XML and Databases

Lecture 9
Properties of XPath

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2009

Outline

1. XPath Equivalence
2. No Looking Back: How to Remove Backward Axes
3. Containment Test for XPath Expressions

A Note on Equality Test
in XPath

A Note on Equality Test

Useful Functions (on Node Sets)

XPath 1.0
Equality ("=") is based on string value of a node!

XPath 2.0 has clearer comparison operators!

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

//a[b/d = c/d]
selects a-node!!!

A Note on Equality Test

p1, p2 XPath (1.0) Expressions

(p1 == p2) is true if there exists a node selected by p1
that is identical to a node selected by p2

XPath 2.0
XQuery 1.0

false (on any document)

//a[b/d == c/d] selects what?

//child::node()[1] == child::node()[position=last()]
A Note on Equality Test

XPath 1.0 simulation of (node) equality test (==)

Instead of (p1 == p2) write:
(count(p1 | p2) < count(p1) + count(p2))

1. XPath Equivalence

EBNF for XPaths that we want to consider now:

path ::= path | path | / | path | path | path [ qualifier ] | axes := nodetest | |
qualifier ::= qualifier and qualifier | qualifier or qualifier | ( qualifier )
path ::= path | path == path | path .
axes ::= reverse axes | forward axes.
reverse axes ::= parent | ancestor | ancestor-or-self |
| preceding | preceding-sibling |
| following | following-sibling |
node test ::= namespace | | text() | node() |
An XPath starting with "/" (root node) is called absolute, otherwise it is called relative.

1. XPath Equivalence

Examples

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>/a//b</td>
<td>/a//b</td>
</tr>
<tr>
<td>[/a</td>
<td>c]</td>
</tr>
</tbody>
</table>

NOT equivalent: child::parent::self::c
- show a counter example!

1. XPath Equivalence

p1, p2 XPaths
q arbitrary qualifier
Rel→Abs If p1 ≡ p2, then /p1 ≡ /p2.
Adjunct If p1 ≡ p2 and p is a relative, then p1 | p ≡ p2 | p.
If p1 ≡ p2 and p1 | p2 relative, then p1 | p | p2 ≡ p2 | p1. p2.
Qualifier Flattening p[p1 | p2] ≡ p[p1 | p2]
ancestor-or-self | descendant-or-self | descendant-or-self | descendant-or-self |
| p | p1 | self::n | self::n |
| p | p2 | self::n | self::n |
| p1 | p2 | p1 | self::node() | p2 | self::node() |
| p2 | p1 | p2 | self::node() | p2 | self::node() |

2. No Looking Back

Dual backward forward

parent ancestor ancestor-or-self preceding preceding-sibling
child descendant descendant-or-self following following-sibling
Thus: dual(parent) = child dual(following) = preceding etc.

Rewrite rule #1 (p unsustainable, ax reverse axis)

<table>
<thead>
<tr>
<th>ax</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>descendant</td>
<td>m</td>
</tr>
</tbody>
</table>

NOT equivalent: child::parent::self::c
- show a counter example!
Rewrite rule #1 (p,s: relative paths, ax: reverse axis)

\[ p[ax::m/s] \rightarrow p[[\text{descendant}::m/s]]/\text{dual}(ax)::\text{node()} == self::\text{node()} \]

any "m-node" in the tree

but, via dual axis, must reach context node

E.g. \( ax = \text{ancestor} \)

\[ p[\text{ancestor}::m] \rightarrow p[[\text{descendant}::m/\text{node()} == self::\text{node()} \]

"any m-node from which the context node can be reached via descendant, must be an ancestor of the context node."

Rewrite rule #1 (p,s: relative paths, ax: reverse axis)

\[ p[ax::m/s] \rightarrow p[[\text{descendant}::m/s]]/\text{dual}(ax)::\text{node()} == self::\text{node()} \]

any "m-node" in the tree

but, via dual axis, must reach context node

E.g. \( ax = \text{preceding-sibling} \)

\[ p[\text{preceding-sibling}::m] \rightarrow p[[\text{descendant}::m/\text{node()} == self::\text{node()} \]

"any m-node from which the context node can be reached via following-sibling, must be a preceding-sibling of the context node."

Rewrite rule #1 (p,s: relative paths, ax: reverse axis)

\[ p[ax::m/s] \rightarrow p[[\text{descendant}::m/s]]/\text{dual}(ax)::\text{node()} == self::\text{node()} \]

any "m-node" in the tree

but, via dual axis, must reach context node

E.g. \( ax = \text{parent} \)

\[ p[\text{parent}::m] \rightarrow p[[\text{descendant}::m/\text{node()} == self::\text{node()} \]

"any m-node from which the context node can be reached via ancestor, must be a parent of the context node."

Similar rules for absolute paths:

\[ /\text{ancestor}::ax::m \rightarrow /\text{descendant}::m/\text{node()} == /\text{ancestor}::ax::m \]
\[ /\text{ancestor}::ax::m \rightarrow /\text{descendant}::m/\text{node()} == /\text{ancestor}::ax::m \]

Rewrite rules #2 and #2a

Rewrite rules #2 and #2a

Removes first reverse axis inside a filter (qualifier).
Use qualifier flattening to replace "any" reverse axis from inside a filter. Qualifier Flattening: \( p[p1[p2]] = p[p1[p2]] \)

Rewrite rules #2 and #2a

Rewrite rules #2 and #2a

E.g.

\[ /\text{descendant}::\text{price}/\text{preceding}::\text{name} \]

is rewritten via Rule #2a into:

\[ /\text{descendant}::\text{name}/\text{following}::\text{price}==/\text{descendant}::\text{price} \]
Similar rules for absolute paths:

Rewrite rules #2 and #2a

E.g.

\[ /\text{descendant::price}/\text{preceding::name} \]

is rewritten via Rule #2a into:

\[ /\text{descendant::name}[\text{following::price}==/\text{descendant::price}] \]

Of course, the “join” can be removed in this example:

\[ /\text{descendant::name}[\text{following::price}] \]

Not needed, in this example.

Similar rules for absolute paths:

Rewrite rules #2 and #2a

E.g.

\[ /\text{descendant::journal}[\text{child::title}]/\text{descendant::price}/\text{preceding::name} \]

becomes

\[ /\text{descendant::name}[\text{following::price}==/\text{descendant::journal}[\text{child::title}]/\text{descendant::price}] \]

Can you avoid the join, also for this example?

Similar rules for absolute paths:

Rewrite rules #2 and #2a

E.g.

\[ /\text{parent::m}/\text{ax}::n/\text{m} \]

\[ /\text{descendant::m}[\text{dual(ax)}::n == /\text{parent::m}/\text{ax}::n] \]

\[ /\text{parent::m}/\text{ax}::n/\text{m} \]

\[ /\text{descendant::m}[\text{dual(ax)}::n == /\text{parent::m}/\text{ax}::n] \]

Can you avoid the “join” as well for this example?

Rules (1),(2),(2a) suffice to remove ALL backward axes from above queries!

Why?

\[ \rightarrow \text{Size increase?} \]

\[ \rightarrow \text{How many joins?} \]

2. No Looking Back

Interaction of back=parent with forward axes:

\[ /\text{parent::m}/\text{ax}::n/\text{m} \]

\[ /\text{descendant::m}[\text{dual(ax)}::n == /\text{parent::m}/\text{ax}::n] \]

(3)

2. No Looking Back

Interaction of back=parent with forward axes:

\[ /\text{parent::m}/\text{ax}::n/\text{m} \]

\[ /\text{descendant::m}[\text{dual(ax)}::n == /\text{parent::m}/\text{ax}::n] \]

(3)

\[ \text{child::m}/\text{parent::m}/\text{m} \]

\[ /\text{child::m}[\text{dual(ax)}::n == /\text{child::m} \]

(4)
2. No Looking Back

Interaction of \texttt{back=parent} with forward axes:

\begin{align*}
\text{descendant.m/parent.m} &= \text{ descendant-or-self::m[child::3]} \\
\text{child.n/parent.m} &= \text{ self::n[child::n]} \\
\text{p/self.n/parent.m} &= \text{ p/self::n/parent::m} \\
\text{p/following-sibling::n/parent.m} &= \text{ p/following-sibling::n/parent::m} \\
\text{p/following::n/parent.m} &= \text{ p/following::n/child::n} \\
\text{p/following-sibling::n/parent.m} &= \text{ p/following-sibling::n/parent::m} \\
\text{p/following::n/parent.m} &= \text{ p/following::n/child::n} \\
\end{align*}
2. No Looking Back

Interaction of `back=ancestor` with forward axes:

\[
\begin{align*}
\text{descendant} & : \text{ancestor} \equiv \text{descendant} \land \text{ancestor} \not\equiv \text{self} \not\equiv \text{descendant} \lor \text{ancestor} \not\equiv \text{self} \not\equiv \text{descendant} \\
\text{descendant} \lor \text{ancestor} & \equiv \text{descendant} \lor \text{ancestor} \not\equiv \text{descendant} \lor \text{ancestor} \not\equiv \text{self} \not\equiv \text{descendant} \\
\text{child} & \equiv \text{ancestor} \lor \text{ancestor} \not\equiv \text{self} \not\equiv \text{descendant} \\
\text{self} & \equiv \text{ancestor} \lor \text{ancestor} \not\equiv \text{self} \not\equiv \text{descendant} \\
\text{following-sibling} & \equiv \text{ancestor} \lor \text{ancestor} \not\equiv \text{self} \not\equiv \text{descendant} \\
\text{ancestor} & \equiv \text{ancestor} \lor \text{ancestor} \not\equiv \text{self} \not\equiv \text{descendant} \\
\end{align*}
\]

Similar rules for `ancestor` in filters.

Rule 33
Wrong, I think! Should be descendant instead!

2. No Looking Back

/\descendant::price/preceding::name is rewritten via Rule #2a into:
/\descendant::name[following::price](\descendant::price)

Now, let us use Rule (33a)
/\descendant::m/preceding::n \rightarrow /\descendant::n[following::m]

We obtain
/\descendant::name[following::price]
Theorem
( from D. Olteanu, H. Meuss, T. Furche, F. Bry
XPath: Looking Forward. EDBT Workshops 2002: 109-127 )
Given an XPath expression $p$ that has no joins of the form $(p_1 == p_2)$ with both $p_1, p_2$ relative, an equivalent expression $u$ without reverse axes can be computed.

Time needed: at most exponential in length of $p$
Length of $u$: at most exponential in length of $p$

(moreover: no joins are introduced when computing $u$)

Questions
→ Can you find a subclass for which Time to compute $u$ is linear or polynomial?
→ What is the problem with joins $(p_1 == p_2)$ for removal of reverse axes?

Applications
→ Decrease online-time of publish/subscribe systems based on XPath
→ Decrease query-time by making use of materialized intermediate results
→ Optimization by ruling out queries with empty result set
etc, etc.

3. XPath Containment Test

Given two XPath expressions $p, q$

$0$-containment For every tree, if $p$ selects a node then so does $q$.
$p \subseteq_0 q$

$1$-containment For every tree, all nodes selected by $p$ are also selected by $q$.
$p \subseteq_1 q$

$2$-containment For every tree, and every context node $N$, all nodes selected by $p$ starting from $N$, are also selected by $q$ starting from $N$.
$p \subseteq_2 q$

1. Inclusion on Booleans
2. Inclusion on Node Sets
3. Inclusion on Node Relations

(If only child and descendant axes are allowed then $\subseteq_1$ and $\subseteq_2$ are the same! — Why?)

Question
Given $p, q$ and the fact $p \subseteq_1 q$, how can you determine from a result set of nodes for $q$, the correct result set of nodes for $p$?

3. XPath Containment Test

Sometimes we want to test containment wrt a given DTD:

$$p = /a/b/d$$
$$q = /a/j/c$$

Want to check if $p \subseteq_0 q$.

NOT
a
b
l
d

But, what if documents are valid wrt to this DTD?

root $\rightarrow a^*$
$\rightarrow b^* | c^*$
$b \rightarrow d+c+$
c $\rightarrow d|e$?
Pattern trees

E.g. \( p = a[a\ldots d]/*//c \)

Note: child order has no meaning in pattern trees!

Test \( \subseteq \) (node set inclusion) using \( \subseteq_0 \) (Boolean inclusion)
- Simply add a new node below the selection node
- New tree is Boolean (no selection node)
- In a given XML tree, pattern matches / does not match.

3. XPath Containment Test

Canonical Model - XPath(//,//,/,*)

Idea: if there exists a tree that matches \( p \) but not \( q \), then such a tree exists of size polynomial in the size of \( p \) and \( q \).

Simple: remember, if you know that the XML document is only of height 5, then \( /a/b/*/c \) could be enumerated by \( /a/b/*/c \mid */a/b/*/c \mid *//*/a/*/c \mid *//*//*/a \ldots \)

Similarly, we try to construct a counter example tree, by replacing in \( p \)

\( \forall \) every \( * \) by some new symbol \( "z" \)
\( \forall \) every \( // \) by \( z//, z/z/, z/z/z/, \ldots \)

Formally, must test 1 and 2 more \( z \)'s at right branch of each of the trees.

Homomorphism \( h \) maps each node of \( q \)'s query tree \( Q \) to a node of \( p \)'s query tree \( P \) such that

(1) root of \( Q \) is mapped to root of \( P \)
(2) if \( (u,v) \) is child-edge of \( Q \) then \( (h(u),h(v)) \) is child-edge of \( P \)
(3) if \( (u,v) \) is descendant-edge of \( Q \), then \( h(v) \) is a "below" \( h(u) \) in \( P \)
(4) if \( u \) is labeled by "e" (not ",") then \( h(u) \) is also labeled by "e".

Example

\( p \)'s pattern tree
\( q \)'s pattern tree

Test for \( q \)-match:

Formally, must test 1 and 2 more \( z \)'s at right branch of each of the trees.

Theorem

\( p \subseteq q \) if and only if there is a homomorphism from \( Q \) to \( P \).
3. XPath Containment Test

Homomorphism $h$ maps each node of $q$'s query tree $Q$ to a node of $p$'s query tree $P$ such that

(1) root of $Q$ is mapped to root of $P$
(2) if $(u,v)$ is child-edge of $Q$ then $(h(u),h(v))$ is child-edge of $P$
(3) if $(u,v)$ is descendant-edge of $Q$, then $h(v)$ is a "below" $h(u)$ in $P$
(4) if $u$ is labeled by "e" (not *), then $h(u)$ is also labeled by "e".

$\Rightarrow$ hom. $h$ exists from $Q$ to $P$, thus $p \subseteq_0 q$ must hold!

Cave: If we add the star (*) then homomorphism need not exist!

$\Rightarrow$ there are $p,q \in \text{XPath}(//,//,[],*)$ such that $p \subseteq_0 q$ and there is no homomorphism from $Q$ to $P$. 

$p = /a//b[c/*//d]/*/c$ 
$q = /a//b[c/*//d]/*//c$ 
Where to map??
If we add the star (\(\ast\)) then homomorphism need not exist!

\[ \exists \ p \neq q \in \text{XPath}(/, //, [], *, |) \text{ such that } p \subseteq q \text{ and there is no homomorphism from Q to P} \]

Where to map??

Let's check the web...  \(\Rightarrow\)  YES  \(p\) contained in \(q\)!

Emptiness test for automata

Proof Idea  construct automaton for all possible counter example trees.  Test if this automaton accepts any tree.

3. XPath Containment Test

Automaton Technique

Recall: for any DTD there is a tree automaton which recognized the corresponding trees.

Similarly, for any \(\text{XPath}(/, //, [], *, |)\) expression \(\text{ex}\) we can construct a (non-deterministic bottom-up) tree automaton \(A\) which accepts a tree if and only if \(\text{ex}\) matches the tree.

Theorem

Containment test of \(\text{XPath}(/, //, [], *, |)\) in the presence of DTDs can be solved in \(\text{EXPTIME}\).

Intersection of automata ("product construction")

Proof Idea  construct automaton for all possible counter example trees.  Test if this automaton accepts any tree.

3. XPath Containment Test

Chase Technique    -- 1979 relational DB's to check query containment in the presence of integrity constraints.

Example

\[
\begin{align*}
\text{root} & \rightarrow a^* \\
\text{a} & \rightarrow b^+ | c^+ \\
\text{b} & \rightarrow d+c \\
\text{c} & \rightarrow b|c \\
\end{align*}
\]

\(p = /a/b/d\)

\(q = /a/c\)

Is \(p\) contained in \(q\) for E-conform documents?

First Possibility: use tree automata

\(\Rightarrow\)  Construct automata \(Ap\), \(Aq\), \(AE\)

\(\Rightarrow\)  Construct \(BCq\) for the complement of \(Aq\) (\(\neg\text{not } q\))

\(\Rightarrow\)  Intersect \(BCp\) with \(Ap\) with \(AE\) (gives automaton \(A\))

\(\Rightarrow\)  Check if \(A\) accepts any tree.
3. XPath Containment Test

Chase Technique -- 1979 relational DB's to check query containment in the presence of integrity constraints.

Example

\[
\text{DTD } E = \begin{align*}
\text{root} & \rightarrow a^* \\
a & \rightarrow b^* \mid c^* \\
b & \rightarrow d+c \\
c & \rightarrow b+c
\end{align*}
\]

Example:

\[
p = /a/b/d \\
q = /a/c
\]

Is \( p \) contained in \( q \) for E-conform documents?

Each b-element has a d-child and a c-child

Constraints:

\[
c1: b \rightarrow d \\
c2: b \rightarrow c
\]

p's pattern tree

\[
\text{p is contained in q in the presence of the DTD } E
\]

Example:

\[
p = /a/b/d \\
q = /a/c
\]

Is \( p \) contained in \( q \) for E-conform documents?

Each b-element has a d-child and a c-child

Constraints:

\[
c1: b \rightarrow d \\
c2: b \rightarrow c
\]

p's pattern tree

\[
p's \text{ pattern tree after chasing with } c1, c2
\]

Example:

\[
p = /a/b/d \\
q = /a/c
\]

Is \( p \) contained in \( q \) for E-conform documents?

Each b-element has a d-child and a c-child

Constraints:

\[
c1: b \rightarrow d \\
c2: b \rightarrow c
\]

p's pattern tree

\[
p's \text{ pattern tree after chasing with } c1, c2
\]