XML and Databases
XPath evaluation using RDBMS

Kim.Nguyen@nicta.com.au

Week 11
**XPath**

To handle XPath expressions correctly:

1) Rewrite your XPath expression in the *concrete syntax*, as per:

   ```
   http://www.w3.org/TR/xpath
   : .  →  self::node()
   // →  /descendant-or-self::node()/
   .../foo →  .../child::foo
   ```
**XPath**

To handle XPath expressions correctly:

1) Rewrite your XPath expression in the *concrete syntax*, as per:

   http://www.w3.org/TR/xpath

   
   `. ` ~> `self::node()`

   `// ` ~> `/descendant-or-self::node()/`

   `.../foo ` ~> `.../child::foo`

2) Use a data-structure for XPath expressions

   \[ p ::= \text{bool} \times [(a_1, l_1, p_1); \ldots; (a_n, l_n, p_n)] \]

   \[ a ::= \text{child} | \text{descendant} | \ldots \]

   \[ l ::= * \text{tagname} | \text{text()} | \text{node()} \]
**XPath**

To handle XPath expressions correctly:

1) Rewrite your XPath expression in the *concrete syntax*, as per:

   - `http://www.w3.org/TR/xpath`

   
   ```
   . \sim \self::node()
   // \sim /\text{descendant-or-self::node()}/
   .../\text{foo} \sim .../\text{child::foo}
   ```

2) Use a data-structure for XPath expressions

   ```
   p ::= bool \times [(a_1, l_1, p_1); \ldots ; (a_n, l_n, p_n)]
   a ::= \text{child}|\text{descendant}|\ldots
   l ::= *|\text{tagname}|\text{text()}|\text{node()}
   ```

A path is a sequence of steps and a boolean, which indicates whether the path is “global” (starts with “/”) or relative (starts without “/”, e.g. `.//a/b`).
XPath

To handle XPath expressions correctly:
1) Rewrite your XPath expression in the *concrete syntax*, as per:

```
http://www.w3.org/TR/xpath
```

```
.  ⟷  self::node()
//  ⟷  /descendant-or-self::node()/
.../foo  ⟷  .../child::foo
```

2) Use a data-structure for XPath expressions

```
p ::=  bool × [(a₁, l₁, p₁); ...; (aₙ, lₙ, pₙ)]
a ::=  child|descendant|...
l ::=  *|tagname|text()|node()
```

A path is a sequence of steps and a boolean, which indicates whether the path is “global” (starts with "/") or relative (starts without "/", e.g. ./a/b).
XPath (example)

The expression:

```xml
//a[./b]//following-sibling::a
```

becomes:

```xml
/descendant-or-self::node()/child::a[self::node()/child::b]/descendant-or-self::node()/following-sibling::a
```

And is represented by:

```xml
true,[
 (descendant-or-self,node(),[]);
 (child,"a",[false,[ (self,node(),[]); (child,"b",[])]]));
 (descendant-or-self,node(),[]);
 (following-sibling,"a",[])
]
```
Compilation of XPath

The general algorithm now is:

1. rewrite the XPath expression;
2. transform it into a sequence of steps;
3. traverse the sequence step by step and build an SQL query

Represent each node of the document by an SQL table containing:
- pre-order, post-order, parent of the node
- its tag in the tag field if the node is an element, NULL otherwise
- its text value if the node is a text node, NULL otherwise

Represent each attribute of the document by an SQL table containing:
- pre-order of the element containing the attribute
- the name of the attribute
- the text value of the attribute

you can use the same table/code as in Assignment 3
Logical encoding of axes

We think of the way to encode the XPath expression. We use propositional formulae:

\[
\begin{align*}
    f & ::= v \mid f \land f \mid f \lor f \mid \neg f \mid P(f, \ldots, f) \\
    v & ::= x \mid y \mid z \mid \ldots \\
    P & ::= pre \mid post \mid parent \mid < \mid > \mid \ldots
\end{align*}
\]

The idea is to write new predicates which represent a particular axis. For instance:

\[
\text{descendant}(x, y) \equiv pre(x) < pre(y) \land post(x) > post(y)
\]

We reads: “node y is a descendant of node x if the pre-order of x is less than the preorder of y and if the post-order of x is larger than the post-order of y”
Logical encoding of axes

Most axes are straightforward. By using formulæ, it is also easy to simplify some formulae by using logical rules:

\[
\begin{align*}
\text{self}(x, y) & \equiv \text{pre}(x) = \text{pre}(y) \\
\text{descendant}(x, y) & \equiv \text{pre}(x) < \text{pre}(y) \land \text{post}(x) > \text{post}(y) \\
\text{descendant-or-self}(x, y) & \equiv \text{pre}(x) \leq \text{pre}(y) \land \text{post}(x) \geq \text{post}(y) \\
\text{child}(x, y) & \equiv \text{descendant}(x, y) \land x = \text{parent}(y) \\
\text{ancestor}(x, y) & \equiv \text{pre}(x) > \text{pre}(y) \land \text{post}(x) < \text{post}(y) \\
\text{preceding}(x, y) & \equiv \text{pre}(x) > \text{pre}(y) \land \text{post}(x) > \text{post}(y) \\
\text{following}(x, y) & \equiv \text{pre}(x) < \text{pre}(y) \land \text{post}(x) < \text{post}(y)
\end{align*}
\]

It is also handy to have a predicate to say “x is the root of the document (the DOCUMENT_NODE)”: \[
\text{root}(x) \equiv \text{pre}(x) = 0
\]
Logical encoding of tests

There are only a few tests. \( T(x) \) is true if the test \( T \) is true for the node \( x \):

\[
\begin{align*}
is\_node(x) & \equiv \text{is always true} \\
is\_text(x) & \equiv \text{is true if } x \text{ is a text node} \\
is\_star(x) & \equiv \text{is true if } x \text{ is an element node}
\end{align*}
\]

We also define the predicate \( tag(x) \) which returns the tag of \( x \) and \( text(x) \) which returns the text of \( x \).

Example: If we are on a context node \( x \) and want to take the step child::a then, we want to select all nodes \( y \) such that:

\[
\text{child}(x, y) \land \text{tag}(y) = "a"
\]

which is equivalent to:

\[
\text{pre}(x) < \text{pre}(y) \land \text{post}(x) > \text{post}(y) \land x = \text{parent}(y) \land \text{tag}(y) = "a"
\]
Example of logical encoding

Consider the path `///b/text()`

1) Rewrite it into the expanded syntax:

   `/child::*/descendant-or-self::node()/child::b/child::text()`

2) Compute the formula step by step:

   \[
   \begin{align*}
   &\text{root}(r_1) \\
   &\quad \land \text{child}(r_1, r_2) \land \text{is\_star}(r_2) \\
   &\quad \land \text{descendant\_or\_self}(r_2, r_3) \\
   &\quad \land \text{child}(r_3, r_4) \land \text{tag}(r_4) = "b" \\
   &\quad \land \text{child}(r_4, r_5) \land \text{is\_text}(r_5)
   \end{align*}
   \]

   Starts at the document root

   The node() test is always true so we don’t put anything
From formulae to SQL

The SQL syntax is close to the one used for the formulae. The previous query: *///b/text(), which is:

/child::*/descendant-or-self::*node()/child::*b/child::*text()
is written in SQL:

```sql
SELECT DISTINCT r5.pre
FROM table r1, table r2, table r3, table r4, table r5
WHERE r1.pre = 0 /* root(r1) */
AND r1.pre < r2.pre AND r1.post > r2.post
  AND r1.pre = r2.parent /* child(r1,r2) */
  AND r2.tag != NULL /* is_star(r2) */
AND r2.pre <= r3.pre AND r2.post >= r3.post
AND r3.pre < r4.pre AND r3.post > r4.post
  AND r3.pre = r4.parent /* child(r3,r4) */
  AND r4.tag = "a"
AND r4.pre < r5.pre AND r4.post > r5.post
  AND r4.pre = r5.parent /* child(r4,r5) */
  AND r5.text != NULL /* is_text(r5) */
ORDER BY r5.pre
```
SELECT DISTINCT r5.pre
FROM table r1, table r2, table r3, table r4, table r5
WHERE r1.pre = 0
;
ORDER BY r5.pre

- SELECT DISTINCT x.pre: returns the set (DISTINCT removes duplicates) of pre-order numbers for the nodes specified by x. x must correspond to the last step of the toplevel query (i.e. not in a filter).
- FROM table r1,...: binds n variable to the element table.
- ORDER BY x.pre ensures that the results are in document order. ORDER BY and SELECT DISTINCT reference the same variable.
following-sibling axis

This axis is a bit trickier. First let’s try to express (logically) the set of siblings $y$ of a node $x$. The siblings of $x$ are the nodes with the same parent as $x$. We would formally write:

\[
\text{ sibling}(x, y) \equiv \exists z, \text{parent}(x, z) \land \text{parent}(y, z)
\]

If we want following or preceding siblings, we just have to add a condition on the pre-order:

\[
\begin{align*}
\text{ preceding-sibling}(x, y) & \equiv \exists z, \text{parent}(x, z) \land \text{parent}(y, z) \land \text{pre}(x) > \text{pre}(y) \\
\text{ following-sibling}(x, y) & \equiv \exists z, \text{parent}(x, z) \land \text{parent}(y, z) \land \text{pre}(x) < \text{pre}(y)
\end{align*}
\]

Thus in SQL, for a step `following-sibling::t` we must introduce 2 variables and not one.
**following-sibling axis**

The query:

```xml
//a/following-sibling::b
```

is rewritten into:

```xml
/descendant-or-self::node()/child::a/following-sibling::b
```

which gives the SQL query:

```sql
SELECT DISTINCT r5.pre
FROM table r1, table r2, table r3, table r4, table r5
WHERE r1.pre = 0
  AND r1.pre <= r2.pre AND r1.post >= r2.post
  AND r2.pre < r3.pre AND r2.post > r3.post
    AND r2.pre = r3.parent AND r3.tag = "a"
  AND r3.pre > r4.pre AND r3.post < r4.post
    AND r3.parent = r4.pre /* parent(r3,r4) */
  r5.pre > r4.pre AND r5.post < r4.post
    AND r5.parent = r4.pre /* parent(r5,r4) */
  AND r3.pre < r5.pre
  AND r5.tag = "b"
ORDER BY r5.pre
```
Filters

Consider: //a[./preceding::b]
Rewrite as:
  /descendant-or-self::node()/child::a[self::node()/preceding::b]
We have two paths:
  /descendant-or-self::node()/child::a (1)
  self::node()/preceding::b (2)

SELECT DISTINCT r3.pre
FROM table r1, table r2, table r3, table r4, table r5
WHERE r1.pre = 0
  AND r1.pre <= r2.pre AND r1.post >= r2.post
  AND r2.pre < r3.pre AND r2.post > r3.post
  AND r2.pre = r3.parent
  AND r3.tag = "a" /* This is exactly like before */
  AND r3.pre = r4.pre /* self::node() */
  AND r4.pre > r5.pre AND r4.post > r5.post /* preceding::b */
  AND r5.tag = "b"
ORDER BY r3.pre

The filter is relative (does not start with /) so we link it to the previous step (here r3)
Filters

Consider: /a[]/b
Rewrite as:

/child::a[/descendant-or-self::node/child::b]

SELECT DISTINCT r2.pre
FROM table r1, table r2, table r3, table r4, table r5
WHERE r1.pre = 0
  AND r1.pre < r2.pre AND r1.post > r2.post
  AND r1.pre = r2.parent
  AND r2.tag = "a"
  AND r1.pre = r3.pre /* Start at the root */
  AND r3.pre <= r4.pre AND r3.post >= r4.post
  AND r4.pre < r5.pre AND r4.post > r5.post
  AND r4.pre = r5.parent
  AND r5.tag = "b"
ORDER BY r2.pre

The filter is absolute (starts with /) so we link it to root (r1).
Multiple filters

//a[./b][./c]: a must have a child "b" and a child "c"

SELECT DISTINCT r3.pre
FROM table r1, table r2, table r3,
     table r4, table r5,
     table r6, table r7,
WHERE r1.pre = 0
    AND r1.pre <= r2.pre AND r1.post >= r2.post
    AND r2.pre < r3.pre AND r2.post > r3.post
    AND r2.pre = r3.parent
    AND r3.tag = "a"
    AND r3.pre = r4.pre
    AND r4.pre < r5.pre AND r4.post > r5.post
    AND r4.pre = r5.parent
    AND r5.tag = "b"
    AND r3.pre = r6.pre
    AND r6.pre < r7.pre AND r6.post > r7.post
    AND r6.pre = r7.parent
    AND r7.tag = "c"
ORDER BY r2.pre
Attributes

Attribute only appear in filters. We use the .pre of the previous step and the attribute name as a key in the attribute table:

```
//a[@x]/b[@y="foo"]
```

becomes:

```
/descendant-or-self::node()/child::a[attribute::x]/child::b[attribute::y="foo"]
```

```
SELECT DISTINCT r5.pre
FROM table r1, table r2, table r3, attr_table r4,
     table r5, attr_table r6
WHERE r1.pre = 0
    AND r1.pre <= r2.pre AND r1.post >= r2.post
    AND r2.pre < r3.pre AND r2.post > r3.post
    AND r2.pre = r3.parent
    AND r3.tag = "a"
    AND r3.pre = r4.pre AND r4.name = "x"
    AND r3.pre < r5.pre AND r3.post > r5.post
    AND r3.pre = r5.parent
    AND r5.tag = "b"
    AND r5.pre = r6.pre AND r6.name = "y"
    AND r6.text = "foo"
ORDER BY r5.pre
Summary

1. Rewrite the XPath query using the extended syntax. This way you don’t have to wonder how to do //preceding::a, //following-sibling::b or .//@x. Once the query is expanded, just use the formulae step by step!

2. Filters are not more difficult. Consider two cases: the filter starts with a “/” (absolute), you must link the path in the filter to the root node. If the filter is relative then just link it to the previous step.

Reminder for assignment 5, you only need to implement:

1. /, //, following-sibling, preceding, *, tag, text() and filters
2. for the bonus part, attributes in filters and test on attributes value.