XML and Databases

Lecture 7
Efficient XPath Evaluation

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2009

Outline

1. Top-Down Evaluation of simple paths
2. Node Sets only: Core XPath
3. Bottom-Up Evaluation of Core XPath
4. Polynomial Time Evaluation of Full XPath

1. Top-Down Evaluation of Simple Paths

Simple paths are of the form
(1) //tag_1/tag_2/…/tag_n
(2) //tag_1/tag_2/…/tag_{n-1}/text()

Selects any node which is (1) labeled tag_n (2) a text node and
is child of a node labeled tag_{n-1}
is child of a node labeled tag_{n-2}...
is child of a node labeled tag_1

Examples
//author/last = select all last names of authors
//strip/characters/character/text() = select all character names from a DilbertML document

(return selected nodes in document order.)

1. Top-Down Evaluation of Simple Paths

Æ evaluate in one single pre-order traversal (using a stack)

Æ push current match position p for every startElement (except for the root node)

Æ query match position: p = 1

Æ partial match. If element name was different from “a”, then p would remain equal to 1)

Æ push(p)

Æ query match position: p = 2

Æ current node is a match!

Æ mark it as match/result

Æ push(result)

Æ p = 1

Æ push(p)

Æ query match position: p = 2

Æ current node is a match!

Æ mark it as match/result

Æ push(result)

Æ p = 1
1. Top-Down Evaluation of Simple Paths

- evaluate in one single pre-order traversal (using a stack)

```
//a/b   =Q
```

query match position: \( p = 2 \)

```
[startElement( a )]
[startElement( b )]
```

\( p=2=\text{length}(Q) \), thus,
- current node is a match!
- Mark it as match/result
- push() \( p \)
- \( p = 1 \)

```
[startElement( a )]
[startElement( b )]
[startElement( c )]
```

```
[endElement( a )]
```

\( p=\text{pop()} \)

```
[startElement( a )]
[startElement( c )]
```

```
[endElement( a )]
```

\( p=\text{pop()} \)

```
[startElement( a )]
```

Question: Why is \( p \) set to 1? What if query was //a/a?

```
[startElement( a )]
[startElement( c )]
```

```
[endElement( a )]
```

\( p=\text{pop()} \)

```
[startElement( a )]
[startElement( a )]
[startElement( b )]
```

\( p=\text{pop()} \)

```
[startElement( a )]
[startElement( a )]
[startElement( b )]
[startElement( a )]
```

\( p=\text{pop()} \)

```
[startElement( a )]
[startElement( b )]
[startElement( a )]
[startElement( c )]
```

\( p=\text{pop()} \)
1. Top-Down Evaluation of Simple Paths

→ evaluate in one single pre-order traversal (using a stack)

(//a/b = Q

query match position: p = 1

[startElement( a )]
[startElement( b )]
[startElement( a )]
[endElement( a )]

Æ

evaluate in one single pre-order traversal (using a stack)

Thus, push(p) and p = p+1 = 2

p=2=length(Q), thus, current node is a match!
→ Mark it as match/result
→ push(p)
→ p = 1

Æ

evaluate in one single pre-order traversal (using a stack)

p=2=length(Q), thus, current node is a match!
→ Mark it as match/result
→ push(p)
→ p = 1

Æ

evaluate in one single pre-order traversal (using a stack)

Æ

evaluate in one single pre-order traversal (using a stack)
1. Top-Down Evaluation of Simple Paths

→ evaluate in one single pre-order traversal (using a stack)

// a/b = Q

query match position: p = 2

[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
[startElement(c)]
[endElement(c)]
[startElement(b)]
[startElement(c)]
push(1)
[endElement(c)]
p = pop() = 1
[startElement(a)]
push(1)
[endElement(a)]
p = pop() = 1
[endElement(b)]
p = pop() = 2
[endElement(a)]
p = pop() = 1
[endElement(b)]
p = pop() = 2
[startElement(b)]
match

Linear time: O(#Nodes) = O(|D|)

Size of document

Even: Streaming Algorithm!☺

→ No need to store the document!!

Can evaluate on SAX event stream.

But, to print result subtrees we need an output buffer☺

Æ evaluate in one single pre-order traversal (using a stack)
1. Top-Down Evaluation of Simple Paths

- evaluate using one single pre-order traversal (using a stack)

```
/ a/ b/ a/ c
```

NOT equal. p>4

What to do next?

```
/ a/ b/ a/ c
```

Postfix (ending) of what we have seen, is prefix (beginning) of the query!!

2. Core XPath

- all 12 axes
- all node tests (but, here, we will simply talk about element nodes only)
- filters with logical operations: and, or, not

E.g.: //descendant::a/child::b[ child::c/child::d or not( following::* ) ]

Full XPath additionally has:
- Node set comparisons & operations (e.g., =, count)
- Order functions (first, last, position)
- Numerical operations (sum, +, -, div, mod, round, etc.) and corresponding comparisons (<=, <, >, <=, >=)
- String operations (contains, starts-with, translate, string-length, etc.)
Axis Evaluation

**Axis = Node Set** (evaluated relative to context-node)

- Forward Axes: Maps a Node Set to a Node Set
  - parent
  - descendant
  - self
  - following-sibling
  - descendant-or-self
  - following

- Backward Axes: Maps a Node Set to a Node Set
  - child
  - descendant
  - ancestor
  - previous-sibling
  - parent
  - ancestor-or-self

**Node Set represented as bit-field**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

**Idea**

- No node is visited more than once!
- Look-up parents, check if we are in result set already.

**Axis Evaluation**

**Axis = Node Set** (evaluated relative to context-node)

- Forward Axes: Maps a Node Set to a Node Set
  - parent
  - descendant
  - self
  - following-sibling
  - descendant-or-self
  - following

- Backward Axes: Maps a Node Set to a Node Set
  - child
  - descendant
  - ancestor
  - previous-sibling
  - parent
  - ancestor-or-self

**Node Set represented as bit-field**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

**Idea**

- No node is visited more than once!
- Look-up parents, check if we are in result set already.

**Axis Evaluation**

**Axis = Node Set** (evaluated relative to context-node)

- Forward Axes: Maps a Node Set to a Node Set
  - parent
  - descendant
  - self
  - following-sibling
  - descendant-or-self
  - following

- Backward Axes: Maps a Node Set to a Node Set
  - child
  - descendant
  - ancestor
  - previous-sibling
  - parent
  - ancestor-or-self

**Node Set represented as bit-field**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

**Idea**

- No node is visited more than once!
- Look-up parents, check if we are in result set already.

**Axis Evaluation**

**Axis = Node Set** (evaluated relative to context-node)

- Forward Axes: Maps a Node Set to a Node Set
  - parent
  - descendant
  - self
  - following-sibling
  - descendant-or-self
  - following

- Backward Axes: Maps a Node Set to a Node Set
  - child
  - descendant
  - ancestor
  - previous-sibling
  - parent
  - ancestor-or-self

**Node Set represented as bit-field**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

**Idea**

- No node is visited more than once!
- Look-up parents, check if we are in result set already.
Axis Evaluation

Axis = Node Set (evaluated relative to context-node)

Idea
→ No node is visited >once!

\[ \text{e.g.: ancestor(\{5, 8, 9\})} \]

look-up parents, check if we are in result set already..

Result Node Set
\[ \{12|3|4|5|6|7|8|9|10|11\} \]

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)

Idea
→ No node is visited >once!

\[ \text{e.g.: ancestor(\{5, 8, 9\})} \]

look-up parents, check if we are in result set already..

Result Node Set
\[ \{12|3|4|5|6|7|8|9|10|11\} \]

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)

Idea
→ No node is visited >once!

\[ \text{e.g.: ancestor(\{5, 8, 9\})} \]

look-up parents, check if we are in result set already..

Result Node Set
\[ \{12|3|4|5|6|7|8|9|10|11\} \]

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)

Idea
→ No node is visited >once!

\[ \text{e.g.: ancestor(\{5, 8, 9\})} \]

look-up parents, check if we are in result set already..

Result Node Set
\[ \{12|3|4|5|6|7|8|9|10|11\} \]

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)

Idea
→ No node is visited >once!

\[ \text{e.g.: ancestor(\{5, 8, 9\})} \]

look-up parents, check if we are in result set already..

Result Node Set
\[ \{12|3|4|5|6|7|8|9|10|11\} \]
Axis Evaluation

Axis = Node Set (evaluated relative to context-node)

Similarly:
For all other axes!
Forward-axes only:
binary (top-down) tree encoding
provides easy linear time evaluation!

Recall:
to access parent / ancestors on binary tree, keep dynamically list of all ancestors.

Idea
No node is visited more!
e.g.: ancestor([5, 8, 9])
look-up parents, check if we are in result set already...

Result Node Set
1 2 3 4 5 6 7 8 9 10
1 1 0 0 0 1 0 0 0 0 + 6 parent look-ups.
+ 5 result look-ups.

+ 6 parent look-ups.
+ 5 result look-ups.

Question
do you see how this works for e.g., descendant axis?

Result Node Set
1 2 3 4 5 6 7 8 9 10
1 1 0 0 0 1 0 0 0 0

Forward-axes only:
binary (top-down) tree encoding
provides easy linear time evaluation!

For all other axes!

Similarly:

Recall:
to access parent / ancestors on binary tree, keep dynamically list of all ancestors.

Idea
No node is visited more!
e.g.: ancestor([5, 8, 9])
look-up parents, check if we are in result set already...

Result Node Set
1 2 3 4 5 6 7 8 9 10
1 1 0 0 0 1 0 0 0 0

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)

Question
do you see how this works for e.g., descendant axis?

descendant(node) = (first-child | next-sibling)* (first-child(node))
descendant({ node_1, node_2, ... , node_k }) =
repeat{
pick node N in S;
(for N's descendants M in pre-order)
{
if (not(M in result set))
add(M to result set) else break;
}
}

Question
do you see how this works for e.g., descendant axis?

descendant(node) = (first-child | next-sibling)* (first-child(node))
descendant({ node_1, node_2, ... , node_k }) =
repeat{
pick node N in S;
(for N's descendants M in pre-order)
{
if (not(M in result set))
add(M to result set) else break;
}
}

Example

descendant(node) = (first-child | next-sibling)* (first-child(node))
Note Set
\[ 1 2 3 4 5 6 7 8 9 10 \]
\[ 1 0 0 0 0 0 0 0 0 0 \]

descendant(1) =
\[ (fc | ns)^(first-child(1)) \] =
\[ (fc | ns)(2) \] =
\[ {2} + \]

Result Node Set
1 2 3 4 5 6 7 8 9 10
0 0 0 0 0 0 0 0 0 0

Example

descendant(node) = (first-child | next-sibling)* (first-child(node))
Note Set
\[ 1 2 3 4 5 6 7 8 9 10 \]
\[ 1 0 0 0 0 0 0 0 0 0 \]

descendant(1) =
\[ (fc | ns)^(first-child(1)) \] =
\[ (fc | ns)(2) \] =
\[ {2} + \]

Result Node Set
1 2 3 4 5 6 7 8 9 10
0 1 0 0 0 0 0 0 0 0

Example

descendant(node) = (first-child | next-sibling)* (first-child(node))
Note Set
\[ 1 2 3 4 5 6 7 8 9 10 \]
\[ 1 0 0 0 0 0 0 0 0 0 \]

descendant(1) =
\[ (fc | ns)^(first-child(1)) \] =
\[ (fc | ns)(2) \] =
\[ {2} + \]

Example

descendant(node) = (first-child | next-sibling)* (first-child(node))
Note Set
\[ 1 2 3 4 5 6 7 8 9 10 \]
\[ 1 0 0 0 0 0 0 0 0 0 \]

descendant(1) =
\[ (fc | ns)^(first-child(1)) \] =
\[ (fc | ns)(2) \] =
\[ {2} + \]

Result Node Set
1 2 3 4 5 6 7 8 9 10
0 1 0 0 0 0 0 0 0 0

This example comes from
Georg Gottlob and Christoph Koch "XPath Query Processing".
Invited tutorial at DBPL 2003
http://www.dbai.tuwien.ac.at/research/omdeed/on/xml-tutorials1.ppt.gz

Example

descendant(node) = (first-child | next-sibling)* (first-child(node))
Note Set
\[ 1 2 3 4 5 6 7 8 9 10 \]
\[ 1 0 0 0 0 0 0 0 0 0 \]

descendant(1) =
\[ (fc | ns)^(first-child(1)) \] =
\[ (fc | ns)(2) \] =
\[ {2} + \]

Result Node Set
1 2 3 4 5 6 7 8 9 10
0 1 0 0 0 0 0 0 0 0

Example

descendant(node) = (first-child | next-sibling)* (first-child(node))
Note Set
\[ 1 2 3 4 5 6 7 8 9 10 \]
\[ 1 0 0 0 0 0 0 0 0 0 \]

descendant(1) =
\[ (fc | ns)^(first-child(1)) \] =
\[ (fc | ns)(2) \] =
\[ {2} + \]

Result Node Set
1 2 3 4 5 6 7 8 9 10
0 1 0 0 0 0 0 0 0 0

Example

descendant(node) = (first-child | next-sibling)* (first-child(node))
Note Set
\[ 1 2 3 4 5 6 7 8 9 10 \]
\[ 1 0 0 0 0 0 0 0 0 0 \]

descendant(1) =
\[ (fc | ns)^(first-child(1)) \] =
\[ (fc | ns)(2) \] =
\[ {2} + \]

Result Node Set
1 2 3 4 5 6 7 8 9 10
0 1 0 0 0 0 0 0 0 0
Example
descendant( node ) =
(first-child | next-sibling)* (first-child( node ))
descendant( { 1 } ) =
(fc | ns)* (first-child( { 1 } )) =
(fc | ns)*({ 2 }) =
{ 2 } +
{ 3, 6 } +
{ 4, 7 } +
Result Node Set
0 1 1 1 0 1 1 0 0 0

Example
descendant( node ) =
(first-child | next-sibling)* (first-child( node ))
descendant( { 1 } ) =
(fc | ns)* (first-child( { 1 } )) =
(fc | ns)*({ 2 }) =
{ 2 } +
{ 3, 6 } +
{ 4, 7 } +
{ 5, 8, 10 } +
{ 9 } +
Result Node Set
0 1 1 1 1 1 1 1 1 1
Core XPath

- all 12 axes
- all node tests (but, here, we will simply talk about element nodes only)
- filters with logical operations: and, or, not

E.g. //descendant::a/child::b

Types
- Node Sets
- Booleans

3. Bottom-Up Evaluation of Core XPath

With respect to query-tree (parse tree)
NOT with respect to document tree!
(algorithm f. simple paths is top-down wrt document tree)

For Core XPath we only need Node Set operations!!

-axis( Set1 ) = Set2
-∪( Set1, Set2 ) = Set3 union of Set1 and Set2
-Å( Set1, Set2 ) = Set3 intersection of Set1 and Set2
--( Set1, Set2 ) = Set3 everything in Set1 but not in Set2
-lab(a) = Set1 all nodes labeled by a

//descendant::a/child::b

\[ \text{child::c/child::d or not(following::*)} \]

lab(a) = \{ 2, 6, 8 \}
lab(b) = \{ 3, 7, 9 \}
lab(c) = \{ 1, 4 \}
lab(d) = \{ 5 \}

\( \text{Å}( \text{descendant(} \text{root} \text{), lab(a)}) \) = \{ 2, 6, 8 \}
\( \text{Å}( \text{lab(b)}, \text{lab(c)}) = \{ 3, 7, 9 \} \)
\( \text{Å}( \text{lab(d)} = \{ 5 \})

For everything else (steps, filters)

\( \text{Å}( \text{child(} \text{Å}( \text{descendant(} \text{root} \text{), lab(a)}) = \{ 2, 6, 8 \}), \text{lab(b)}) = \{ 3, 7, 9 \} \)
\( \text{Å}( \text{lab(c)}, \text{lab(d)} = \{ 5 \})

For everything else (steps, filters)

\( \text{Å}( \text{lab(a)} = \{ 2, 6, 8 \} \)
\( \text{Å}( \text{lab(b)} = \{ 3, 7, 9 \} \)
\( \text{Å}( \text{lab(c)} = \{ 1, 4 \} \)
\( \text{Å}( \text{lab(d)} = \{ 5 \})

Axis evaluation: \( O( |N| ) = O( |D| ) \) for node tests

Node Set operation: \( O( |D| ) \) for everything else (steps, filters)

Core XPath query Q can be evaluated in time \( O( |Q| \cdot |D| ) \)
nodes x such that
\( \exists y: \text{child}(x,y) \) and \( y \in \text{lab}(c) \) and
\( \exists z: \text{child}(y,z) \) and \( z \in \text{lab}(d) \)

\( \text{lab}(a) = \{ 2, 6, 8 \} \)
\( \text{lab}(b) = \{ 3, 7, 9 \} \)
\( \text{lab}(c) = \{ 1, 4 \} \)
\( \text{lab}(d) = \{ 5 \} \)

\( \text{dom} = \{ 1, \ldots, 9 \} \)

Bottom-Up (parse “right-branching”)

Candidates for \( z \): \( \text{lab}(d) \)
For \( y \): \( \text{parent}(\text{lab}(d)) \) and labeled \( c \)
\( = \bigcup ( \text{lab}(c), \text{parent}(\text{lab}(d)) ) \)

For \( x \): parent( … )

\( \text{dom} \)

\( \text{root} \)

becomes union!
IE6 (native code, Windows)

Core XPath query (below, size 3. Size in experiment: 20)
a/b/ancestor::a/b/ancestor::a/b/ancestor::a/b]]

IE6 (native code, Windows)

Quadratic-time evaluation

Compare this to the top-down algorithm for simple queries / a/b/c

BU Core XPath Algorithm:
(1) Translate query (parse tree) into the node-set-operations tree below.
(2) Evaluate node-set-ops tree.

Question

Can you extend the top-down look-up algorithm from simple queries (/a/b/c) to all Core XPath queries?

How big are look-up tables (if you want to have one look-up per node)?

Much faster than node-set based algorithm?

4. Polynomial Time Evaluation of Full XPath

All following slides are taken from Georg Gottlob and Christoph Koch “XPath Query Processing”. Invited tutorial at DBPL 2003
http://www.dbai.tuwien.ac.at/research/xmlaskforce(xpath-tutorial1.ppt.gz)
XPath expressions are evaluated with respect to a context $\langle x, k, n \rangle$ consisting of a node position $k$ and a size $n$. These values specify a current "situation" in which a query or subquery should be evaluated. Determined by preceding XSL or XPath computations.

A previously computed node-set $\{n_1, n_3, n_5, n_9\}$.

Continuation of computation $\langle n_3, 3, 3 \rangle$:

This is the context information used for the further query evaluation starting at $n_3$.

Example of an XPath query not in Core XPath

Sample document $D$:

$$\langle a \rangle \langle b/\rangle \langle c/\rangle \langle b/\rangle \langle c/\rangle \langle/\rangle$$

Sample query $Q$:

child::b/following::*[position() > 2]

Example: Formal Semantics of XPath

Relational Operators

First formal semantics of a relevant fragment of XPath: Phil Wadler 1999

Example: Formal Semantics of XPath

Relational Operators

Std. Semantics of Location Paths

First formal semantics of a relevant fragment of XPath: Phil Wadler 1999
Context-value Tables (CVT)

- Four types of values (nset, num, str, bool)
- Defined for each XPath expression e
- The CVT of e is a relation $R \subseteq C \times (\text{nset} \cup \text{num} \cup \text{str} \cup \text{bool})$

Parse Tree of the Query

Query:
```
child::b/following::*[position() != last() and self::b]
```

Query Tree:
```
N_1: child::b/N_2
N_2: following::*/N_3
N_3: position() N_4: last() N_5: self::b
```

(In fact, this is only a relevant subset of the full tables.)
Context-Value Table Principle

if CVT for each operation $\mathcal{O}(e_1, \ldots, e_n)$ can be computed in polynomial time given the CVTs for sub-expressions $e_1, \ldots, e_n$

then CVT of overall query can be computed (bottom-up) in polynomial time.

Efficiency of the PTIME Algorithm

- Time Complexity $O(D^2 \cdot |Q|^2)$
- Space Complexity $O(D^4 \cdot |Q|^2)$
- In practice, most queries run in quadratic time
- This is for main-memory implementations.
- Adaptation to secondary storage algorithms with PTIME complexity is easy (but with worse bounds than the ones given above).

Alternative Context Representation

- Contexts represented as ("previous context node", "current context node") rather than ("context node", "position", "size").
- Need to recompute "position" and "size" on demand.
- Complexity lowered to time $O(|data|^4 \cdot |query|^2)$, space $O(|data|^3 \cdot |query|^2)$.

Context Simplification Technique

1. Only materialize relevant context.
2. Core XPath evaluation algorithm for outermost and innermost paths /child//..[..]e[..] to \[/child//..].
3. Treating "position" and "size" in a loop.
   - Because of tree shape of query, loops never have to be nested.

Linear Space Fragment

- "Wadler Fragment" [Wadler, 1999]: Core XPath + position(), last(), and arithmetics.
- Evaluation in quadratic time and linear space.

```latex
\text{For } x \in [i..a] \text{ compute contexts } (y, p, n) \text{ in } x \cdot [b].
\text{Compute } Y = \{ y \mid (y, p, n) \in x \cdot [b] \text{ and } p \geq 2n \}.
\text{Similarly, compute } Z = \{ z \mid z \cdot \text{position}(z) = \text{last}(z) \}.
\text{Compute } X = \{ (x, z) \mid z \subseteq Z \text{ and } \text{child}(x, z) \} \text{ - in linear time.}
\text{Result is } \{ (w, v) \mid v \in X \cap Y, w \in v \cdot \text{descendant}(w) \}.
```
Summary

Full XPath
- Bottom-up algorithm based on CVT
  - Time $O(|data|^2 \cdot |query|^2)$, space $O(|data|^2 \cdot |query|^2)$.
- Top-down evaluation
  - Time $O(|data|^2 \cdot |query|^2)$, space $O(|data|^2 \cdot |query|^2)$.
- Context-reduction technique
  - Time $O(|data|^2 \cdot |query|^2)$, space $O(|data|^2 \cdot |query|^2)$.

Wadler fragment
- Time $O(|data|^2 \cdot |query|^2)$, space $O(|data|^2 \cdot |query|^2)$.

Core XPath
- Time and space $O(|data| \cdot |query|)$. 