Accelerating XPath Location Steps

Brendan Briody
Hohenhausgasse 14
78462 Konstanz
E-Mail: briody@inf.uni-konstanz.de
Matrikel-Nr.: 01/461643
1. Introduction

The awareness of tree structured data in today’s IT world is nothing really new. Due to introduction of standards such as XML or tree structured directory services though, querying trees to gain specific information in able to support conventional relational database management systems (RDBMS) or arbitrary commercial data base management systems (DBMS) is an interesting area for future research and development.

At first, independent of database systems, trees can be queried by path expressions. A modern query for this is a standard called XPath, basically working with path- and axis expressions. XPath is specially made to query XML. This and XML are used as a concrete tree structured data implementation and query example in this paper.

The whole idea though is to store tree structured data on relational databases and query it by using combined path and conventional / hypothetical SQL queries.

So, secondly we want to know how to store trees into relational databases without causing inconsistencies or unnecessarily exposing too much information about the trees’ structure in the database.

Moreover the transformation of tree information into DBMS, considerable query acceleration is of great importance due to the fact that tree type structures can hold an enormous amount of nodes slowing down query time tremendously. This is one important research area on Very Large Databases (VLDB) and is the main subject of this paper.
2. Tree Traversals and Mapping (part 1)

The first step is to gain information independent of tree type or tree implementation (XML). To achieve this we need to load the tree document / structure and assign basic information to each tree node. Tree traversing is a well-known method to visit every node, beginning at the tree root and moving down the branches in a left first depth first manner. Figure 1 exhibits the tree traversal and figure 2 shows the corresponding example of typical tree type data in form of an XML document.

A well-known tree parser for XML traversals is SAX\(^1\) “Simple API for XML”. This parser allows gathering information of a document tree without maintaining the whole tree structure in memory but just scanning each node sequentially and triggering events whenever a node beginning or node end is reached. This event triggering is useful to assign values to nodes. In this case when a node beginning is found a “pre” value is assigned to a node and its value is iterated throughout the traversal. This also counts for the “post” value when the end of a node is reached.

\(^1\)http://www.saxproject.org
The node values are maintained on a stack, pushed on top at a node beginning and popped off the stack when the node end is reached. In the example above the first node holding both pre and post values and popped off the stack is node “d” Fig. 1.

Not only pre and post values are maintained while parsing but also further information is provided by SAX such as tag names and attributes, mentioned in chapter 4.

After parsing a list of nodes containing information is available. At this time we only want to examine the pre – post relationships between nodes by mapping all nodes in coordinates. Below in figure 3 the “simple” node list is shown pre-sorted with the according mapped pre-post coordinates.

<table>
<thead>
<tr>
<th>Node</th>
<th>pre</th>
<th>post</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>h</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

Fig. 3: Pre – post node list and corresponding mapping

Successfully transformed from a recursive tree structure into a 2-dimensional plane we now want to discover the relationships between nodes in a regional sense. A random node, for simplicity the node “e”, holds its 2 values pre and post which enclose a distinct region containing the nodes “g”, “f” and “h”.

Accelerating XPath Location Steps / Brendan Briody
Comparing the enclosed nodes and the tree structure, obvious relationships are discovered. This lower right region simply describes all descendants of node “e”. Carrying on, the lower left region shows all preceding siblings of “e” namely “d”. 
3. XPath Major Axes

The four major axes can be depicted as in figure 4 below.

To support all XPath axes though we need little more information. The partial “or-self” keywords of \( e/descendant-or-self::* \) for example can easily be described by just including the pre-post values of “e” into the region. The information needed to get child, parent, following- and preceding-sibling is characterised by the parent value.

If a child node of “e” is required, the information this requested node has to contain is who its parent node is. Hence node “g” would have to contain the pre value of node “e”.

To distinguish if a regarded node is an attribute or not, a Boolean value is also needed. If true we are dealing with an attribute of a node. As last of course the name of a node must be given, too. As a result we end up with a so-called 5 dimensional descriptor for any node “v” of a tree. 

\[
descr(v) = (pre(v),post(v),par(v),att(v),tag(v))
\]
4. Axes and Range Queries

Using this 5-dimensional descriptor all XPath axes can be expressed as windows displayed below in a table for an arbitrary context node “v”. Note here that the pre-post values are now intervals with round brackets excluding a value and square brackets including a value.

<table>
<thead>
<tr>
<th>Axis a</th>
<th>Query window(a, v)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pre</td>
</tr>
<tr>
<td>child</td>
<td>&lt;[pre(v), ∞)</td>
</tr>
<tr>
<td>descendant</td>
<td>&lt;[pre(v), ∞)</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td>&lt;[pre(v), ∞)</td>
</tr>
<tr>
<td>parent</td>
<td>&lt;[par(v), par(v)]</td>
</tr>
<tr>
<td>ancestor</td>
<td>&lt;[0,pre(v)]</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>&lt;[0,pre(v)]</td>
</tr>
<tr>
<td>following</td>
<td>&lt;[pre(v), ∞)</td>
</tr>
<tr>
<td>preceding</td>
<td>&lt;[0,pre(v)]</td>
</tr>
<tr>
<td>following-sibling</td>
<td>&lt;[pre(v), ∞)</td>
</tr>
<tr>
<td>preceding-sibling</td>
<td>&lt;[0,pre(v)]</td>
</tr>
<tr>
<td>attribute</td>
<td>&lt;[pre(v), ∞)</td>
</tr>
</tbody>
</table>

Table 1

The only disturbing feature here is that the intervals are nearly all expressed infinitely with ∞. To make it handier we need to shrink the intervals to fixed sizes. How this can be done is explained in chapter 6.

Beforehand, as a motivation we want to introduce a hypothetical SQL query example using axes and windows.
5. Representation in SQL

Thinking back about our 5-dimensional descriptor we can use these elements and create an SQL schema calling it “accel”.

```
accel
  | pre | pos | par | att | tag |
```

Being unique, the pre or even post attribute can be used as a primary key.
A hypothetical query here using “e” as a path expression and “a” as an XPath axis would be...

```
query(e/a) = SELECT v'.*
  FROM query(e) v , accel v'
  WHERE v' INSIDE window(a,v)

Query 1
```

meaning: Select a node v’ from a query path “e” with context node “v” and out of table accel where v’ is inside the window corresponding to the context node “v” and axis a.
The idea of query windows is well supported by R and B-Trees but is not to be further discussed in this paper.
A conventional SQL query for a path example: /descendant::n1/preceding-sibling::n2 is...

```
SELECT v2.*
  FROM accel v1, accel v2
  WHERE 0 < v1.pre
    AND v1.tag = n1
    AND v2.pre < v1.pre AND v2.post < v1.post
    AND v2.par = v1.par
    AND v2.tag = n2

Query 2
```

…meaning for the path: get all descendants under the root with the name n1 and find their preceding siblings named n2.
In SQL:
\[
\text{select } v_2 \text{ with } v_1 \text{ and } v_2 \text{ from the } \text{accel} \text{ table where } v_1 \text{ is not the root node}
\]
- \text{and } v_1 \text{ has the tag name n1}
- \text{and } v_2 \text{ (pre post) is inside the pre-post window boundaries of } v_1
- \text{and } v_1, v_2 \text{ have the same parent (sibling relationship)}
- \text{and } v_2 \text{ has the tag n2}

Of course the information for the 5-dimensional descriptor must be obtained at document loading time. Referring to chapter 2, two callback procedures are needed to gain the necessary information, a callback for the beginning of a node and one for the end of a node.

So, every time SAX triggers an element beginning we need a callback procedure described in a simple form as follows:

```plaintext
StartElement(t, a, atts)
descriptor v <--(pre = gpre, post = null, (stack.top()).pre, att=a, tag=t)
Stack.push(v)
gpre = gpre + 1
for v’ in atts  // get all attributes v’ of node v
   StartElement(v’, true, null)
EndElement(v’)
```

For a node instance “v” of the type descriptor the global integer value “gpre” is assigned to pre, “null” to post (the post value is not known at this time), the parent of this node is always the top “pre” integer value of a stored node on the stack (stack.top()..), the Boolean “a” is assigned to att and “t” to tag.

The SAX event for a node end would look like this:

```plaintext
EndElement(v’)
descriptor v <-- Stack.pop()
v.post = gpost
gpost = gpost + 1
insert v into accel
```

Every time we reach the end of an element the descriptor is popped off the stack and assigned to its post value. Now its descriptor is complete and can be stored to the accel table.
Not mentioned up to now is how we can efficiently store string data or in XML terms CDATA. Having to compare strings in queries can be tedious. In our former path in Query 2 we have to check all nodes under the root for tag name n1. A reasonable approach is to store the tag names in a separate table with the pre value as a primary key and the strings as hash values that can then be looked up in a hash table.
6. Shrink-wrapping the descendant axis

Moving away from queries themselves we want to turn to the problem of infinite intervals in query windows. To shrink windows to a fixed size, we need to gain more knowledge concerning the pre - post relationships, sizes and levels of sub nodes according to a context node.

![Sub tree with levels]

At first discovering relationships between the pre-post values of a context node and its level, the size of a sub tree according to its context node is concluded in equation (1). An obvious assumption in (2) is also that the size of a sub tree must be the pre value of the context node holding the minimum pre value for this sub tree, subtracted from the right most leaf node holding the maximum pre value.

Now taking (2) and forming it to: \( \text{pre}(h) = \text{pre}(e) + \text{size}(e) \) and replacing size(e) with (1) we get: \( \text{pre}(h) = \text{post}(e) + \text{level}(e) \)

Regarding our node descriptor we do not know about the level of any node, but we can get the height of the whole document at loading time. However height(t) is not always equal to level(v) but is definitely always larger or equal to level(v) of any node in the tree.

\[
\text{Level}(e) = \text{height}(t)
\]
With the post value of the leftmost leaf being the smallest and the rightmost leaf pre value being the largest, two inequalities can be defined by merging height(t) into:

\[
\begin{align*}
\text{pre}(h) &= \text{post}(e) + \text{height}(t) \\
\text{post}(f) &= \text{pre}(e) - \text{height}(t)
\end{align*}
\]

This gives us a fixed pre and post range for descendants of node “e” with document height(t) being 4.

\[
\begin{align*}
(3, \text{post}(e) + \text{height}(t)] & \text{ leads to } (3, 8] \text{ for the pre range} \\
[\text{pre}(e) - \text{height}(t), 4) & \text{ leads to } [-1, 4) \text{ for the post range}
\end{align*}
\]

Applying these intervals to the descriptor for the descendant axis…

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>par</th>
<th>att</th>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;(pre(v), 8),</td>
<td>[0,post(v)],</td>
<td>*</td>
<td>false</td>
<td>* &gt;</td>
</tr>
<tr>
<td>… to…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
(3, 8] \quad (-1, 4)
\]

... we get slightly larger intervals due to the fact that height(t) is a relatively small estimator which is desirable thus not giving us exact intervals such as (3, 6] and (-1, 4). Further on, we cannot only shrink the pre – post plane but also find a boundary for the lowest leaf in a tree with:

\[
\text{post} = \text{pre} - \text{height}(t) \quad (3)
\]

In our case, the lowest leaf would be node “h” which fulfils equation (3)

\[
\text{post}(h) = \text{pre}(h) - \text{height}(t) \text{ which means the farthest pre scan would be } (3, 6]
\]
The so-called shrink-wrapping of the regions accelerates queries tremendously. Tested on a B-tree based XPath Accelerator on top of a IBM DB2 and an XML instance of 1.1 MB (21051 nodes) we get ...

<table>
<thead>
<tr>
<th>Query</th>
<th>t shrunk [s]</th>
<th>t[s]</th>
<th># Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>//open_auction//description</td>
<td>0.2</td>
<td>53</td>
<td>120</td>
</tr>
<tr>
<td>//open_auction//description//listitem</td>
<td>0.32</td>
<td>55.5</td>
<td>126</td>
</tr>
<tr>
<td>//open_auction//description//listitem//keyword</td>
<td>0.34</td>
<td>124</td>
<td>90</td>
</tr>
</tbody>
</table>

... on an auction document stressing the descendant axis. The query time on a shrink-wrapped region stays nearly constant towards the non-shrunk which query time doubles between //listitem and ..//listitem//keyword . More remarkable, the XPath accelerator takes only 0.4 % of the non-shrunk query time.
7. Tree Traversals and Mapping (part 2)

A problem with pre and post scans is that they are independently performed which means a
pre scan for instance yields not only potential results but also false hits. Figure 7. exhibits the
problems of either pre or post scans.

Moving back again to chapter 2, we want to introduce another traversal and mapping method.
We apply the pre and post values in a way that the pre value, when encountering a node
beginning, is iterated as before, but at an end of an element it takes the pre value, iterates and
the post value is assigned to this value. This happens vice versa to the pre values, too. We
end up with pre post values in figure 8 and the corresponding mapping in figure 9 resolving to
a stretched form. The stretching is referred to the range of pre and post values that earlier
reached the maximum value of 7 for pre, which now is 13 and 7 for post, now holding the
maximum of 15.
The stretching not only avoids false pre post hits by setting the scan ranges according to the pre post values of a context node (figure 9) but all the leaves “l” lie on a fixed diagonal function of:

\[ \text{post}(l) = \text{pre}(l) + 1 \quad \text{or} \quad f(\text{pre}) = \text{pre} + 1 \]

This can simplify shrink-wrapping of regions.
Stretching also still keeps the features of a non-stretched mapping. The pre and post values are still unique and can be used as primary keys in the database. All axis windows (see table 1 on page 6) are valid as before and the relationships between nodes are maintained due to non absolute values. The size estimation of an arbitrary sub tree is now exact with:

\[ \text{size}(v) = \frac{1}{2} (\text{post}(v) - \text{pre}(v) - 1) \]
The only obvious drawback is that the pre post values are not continuous, causing “bumpy” pre-post scans displayed in table 2 on the right. The pre-sorted nodes can be queried for all descendants of “e” by setting the pre post interval $(4,11]$ and calculating the sub-tree with the size(e) equation on the previous page. Due to size(e) being 3 the pre scan only has to be 3 nodes large namely 5, 7, 8 inside the set interval. In case a post scan is needed, the nodes would have to be all sequentially scanned because the post values in the table do not underlie an interval $(6, 10, 9)$.

<table>
<thead>
<tr>
<th>node</th>
<th>pre</th>
<th>post</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>f</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>g</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>h</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>c</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 2
8. Performance measurements

To give an impression of the XPath Accel acceleration in comparison to another mapping scheme, here the “Edge Map”, in matters of performance on different sized documents, the Accel scheme is in all cases averagely 20% faster than the Edge Map.

<table>
<thead>
<tr>
<th>File size [MB]</th>
<th>0.12</th>
<th>0.15</th>
<th>1.1</th>
<th>31</th>
<th>55</th>
<th>120</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge map [s]</td>
<td>0.17</td>
<td>0.7</td>
<td>1.5</td>
<td>20.4</td>
<td>98.8</td>
<td>197</td>
</tr>
<tr>
<td>XPath Accel [s]</td>
<td>0.03</td>
<td>0.15</td>
<td>4.34</td>
<td>2.4</td>
<td>12.9</td>
<td>44</td>
</tr>
</tbody>
</table>

The diagram above is mapped in a logarithmic scale to make all measurements easily visible. Every document was queried about 10 times on each document file size for each query implementation to guarantee sturdy query times.
9. Conclusion

Summarising XPath acceleration the three main steps of gaining tree information firstly by traversing the tree document or –structure and setting pre-post node information in a stretched or non-stretched form, secondly defining all XPath axes as windows in a pre-post interval manner and creating a descriptor useful for an SQL schema and R / B-tree indexing, finally shrink-wrapping regions to minimise regional scans.

The most remarkable feature of this development is the fact that all procedures to query trees are based on simple calculations of the pre, post and height integer value.

Again, not only applied to XML but to any other non standard tree structured data type, XPath acceleration is certainly a very interesting framework development for the future, combining tree structured queries with relational database systems.