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
dfs
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

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

//a/b =Q

query match position: p = 1

[startElement a]

Æ partial match. If element name was different from "a", then p would remain equal to 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]

Æ push(p)

Æ p = 1

Push current match position p for every startElement (except for the root node)
1. Top-Down Evaluation of Simple Paths

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

```
query match position: p = 2
```

```
//a/b = Q

[startElement a]
[startElement b]

p = pop()
```

1. Top-Down Evaluation of Simple Paths

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

```
query match position: p = 2
```

```
//a/b = Q

[startElement a]
[startElement b]

p = pop()
```

Question: Why is p set to 1? What if query was //a/a?
1. Top-Down Evaluation of Simple Paths

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

//a/b → \( Q \)

query match position: \( p = 1 \)

\[
\text{[startElement( a )]}
\text{[startElement( b )]}
\text{[startElement( a )]}
\text{[endElement( a )]}
\]

Thus, push(0) and \( p = p + 1 = 2 \)

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

\[
\text{[startElement( a )]}
\text{[startElement( b )]}
\text{[startElement( a )]}
\text{[endElement( a )]}
\text{[startElement( a )]}
\text{[startElement( a )]}
\]

\( Q[1] \)

\( p = 2 \), thus, current node is a match!

→ Mark it as match/result

→ push(0)

æ Mark it as match/result

æ push(0)

æ Mark it as match/result

æ push(0)

æ Mark it as match/result

æ push(0)

æ Mark it as match/result

æ push(0)
1. Top-Down Evaluation of Simple Paths

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

*a* / *a* / *b*   = *Q*

query match position: *p* = 2

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

*p* = pop() = 2

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

Linear time: \( O(\#\text{Nodes}) \) = \( O(|D|) \)

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 ☹

SAX-based path query evaluation (sketch):

- Preparation:
  - Represent path query \( /a/b\), \( /a/b/c\), etc.
  - via the step array
    \[ p[0] = \text{root}, \quad p[1] = \text{parent}(0), \quad \vdots, \quad p[n] = \text{end} \]
  - Maintain an array \( r[i] \) of \( i = 0, \ldots, n \), the current step in the path.
  - Maintain a stack \( S \) of index positions.
- [startDocument]
  - Empty stack \( S \). We start with the first step.
  - [startElement]
    - If the current step's tag name \( v \) and the reported tag name match, proceed to next step.
    - Otherwise make a failure transition \( \varepsilon \). Remember how far we have come already: \( \text{push the current step i onto S} \).
  - [endElement]
    - The parser ascended to a parent element. Resume path traversal from where we left earlier: \( \text{pop old i from S} \).
  - [character]
    - If the current step path \( i \) = \( \text{test} \) we have found a match. Otherwise do nothing.

This failure function \( \varepsilon \) is to be explained in the tutorial.
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?

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.: \texttt{//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 (≤, ≥, <, >, etc.)
- String operations (contains, starts-with, translate, string-length, etc.)

Thus in Core XPath

- Select nodes only depending on labels.
  No counting. No values.
Axis Evaluation

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

Forward Axes:
- self
- child
- descendant
- follow

Backward Axes:
- parent
- ancestor
- preceding
- follow

Axis Evaluation
Maps a Node Set to a Node Set

Node Set represented as bit-field
12345678901011
00011111000000

In doc order

In doc order

Can be done for ANY axes in linear time wrt number of nodes
If we have constant time look-up for:
- first-child(node), parent(node)
- next-sibling(node), previous-sibling(node)

For linear time forward axis evaluation:
- enough to have first-child/next-sibling.

Axis Evaluation
Maps a Node Set to a Node Set

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

Node Set represented as bit-field
12345678901011
00011111000000

Naïve: Node-Set = \{ 3, 4, 5, 7 \}

axis( Node-Set ) = axis(3) \cup axis(4) \cup axis(5) \cup axis(7)

O( #Nodes #Nodes ) = quadratic time \( \Theta \)

Axis Evaluation
Maps a Node Set to a Node Set

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

Node Set represented as bit-field
12345678901011
00011111000000

Naïve: Node-Set = \{ 3, 4, 5, 7 \}

axis( Node-Set ) = axis(3) \cup axis(4) \cup axis(5) \cup axis(7)

O( #Nodes #Nodes ) = quadratic time \( \Theta \)

Axis Evaluation
Maps a Node Set to a Node Set

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

Node Set represented as bit-field
12345678901011
00011111000000

Naïve: Node-Set = \{ 3, 4, 5, 7 \}

axis( Node-Set ) = axis(3) \cup axis(4) \cup axis(5) \cup axis(7)

O( #Nodes #Nodes ) = quadratic time \( \Theta \)

Axis Evaluation
Maps a Node Set to a Node Set

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

Node Set represented as bit-field
12345678901011
00011111000000

Naïve: Node-Set = \{ 3, 4, 5, 7 \}

axis( Node-Set ) = axis(3) \cup axis(4) \cup axis(5) \cup axis(7)

O( #Nodes #Nodes ) = quadratic time \( \Theta \)

Axis Evaluation
Maps a Node Set to a Node Set

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

Node Set represented as bit-field
12345678901011
00011111000000

Naïve: Node-Set = \{ 3, 4, 5, 7 \}

axis( Node-Set ) = axis(3) \cup axis(4) \cup axis(5) \cup axis(7)

O( #Nodes #Nodes ) = quadratic time \( \Theta \)

Axis Evaluation
Maps a Node Set to a Node Set

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

Node Set represented as bit-field
12345678901011
00011111000000

Naïve: Node-Set = \{ 3, 4, 5, 7 \}

axis( Node-Set ) = axis(3) \cup axis(4) \cup axis(5) \cup axis(7)

O( #Nodes #Nodes ) = quadratic time \( \Theta \)

Axis Evaluation
Maps a Node Set to a Node Set

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

Node Set represented as bit-field
12345678901011
00011111000000

Naïve: Node-Set = \{ 3, 4, 5, 7 \}

axis( Node-Set ) = axis(3) \cup axis(4) \cup axis(5) \cup axis(7)

O( #Nodes #Nodes ) = quadratic time \( \Theta \)

Axis Evaluation
Maps a Node Set to a Node Set

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

Node Set represented as bit-field
12345678901011
00011111000000

Naïve: Node-Set = \{ 3, 4, 5, 7 \}

axis( Node-Set ) = axis(3) \cup axis(4) \cup axis(5) \cup axis(7)

O( #Nodes #Nodes ) = quadratic time \( \Theta \)

Axis Evaluation
Maps a Node Set to a Node Set

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

Node Set represented as bit-field
12345678901011
00011111000000

Naïve: Node-Set = \{ 3, 4, 5, 7 \}

axis( Node-Set ) = axis(3) \cup axis(4) \cup axis(5) \cup axis(7)

O( #Nodes #Nodes ) = quadratic time \( \Theta \)

Axis Evaluation
Maps a Node Set to a Node Set

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

Node Set represented as bit-field
12345678901011
00011111000000

Naïve: Node-Set = \{ 3, 4, 5, 7 \}

axis( Node-Set ) = axis(3) \cup axis(4) \cup axis(5) \cup axis(7)

O( #Nodes #Nodes ) = quadratic time \( \Theta \)

Axis Evaluation
Maps a Node Set to a Node Set

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

Node Set represented as bit-field
12345678901011
00011111000000

Naïve: Node-Set = \{ 3, 4, 5, 7 \}

axis( Node-Set ) = axis(3) \cup axis(4) \cup axis(5) \cup axis(7)

O( #Nodes #Nodes ) = quadratic time \( \Theta \)

Axis Evaluation
Maps a Node Set to a Node Set

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

Node Set represented as bit-field
12345678901011
00011111000000

Naïve: Node-Set = \{ 3, 4, 5, 7 \}

axis( Node-Set ) = axis(3) \cup axis(4) \cup axis(5) \cup axis(7)

O( #Nodes #Nodes ) = quadratic time \( \Theta \)

Axis Evaluation
Maps a Node Set to a Node Set
Axis Evaluation

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

Node Set:
1234567891011
000000000010 0

After one parent look-up.

Result Node Set:
1234567891011
000000000010 0

In result set already.

Axis Evaluation

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

Node Set:
1234567891011
000000000010 0

After 1 parent look-up.

Result Node Set:
1234567891011
000000000010 0

In result set already.

Already in result set.

Axis Evaluation

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

Node Set:
1234567891011
000000000010 0

After 2 parent look-ups.

Result Node Set:
1234567891011
000000000010 0

In result set already.

Axis Evaluation

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

Node Set:
1234567891011
000000000010 0

After 3 parent look-ups.

Result Node Set:
1234567891011
000000000010 0

In result set already.

Axis Evaluation

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

Node Set:
1234567891011
000000000010 0

After 4 parent look-ups.

Result Node Set:
1234567891011
000000000010 0

In result set already.

Axis Evaluation

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

Node Set:
1234567891011
000000000010 0

After 5 parent look-ups.

Result Node Set:
1234567891011
000000000010 0

In result set already.

Axis Evaluation

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

Node Set:
1234567891011
000000000010 0

After 6 parent look-ups.

Result Node Set:
1234567891011
000000000010 0

In result set already.

Finished!
Axis Evaluation

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

**Similarly:**
For all other axes!
Forward-axis 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 >once!
  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|11
1|1|0|0|0|1|0|0|0|0|0
After 6 parent look-ups.
+ 5 result look-ups.

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

**Similarly:**
For all other axes!

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

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

**Example**

descendant( node ) =
(first-child | next-sibling)* (first-child( node ))

**Example**

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;
  }
}

This example comes 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
descendant( node ) = (first-child | next-sibling)* (first-child( node ))

Example

descendant( { 1 } ) = (fc | ns)*(first-child( { 1 } )) =

Result Node Set

0|1|1|1|1|1|1|1|1|1

Example

descendant( { 1 } ) = (fc | ns)*(first-child( { 1 } )) =

Result Node Set

0|1|1|1|1|1|1|1|1|1

Example

descendant( { 1 } ) = (fc | ns)*(first-child( { 1 } )) =

Result Node Set

0|1|1|1|1|1|1|1|1|1

Example

descendant( { 1 } ) = (fc | ns)*(first-child( { 1 } )) =

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

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!!

- \( \text{axis} (\text{Set}_1) = \text{Set}_2 \)
- \( \cup (\text{Set}_1, \text{Set}_2) = \text{Set}_3 \) union of \text{Set}_1 and \text{Set}_2
- \( \cap (\text{Set}_1, \text{Set}_2) = \text{Set}_3 \) intersection of \text{Set}_1 and \text{Set}_2
- \( \neg (\text{Set}_1, \text{Set}_2) = \text{Set}_3 \) everything in \text{Set}_1 but not in \text{Set}_2
- \( \text{lab}(a) = \text{Set}_1 \) all nodes labeled by \( a \)

For everything else (steps, filters)

- \( \text{axis}_c(\text{Set}) = \text{Set}_2 \)
- \( \cup (\text{Set}_1, \text{Set}_2) = \text{Set}_3 \) union of \text{Set}_1 and \text{Set}_2
- \( \cap (\text{Set}_1, \text{Set}_2) = \text{Set}_3 \) intersection of \text{Set}_1 and \text{Set}_2
- \( \neg (\text{Set}_1, \text{Set}_2) = \text{Set}_3 \) everything in \text{Set}_1 but not in \text{Set}_2
- \( \text{lab}_c(a) = \text{Set}_1 \) all nodes labeled by \( a \)

for node tests

for everything else (steps, filters)

Size of the Document

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

Axis evaluation: \( \Theta (\#\text{Nodes}) = \Theta (|\text{D}|) \)

Node Set operation: \( \Theta (|\text{Q}| |\text{D}|) \) linear time!
Document
lab(a) = { 2, 6, 8 }
lab(b) = { 3, 7, 9 }
lab(c) = { 1, 4 }
lab(d) = { 5 }

nodes x such that ∃ y: child(x,y) and y ∈ lab(c) and ∃ z: child(y,z) and z ∈ lab(d)
= { 3, 7, 9 }

Bottom-Up (parse "right-branching")
Candidates for z: lab(d)
For y: parent(lab(d)) and labeled c
= Å ( lab(c), parent(lab(d)) )
For x: parent( ... )
**4. Polynomial Time Evaluation of Full XPath**

All following slides are taken from [[source]](http://www.dbai.tuwien.ac.at/research/xmlaskforce/xpath-tutorial1.ppt.gz)

Invited tutorial at DBPL 2003

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?
XPath expressions are evaluated w.r.t. context: \( (x, k, n) \)

- node: position
- size

These values specify a current "situation" in which a query or subquery should be evaluated.

Determined by preceding XSL or XPath computations.

Previously computed node-set:

\[ (n_1, n_3, n_5) \]

Continuation of computation:

\[ (n_5, 3, 4) \]

This is the context information used for the further query evaluation starting at \( n_5 \).

\[ \langle a \rangle \ <b/> \ <c/> \ <b/> \ <c/> \ </a> \]

Sample document \( D \):

Sample query \( Q \):

\[ \text{child::b/following::*[position() > 2]} \]

Example of an XPath query not in Core XPath

Sample document \( D \):

Sample query \( Q \):

\[ \text{child::b/following::*[position() > 2]} \]

Continuation of computation:

\[ (n_3, 3, 3) \]

\[ (n_5, 1, 3) \]

Example: Formal Semantics of XPath

Relational Operators

\[ \begin{align*}
\text{std 
}
\text{num @} \text{str @} \text{bool @} \text{node @} \text{set @} \text{path @} \text{index} \\
\end{align*} \]

Example: Formal Semantics of XPath

Relational Operators

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

\[ \begin{align*}
\text{std 
}
\text{num @} \text{str @} \text{bool @} \text{node @} \text{set @} \text{path @} \text{index} \\
\end{align*} \]
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]

Tree:
- $N_1$: child::b
- $N_2$: following::*
- $N_3$: boolean($N_8$)
- $N_5$: position()
- $N_7$: last()
- $N_9$: self::b

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

if CVT for each operation $O(p(\epsilon_1, ..., \epsilon_n))$ can be computed in polynomial time given the CVTs for sub-expressions $\epsilon_1, ..., \epsilon_n$

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

Efficiency of the PTIME Algorithm

- Time Complexity $O(|D|^4 \times |Q|^2)$
- Space Complexity $O(|D|^4 \times |Q|^2)$
- 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 $O(|data|^4 \times |query|^2)$, space $O(|data|^3 \times |query|^2)$.

Context Simplification Technique

1. Only materialize relevant context.
2. Core XPath evaluation algorithm for outermost and innermost paths $(a/b)(d//e)/f$.
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.

Time and Space Bounds

Bottom-up evaluation based on CVT:
- Time $O(|data|^4 \times |query|^2)$, Space $O(|data|^4 \times |query|^2)$.

Space bound (n ... number of nodes in input document):
- Contexts are at most triples: at most $n^3$ contexts.
- Sizes of values:
  - Node sets: at most $O(n)$
  - Strings, numbers: at most $O(|data|^* |query|)$ – (iterated concatenation of strings, multiplication of numbers).
- Each CVT is of size $O(|data|^* |query|)$.

Need to compute a CVT for each query node and each input node $\rightarrow (|data|^* |query|) = O(n^2)$ – Relational operation "*" on node sets (e.g. $a/b//c//d//e//f//g = h/i//j$).
Summary

Full XPath
- Bottom-up algorithm based on CVT
  - Time $O(|data|^5 \cdot |query|^2)$, space $O(|data|^4 \cdot |query|^2)$.
- Top-down evaluation
  - Time $O(|data|^4 \cdot |query|^2)$, space $O(|data|^3 \cdot |query|^2)$.
- Context-reduction technique
  - Time $O(|data|^4 \cdot |query|^2)$, 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 $O(|data| \cdot |query|)$.