nodes
- namespace nodes
- processing instruction nodes
- comment nodes
XPath Data Model

Document D is modeled as a tree.

There are seven types of nodes in the XPath Data Model:

- root nodes
- element nodes
- text nodes
- attribute nodes
- namespace nodes
- processing instruction nodes
- comment nodes

For rest of lecture: this is all you need to know about XML nodes! ☺

5.2.1 Unique IDs

An element node may have a unique identifier (ID).

- Value of the attribute that is declared in the DTD as type ID.
- No two elements in a document may have the same unique ID.
- If an XML processor reports two elements in a document as having the same unique ID (which is possible only if the document is invalid) then the second element in doc. order must be treated as not having a unique ID.

Note: If a document has no DTD, then no element will have a unique ID.

Simple Examples

In abbreviated XPath syntax.

Q0: / selects the document root (always the parent of the document element)

Q1: //book/year selects the document root (always the parent of the document element)

Note: XPath Evaluators usually return the full subtree of the selected node.
Simple Examples

In abbreviated syntax:
Q1: /bib/book/year

Document:
<bib>
  <book>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
    <title>Foundations of Databases</title>
    <year>1995</year>
  </book>
  <book>
    <author>Ullmann</author>
    <title>Principles of Database and Knowledge Base Systems</title>
    <year>1998</year>
  </book>
</bib>

Result of query Q1:
- Document element, if labeled bib
- child nodes that are labeled book
- child nodes that are labeled year

In abbreviated syntax.
Q2: //author

Document:
<bib>
  <book>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
    <title>Foundations of Databases</title>
    <year>1995</year>
  </book>
  <book>
    <author>Ullmann</author>
    <title>Principles of Database and Knowledge Base Systems</title>
    <year>1998</year>
  </book>
</bib>

Result of query Q2:
- sequence of (element) nodes (N1, N2, N3, N4)

In abbreviated syntax.
Q3: /a/b//d

Result of query Q3:
- ALL d-nodes in these subtrees

In abbreviated syntax.
Q4: /a/b//d

Result of query Q4:
- ALL d-nodes in these subtrees
Simple Examples
In abbreviated syntax.

Q4: /*/c

Simple Examples
In abbreviated syntax.

Q5: //c

Simple Examples
In abbreviated syntax.

Q6: //*

Abbreviations, so far
In abbreviated syntax.

// is abbreviation for /descendant-or-self::node()/
. is abbreviation for parent::node()
Æ Child and descendant-or-self are only 2 out of 12 possible axes.
An “axis” is a sequence of nodes. It is evaluated relative to a context-node.
Other axes:
Æ descendant
Æ parent
Æ ancestor-or-self
Æ ancestor
Æ following-sibling
Æ preceding-sibling
Æ attribute
Æ following
Æ preceding
Æ self

Examples: Predicates
In abbreviated syntax.

Q7: //c[./b] "has b-child" (context-nodes are all c-nodes...)
Examples: Predicates

In abbreviated syntax.
Q8: //c[./b]/d

"has b-child"

Q9 selects c-nodes that "have a b-child AND a d-child"

More direct way: //c[./b and ./d]

(same as //c[./b] on "this" tree...!)

Examples: Predicates

In abbreviated syntax.
Q9: //c[./b]/d/..

"has b-child"

select parent(s)

of context-node(s)

Q9 selects c-nodes that "have a b-child AND a d-child"

More direct way: //c[./b and ./d]

(same as //c[./b] on "this" tree...!)

Examples: Predicates (or “Filters”)

In abbreviated syntax.
//c[.a and d]

A “Filter”

evaluates to true/false

c-nodes that “have a b-child AND a d-child”

Examples: Predicates (or “Filters”)

In abbreviated syntax.
//c[.a and d]

A “Filter”

evaluates to true/false

Question

How to only select the other c-node?

Can use "not( ... )" in a filter!

//c[not(b)]

"does not have a b-child"
Examples: Predicates

In abbreviated syntax.
`//c[b and d]` evaluates to true/false

Question
How to only select the other c-node?

Many more possibilities, of course:
`//c[parent::b]`  
`//c[../b]`  
`//c[../../b]`

Can use "not( ... )" in a filter!
`//c[not(b)]`

Cave: what does `//c[not(b)]` give?

Question
How to only select the other c-node?

Many more possibilities, of course:
`//c[parent::b]`  
`//c[../b]`  
`//c[../../b]`

Can use "not( ... )" in a filter!
`//c[not(b)]`

Can you say "c-node that has only d-children"?

YES! needs a bit of logic...
`//c[not(child::*[not(self::d)])]`

"not the case that all children are not labeled d"
holds if and only if
"all children are labeled d"

Duplicate elimination
`//c[not(b)]/d/..`
context-nodes for parent selection `(/..)`
Examples: Predicates

In abbreviated syntax.

```xml
//c[not(b)]
```

```
//c[not(child::*[not(self::d)])]
```

- "not the case that all children are not labeled d"
- "all children are labeled d"

Duplicate elimination

```
//c[not(b)]/ancestor::*
```

- "context-nodes for ancestor selection"

```
//c[not(b)]/ancestor::*
```

- "Equivalent one, without use of ancestor?"

```
//*[.//c[not(b)]]
```

- "No use of ancestor?"

```
//*[descendant-or-self::c[not(b)]]
```

- "How to select the c-node?"

```
//*[.//c[not(b)] or not(child::*[not(self::d)]) and .]
```

- "only d-children"
- "has child (not leaf)"

More Details

Evaluate Q on D (in XPath data model)

XML document D

- sequence of result nodes

NOT correct (at least not for intermediate expr's)

An expression evaluates to an object, which has one of the following four basic types

- node-set (an unordered collection of nodes w/o duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of UCS characters)
Location Steps & Paths

A Location Path is a sequence of Location Steps

**Location Paths**

1. LocationPath ::= RelativeLocationPath | AbsoluteLocationPath

2. AbsoluteLocationPath ::= '/' RelativeLocationPath | AbbreviatedAbsoluteLocationPath

3. RelativeLocationPath ::= Step | RelativeLocationPath * Step | AbbreviatedRelativeLocationPath

**Location Steps**

4. Step ::= AxisSpecifier NodeTest Predicate* | AbbreviatedStep

5. AxisSpecifier ::= AxisName '::' | AbbreviatedAxisSpecifier

**Filters** (aka predicates, (filter) expressions)
evaluate to true/false

XPath queries, evaluated with context-node = current node

Boolean operators: and, or

Empty string/sequence are converted to false

---

**Axis** = a sequence of nodes (is evaluated relative to context-node)

**Forward Axes:**

- self
- child
- descendant-or-self
- descendant
- following
- following-sibling

**Backward Axes:**

- parent
- ancestor
- ancestor-or-self
- preceding
- preceding-sibling

---

**Initial Context** will be the root node

**Example**

child::text() “select all text node children of the context node”

the nodetest node() is true for any node.

attribute::* “select all attributes of the context node”
Location Steps & Paths

Axis = a sequence of nodes  (is evaluated relative to context-node)

Forward Axes:
- self
- child
- descendant-or-self
- descendant
- following
- following-sibling

Backward Axes:
- parent
- ancestor
- ancestor-or-self
- preceding
- preceding-sibling
- attribute

Application determines the Initial Context.

If path starts with "/", then Initial Context has:
- context-node = root node
- context-position = context-size = 1
Location Path Semantics

→ A Location Path $P$ is a sequence of Location Steps

\[ a_1::n_1[F_1_1][F_1_2]...[F_1_n1] \]
\[ /a_2::n_2[F_2_1][F_2_2]...[F_2_n2] \]
\[ /a_m::n_m[F_m_1][F_m_2]...[F_m_nm] \]

$S_0 =$ initial sequence of context-nodes

(1) (to each) context-node $N$ in $S_0$, apply axis $a_1$ gives sequence $S1$ of nodes
→ test $n_1$ evaluates to false
→ any of filters $F_1_1,...,F_1_n1$ evaluate to false.

Apply steps (1)&(2) for step 2, to obtain from $S1$ the sequence $S2$
3, $S2$, $S3$

...$m$, $S(m-1)$, $Sm$ = result of $P$

No Looking Back

Backward Axes are not needed!!

→ possible to rewrite most XPath queries into equivalent ones that do not use backward axes.

Very nice result!!

Can you see how this could be done?

→ We saw an example of removing ancestor axis. But, of course the rewritten query must be the same ON EVERY possible tree!!

Questions
how much larger does the query get, when you remove all backward axis?
Is this useful for efficient query evaluation?!

Attribute Axis

How to
→ test attribute nodes

Examples

//attribute::*

Result:

$b=1$
$\text{a"1"}$
$\text{a"2"}$
$\text{a"1.0"}$

Remember, these are just NODES.

/attribute::"." gives same result

And /attribute::a, gives

Example:

$d>c>d>b>c>d>d$ (selects the two red nodes)

Attribute Axis & Value Tests

How to
→ test attribute values

Examples

//*[attribute::a=1] (selects the two red nodes)

Watch out

/*[attribute::a="1"] only gives
/*[attribute::a="1.0"] only gives

string comparison

attribute:: is abbreviated by @
Attribute Axis & Value Tests

How to test attribute values

Examples
- 
  /'[attribute::a=1]' (selects the two red nodes)

Watch out
- 
  /'[attribute::a="1"]' only gives
  /'[attribute::a="1.0"]' only gives

<table>
<thead>
<tr>
<th>attribute::</th>
</tr>
</thead>
<tbody>
<tr>
<td>is abbreviated by @</td>
</tr>
</tbody>
</table>

selects both c-nodes
selects only left c-node
selects what? (hint: "=" is string comp. here)

Text Nodes

How to test text nodes & values

Result:
- foo
- ""foo"
- "foo"
- "foo"
- "bar"

Question:
What is the result for

Useful Functions (on Booleans)

true():boolean
false():boolean
lang(string):boolean

Useful Functions (on Node Sets)

count

What is the result?

Useful Functions (on Node Sets)

What is the result?

Tests in Filters

- or
- and
- c, d
- <=, <, >=, >

The operators are all left associative.

For example, 3 > 2 > 1 is equivalent to (3 > 2) > 1, which evaluates to false.

But, 3 > 2 > 0.9 evaluates to true. Can you see why?

For two strings u, v

- unless both u and v are numbers,
  
  "1.0"="1" evaluates to true.

- u<>v
  u<>v
  u>v
  always return false!
- for other objects, conversion depends on type

Useful even for use with self-axis:

- child::*[self::chapter or self::appendix]
Useful Functions (on Node Sets)

- `count`:
  - Counts number or results
  - Example: `a[count(*[text()=//b/text()])=2]` (same result as: `a[count(*[text()=//b/text()])=2]`)

- `last()`:
  - Returns context-size from the evaluation context

- `position()`:
  - Returns context-position from the evaluation context

- `[@position()=2]` (same as `[@position()=2 and ./b]`)

- `[@position()=last()-1]` (same as `[@position()=last()-1 and ./text()="foo"]`)
Useful Functions (on Node Sets)

- `last()` returns context-size from the evaluation context
- `position()` Returns context-position from the eval. context

Useful:

- `child::*[self::chapter or self::appendix][position()=last()]` selects the last chapter or appendix child of the context node

### Abbreviation:

- `child::*[self::chapter or self::appendix][position()=last()]` can be abbreviated as `self::*[last()]`

### Example:

```
books
  book
  book
  book
  ... book
```

- How do you select the last 20 book-children of books?

```
/last[20]
```

---

### Diagram:

```
books
  book
  book
  book
  ... book
```

How to select the last 20 book-children of books?
Useful Functions (on Node Sets)

- `last()`: number
  - returns context-size from the evaluation context

- `position()`: number
  - returns context-position from the eval. Context

- `id(object)`: node-set
  - selects the element with unique ID `foo`

- `local-name(node-set?)`: string
  - returns the local part of the expanded-name of the node

- `namespace-uri(node-set?)`: string
  - returns the namespace URI of the expanded-name of the node

- `name(node-set?)`: string
  - returns a string containing a QName representing the expanded-name of the node

Useful Functions (Strings)

The `string-value` of an element node is the concatenation of the `string-values` of all text node `descendants` in document order.

```
//a[*[1]=*[2]]
//a[b/d = c/d]
```

XPath 2.0 has much clearer comparison operators!!

Careful with equality ("=")

```
//a[b/d = c/d]
```

there exists a node in the node set for `b/d` with same string value as a node in node set `c/d`

What about `//a[b/d != c/d]`?
Useful Functions (Strings)

The `string-value` of an element node is the concatenation of the `string-values` of all text node descendents in document order:

```xml
// *[.@="foo"]
// *[.@="foobar"]
```

- `normalize-space(" foo   bar a ") = "foo bar a"
- `translate("bar","abc","ABC") = BAr`

**NOTE:** The `translate` function is not a sufficient solution for case conversion in all languages.

Useful Functions (Numbers)

- `number(object):number`
  - Converts argument to a number
    - the boolean true is converted to 1, false is converted to 0
    - a string that consists of optional whitespace followed by an optional minus sign followed by a `number` followed by whitespace is converted to the IEEE 754 number that is nearest to the mathematical value represented by the string.

- `sum(node-set):number`
  - returns sum, for each node in the argument node-set, of the result of converting the `string-value` of the node to a number

- `floor(number):number`
- `ceiling(number):number`
- `round(number):number`

Operators on Numbers

+,-,*,div,mod

Display Number Result…

```
// *[text()=7 mod (count(//b)+2))]/text()
```


Display Number Result…

Similar for arbitrary large numbers / booleans, node-sets… Try it…

XPath Query Evaluation

How to implement?
How expensive? complexity?
What are the most difficult queries?

Next time

Efficient Algorithms: which queries run how fast?
First, focus on navigational queries: only /, //, label-test, [ filters ]
(techniques for value comparison/queries already well-known from rel. DB’s…)

Experiments with current systems

means year 2003…

Next 4 slides from
Georg Gottlob and Christoph Koch "XPath Query Processing."
Invited tutorial at DBPL 2003
http://www.dbai.tuwien.ac.at/research/emitaskforce/xpath-tutorial1.ppt.gz
XPath Query Evaluation

Static Methods (used, e.g., for Query Optimization…)

Given XPath queries Q1, Q2:

→ Is result set of Q1 included in result set of Q2?
→ Are result sets equal?
→ Is their intersection empty?

for all possible documents

(probably we will look at this in Lecture 8 or 9)

Simple Examples

Is

```xml
//c[count(d)=count(*)]
```

equivalent to

```xml
//*[not(child::*[not(self::d)])]
```

on all possible trees?
END
Lecture 6