XML Storage in Relational Systems

Why should we want to use a relational back-end?

- Benefit from existing technology:
  - storage and index techniques,
  - optimization,
  - transaction management, recovery.
- Interoperate with (existing) relational data.
- Don’t re-invent the wheel.
- **But:** How to store XML in an RDBMS?

First Approach: Schema-Based Storage

**Idea:** Use schema information (from a DTD) to find a suitable relational schema.

- Proposed, e.g., by Shanmugasundaram et al., 1999.
- Example:

```xml
<persons>
  <person>
    <name>John Doe</name>
    <address>
      <street>13 Main Street</street>
      <city>Miami, FL 12345</city>
    </address>
  </person>
  <person>
    <name>Martha Johnson</name>
    <address>
      <street>42 Kings Road</street>
      <city>New York, NY 54321</city>
    </address>
  </person>
</persons>
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>street</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John Doe</td>
<td>13 Main Street</td>
<td>Miami, FL 12345</td>
</tr>
<tr>
<td>2</td>
<td>Martha Johnson</td>
<td>42 Kings Road</td>
<td>New York, NY 54321</td>
</tr>
</tbody>
</table>
Some benefits and drawbacks of schema-based storage:

- Takes full advantage of specialized data types, indexes,…
- Requires DTD/Schema information.
- Many issues with XML/XQuery semantics:
  - XML reconstruction,
  - document order,
  - recursive XML documents,…

Schema-based approaches are thus mainly used

- for data-centric XML, or
- if interoperation with “relational” applications is required.

There are schema-independent ways to store XML.

- XPath accelerator [Grust 2002]
  - Advantages:
    - support for all XPath axes, from any context node,
    - low space overhead,
    - efficiently implementable.
  - Idea:
    - The sequence of start/end tags in the serialized XML document implies the full tree information.
    - Encode this sequence in a numbering scheme.

The tag sequence corresponds to a depth-first, left-to-right tree traversal.

<table>
<thead>
<tr>
<th>v</th>
<th>pre</th>
<th>post</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>g</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Start tag  ➞ preorder rank \( \text{pre}(v) \)
End tag    ➞ postorder rank \( \text{post}(v) \)
The pre/post Table

Additional attributes encode XML node properties.

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>kind</th>
<th>prop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>doc</td>
<td>persons.xml</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>elem</td>
<td>persons</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>elem</td>
<td>person</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>elem</td>
<td>name</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>text</td>
<td>John Doe</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>elem</td>
<td>address</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>elem</td>
<td>street</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>text</td>
<td>13 Main Street</td>
</tr>
</tbody>
</table>

Analysis

Let us briefly assess some features of the XPath accelerator.

- No DTD/Schema information required.
- Encoding can be generated in a single document read (SAX).
- Original document can be fully reconstructed.
  - All the information is there to fully support XQuery.
  - Reconstruction can be implemented very efficiently.
- Observe that the pre value encodes document order.

XPath on pre/post Encoded Trees

Similar ideas hold for other axes.

- The ancestor axis:
  \[ v' \in v/\text{ancestor} \iff \pre(v') < \pre(v) \land \post(v') > \post(v) \]

- The preceding axis:
  \[ v' \in v/\text{preceding} \iff \pre(v') < \pre(v) \land \post(v') < \post(v) \]

- Recursion turns into simple range predicates.

  \[ v' \in v/\text{descendant} \iff \pre(v') > \pre(v) \land \post(v') < \post(v) \]
The encoding makes it easy to express XPath in SQL.

- **Example:** context/descendant::name/following::street
  
  ```sql
  SELECT DISTINCT d2.*
  FROM context ctx, document d1, document d2
  WHERE d1.pre > ctx.pre AND d1.post < ctx.post
  AND d1.kind = elem AND d1.prop = 'name'
  AND d2.pre > d1.pre AND d2.post > d1.post
  AND d2.kind = elem AND d2.prop = 'street'
  ORDER BY d2.pre
  ```

- XPath semantics requires DISTINCT and ORDER BY!

---

**The pre/post Plane**

**XPath accelerator maps trees into a two-dimensional space.**

- Formerly recursive queries turn into region queries.
- Other node properties (node kind, tag names,…) can be mapped into additional dimensions.
- Multi-dimensional data can be efficiently supported by modern database technology:
  - **R-trees**, extend B⁺-trees to n dimensions.

---

**Another View on the pre/post Table**

**pre/post as coordinates in a two-dimensional pre/post plane.**

---

**A Closer Look at the pre/post Plane**

**The pre/post plane shows typical characteristics.**

- 100 node XML instance.
- Nodes accumulate at diagonal.
- Some nodes above diagonal, none below.
- Leaf nodes near diagonal, root node top left.
This situation is well supported by R-trees.

- R-tree for the 100 node XML instance.
- Adapts well to this distribution.
- 5-dimensional R-tree evaluates XPath axis and name test in parallel.

We could as well use two $B^+$-trees (on pre and post).

Two options:
1. Use index on pre.
   - Start at v and scan along pre.
   - Many false hits!
2. Use index on post.
   - Start at v and scan along post.
   - Many false hits!

Many false hits either way!

Let’s do some math...

$$pre(v) = |v/ancestor| + |v/preceding|$$
$$post(v) = |v/preceding| + |v/descendant|$$
$$pre(v) - post(v) = \frac{|v/ancestor| - |v/descendant|}{level(v)} - size(v) \quad (*)$$

- The pre value of the last descendant node $v'$ is $pre(v') = pre(v) + size(v)$

The combination gives smaller scan regions for pre and post.

$\forall v' \in v/descendant$

$\iff$

$$pre(v') > pre(v) \land pre(v') \leq post(v) + level(v)$$
$$post(v') < post(v) \land post(v') \geq pre(v) - level(v)$$

- Similar ideas apply to other axes.
- The XML tree height is a good estimation for $level(v)$.
  - Note that the height is typically small, even for huge documents.
Shrink-Wrapping the Descendant Region

The effect is best visible in the pre/post plane.

- We now specified actual ranges.
- Range on pre or post is in fact sufficient.
- Scan range independent of document size.
- Well supported by $\B^+$-trees.

- Add level to the pre/post table.

*More on the pre/post Encoding*

So far we have mainly talked about the descendant axis.

- The preceding and following axes show similar benefit.
- The level property makes non-recursive axes easier to express:
  $$v' \in v/\text{child} \iff v' \in v/\text{descendant} \land \text{level}(v') = \text{level}(v) + 1$$
  $$v' \in v/\text{parent} \iff v' \in v/\text{ancestor} \land \text{level}(v') = \text{level}(v) - 1$$

*An Extended pre/post Table*

We include level into the pre/post table.

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>level</th>
<th>kind</th>
<th>prop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>0</td>
<td>doc</td>
<td>persons.xml</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>1</td>
<td>elem</td>
<td>persons</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>2</td>
<td>elem</td>
<td>person</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>elem</td>
<td>name</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>text</td>
<td>John Doe</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
<td>elem</td>
<td>address</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>4</td>
<td>elem</td>
<td>street</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>5</td>
<td>text</td>
<td>13 Main Street</td>
</tr>
</tbody>
</table>

*XPath Accelerator Performance*

[Grust 2002]
Handling More Than One Document

In practice, an XML system needs to deal with more than one document.

- **Idea:** Join several document trees into a single tree with a virtual new root.

![Tree diagram]

- Store this tree into a single pre/post table.
- Preserves node identity.

Jens Teubner, DBIS Group, U Konstanz

XML Query Processing on Relational Back-Ends

Handling More Than One Document

We unify more than one document into a single tree.

- **Problem:** Scans for preceding and following may return nodes from wrong documents.
- **Solution:** Add column `frag` to the table identifying the node’s document (fragment).

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>level</th>
<th>kind</th>
<th>prop</th>
<th>frag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>0</td>
<td>doc</td>
<td>persons.xml</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>1</td>
<td>elem</td>
<td>persons</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>17</td>
<td>68</td>
<td>0</td>
<td>doc</td>
<td>library.xml</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>67</td>
<td>1</td>
<td>elem</td>
<td>books</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Jens Teubner, DBIS Group, U Konstanz

XML Query Processing on Relational Back-Ends

Pre/post Variants

Variations have been derived from the pre/post encoding.

- Original axis conditions are inequalities on `pre` and `post`.
  - `pre` and `post` need not be dense, just ascending.
  - Introduce “gaps” to hold more information (→ “stretched” pre/post encoding).
- With equality (⋆), it is sufficient to store either three of `pre`, `post`, `size`, `level`.
  - Our implementation uses `pre`, `size`, and `level`.
  - `size` is invariant when copying or moving nodes.
  - Makes element construction easier to implement.