Fully (or Purely) Relational XPath and XQuery Processors

Maurice van Keulen, University of Twente, The Netherlands
Torsten Grust, University of Konstanz, Germany
Goal

Turn an ordinary RDBMS into an efficient XQuery engine for high volume XML document collections

• Existing relational DBMS as XQuery backend
  ➢ Complete relational representation of XML
  ➢ Efficient evaluation of XQuery inside the DBMS
  ➢ By making RDBMS more tree aware, significant improvements in efficiency can be obtained
  ➢ Support full XQuery exactly in accordance with XQuery semantics
  ➢ Keep change to RDBMS engine minimal
Need for **efficient XQuery evaluation** drives the construction of the following development stack:

Compilation of XQuery core to relational algebra (wednesday morning by Torsten Grust) → Validation of encoded XML against schema definition (wednesday morning by Torsten Grust) → Relational XPath evaluation (also **today**) → Relational XML encoding (**now**) → SQL, relational algebra

- **XQuery**
- **Validation**
- **XPath Axes**
- **Tree Encoding**
- **RDBMS**
• Data-centric
  – Highly structured
  – Usage stems from exchange of data from databases

• Document-centric
  – Semi-structured
  – Embedded tags
  – Fulltext search
  – Usage stems from exchange of formatted text

```
<site>
  <item ID="I001">
    <name>Chair</name>
    <description>This chair is in good condition ...</description>
  </item>
  <item ID="I002">
    <name>Table</name>
    <description>...</description>
  </item>
...</site>
```

```
<memo>
  <author>John Doe</author>
  <title>...</title>
  <body>
  This memo is meant for all persons responsible for
  <list bullets="1">
    <item>either <em>customers</em> abroad,</item>
    <item>or <em>suppliers</em> abroad.</item>
  </list>
  ... 
</memo>
```
Mapping a DTD to a relational schema: Inlining

```xml
<!ELEMENT site        (item*)>
<!ELEMENT item        (name,description*)><!ATTLIST item        (id ID #REQUIRED)>
<!ELEMENT name        (#PCDATA)>
<!ELEMENT description (#PCDATA)>
```

Disadvantages

- Suitable for data-centric XML only
- DTD or XML schema required

```sql
CREATE TABLE item (  
id INTEGER NOT NULL PRIMARY KEY,  
name VARCHAR(250) NOT NULL
)
CREATE TABLE description (  
itemID INTEGER NOT NULL,  
pCdata TEXT NOT NULL
)```
<a>
  <b>
    <c>
      <d/> <e/>
    </c>
  </b>
  <f>
    <g/>
    <h>
      <i/> <j/>
    </h>
  </f>
</a>
XML tree  

Pre/Post Plane Encoding  

RDBMS table

<table>
<thead>
<tr>
<th></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>
<tr>
<td>h</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>j</td>
<td>9</td>
<td>6</td>
</tr>
</tbody>
</table>

Preorder/coincides with order of opening tags  
Postorder/coincides with order of closing tags
Document storage

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>level</th>
<th>kind</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>0</td>
<td>doc</td>
<td>a.xml</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>elem</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>elem</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>attr</td>
<td>id</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>text</td>
<td></td>
</tr>
</tbody>
</table>

One node in tree = one row in table

**Text**

pre | text
--- | ---
4   | “…”

**Attribute**

pre | name | value
--- | --- | ---
3   | id   | “10”
**Loading & Serialization**

- **Loading**: “how to convert (textual) XML document to rows in table structure?”
  - Only one pass needed!

  - **At `startElement`:**
    1. assign `pre` + increase
    2. push `v` on stack
    3. process attributes
  - **At `endElement`:**
    1. pop `v` from stack
    2. assign `post` + increase
    3. insert into table

- **Serialization**: “how to convert table contents to (textual) XML-doc?”
  - Only one pass needed!

  - foreach `v` in table do
    1. for each node on stack with `post < post(v)`, pop from stack & print end tag
    2. if `v` is element, push on stack, print start tag; else ..
  - perform (1) for all remaining nodes on stack
Exercise 1

1. Draw tree for this XML document

2. Assign preorder and postorder ranks

3. Fill table document

```xml
<?xml version="1.0" encoding="iso-8859-1"?>
<orders>
  <order total="10.89">
    <line>
      <article id="10">Pencil</article>
      <price>1.95</price>
    </line>
    <line>
      <article id="23">
        Paper (<weight>80gr</weight>)
      </article>
      <price>6.99</price>
    </line>
  </order>
  <order total="1.95">
    <line>
      <article id="10">Pencil</article>
      <price>1.95</price>
    </line>
  </order>
</orders>
```
Need for efficient XQuery evaluation drives the construction of the following development stack:

Compilation of XQuery core to relational algebra (wednesday morning by Torsten Grust) →
Validation of encoded XML against schema definition (wednesday morning by Torsten Grust) →
Relational XPath evaluation (now) →
  XPath accelerator √

  SQL, relational algebra
**XPath axes**

in the pre/post plane

```
XPath axes

ancestor
parent
self
following sibling
preceding sibling
child
attribute
descendant
namespace
ancestor-or-self = ancestor ∪ self
descendant-or-self = descendant ∪ self
```

pre

post

a
b
c
d
e
f
h
i
j

(0,0) 8

5
<table>
<thead>
<tr>
<th>Axis</th>
<th>pre</th>
<th>post</th>
<th>Level</th>
<th>kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>child</td>
<td>(pre(ν),∞)</td>
<td>[0,post(ν)]</td>
<td>level(ν)+1</td>
<td>elem</td>
</tr>
<tr>
<td>descendant</td>
<td>(pre(ν),∞)</td>
<td>[0,post(ν)]</td>
<td>*</td>
<td>elem</td>
</tr>
<tr>
<td>desc-or-self</td>
<td>(pre(ν),∞)</td>
<td>[0,post(ν)]</td>
<td>*</td>
<td>elem</td>
</tr>
<tr>
<td>parent</td>
<td>[0,pre(ν)]</td>
<td>(post(ν),∞)</td>
<td>level(ν)-1</td>
<td>elem</td>
</tr>
<tr>
<td>ancestor</td>
<td>[0,pre(ν)]</td>
<td>(post(ν),∞)</td>
<td>*</td>
<td>elem</td>
</tr>
<tr>
<td>anc-or-self</td>
<td>[0,pre(ν)]</td>
<td>(post(ν),∞)</td>
<td>*</td>
<td>elem</td>
</tr>
<tr>
<td>following</td>
<td>(pre(ν),∞)</td>
<td>(post(ν),∞)</td>
<td>*</td>
<td>elem</td>
</tr>
<tr>
<td>preceding</td>
<td>[0,pre(ν)]</td>
<td>[0,post(ν)]</td>
<td>*</td>
<td>elem</td>
</tr>
<tr>
<td>foll-sibling</td>
<td>(pre(ν),∞)</td>
<td>(post(ν),∞)</td>
<td>level(ν)</td>
<td>elem</td>
</tr>
<tr>
<td>prec-sibling</td>
<td>[0,pre(ν)]</td>
<td>[0,post(ν)]</td>
<td>level(ν)</td>
<td>elem</td>
</tr>
<tr>
<td>attribute</td>
<td>(pre(ν),∞)</td>
<td>[0,post(ν)]</td>
<td>level(ν)+1</td>
<td>attr</td>
</tr>
</tbody>
</table>
XPath evaluation (approach)

XPath path expression $s_1 / s_2 / \ldots / s_n$
- Each step $s_i$ is of the form $axis::nodetest$
- Each step $s_i$ results in context node sequence for step $s_{i+1}$

Evaluation in DBMS
- Apply each location step to all nodes in context (bulk-oriented query processing)
- Preserve document order and not produce duplicate nodes

Initial focus on major axis steps
- Major: ancestor, descendant, preceding, and following
- Other axes define efficiently computable subsets of the four major axes
Example query:
\[
\text{context}/\text{following::node()} / \text{descendant::node()}
\]

SQL Query

```sql
SELECT DISTINCT v2.pre
FROM context c, doc v1, doc v2
WHERE v1.pre > c.pre
AND v1.post > c.post
AND v2.pre > v1.pre
AND v2.post < v1.post
ORDER BY v2.pre
```

IBM DB2 Query Plan

Efficient with concatenated \textit{pre/post} B-tree over \textit{doc}, but remains \textit{ignorant of many useful tree properties}
• Write the SQL-query for
\( \text{context/following::node() / descendant::node()} \)
(all descendant elements that have a child with name ‘e’)

Example query:
\( \text{context/following::node() / descendant::node()} \)

SQL Query
```
SELECT DISTINCT v2.pre
FROM context c, doc v1, doc v2
WHERE v1.pre > c.pre
    AND v1.post > c.post
    AND v2.pre > v1.pre
    AND v2.post < v1.post
ORDER BY v2.pre
```

document

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>level</th>
<th>kind</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>0</td>
<td>doc</td>
<td>a.xml</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>elem</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>elem</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>attr</td>
<td>id</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>elem</td>
<td>e</td>
</tr>
</tbody>
</table>
Avoiding duplicates: \[(f, g, h, j)/\text{ancestor-or-self::*}\]

\[= (g, j)/\text{ancestor-or-self::*}\]

Pruning requires only a single sequential scan over the \textit{pre/post} table.
Tree knowledge 2: partitioning

(a, b, c, d, e, f, g, h, i, j)

partition 1

pre

(a, b, c, d, e, f, g, h, i, j)

partition 2
Basic Staircase Join Algorithm

\[ \text{context \hspace{1cm} doc} \]

\begin{verbatim}
context \hspace{1cm} doc =
begin
    desc doc =
        result = new table (pre, post);
        /* partition \( c_{from} \ldots c_{to} \) */
        c_{from} = first node in context;
        while (c_{to} = next node in context) do
            if c_{to}.post < c_{from}.post then
                /* prune */
            else
                scanpartition_desc (c_{from}.pre+1, c_{to}.pre-1, c_{from}.post);
                c_{from} = c_{to};
        n = last node in doc;
        scanpartition_desc (c_{from}.pre+1, n.pre, c_{from}.post);
        return result;
end

scanpartition_desc (pre1, pre2, post)
begin
    for i from pre1 to pre2 do
        if doc[i].post < post then
            append doc[i] to result;
    end
\end{verbatim}

\[(b,c,h) \xrightarrow{\text{desc}} \text{doc}\]
1) It scans doc and context tables *sequentially*
2) It scans both tables only *once* for an entire context node sequence
3) It *never* delivers duplicate nodes
4) Result nodes are produced in *document order*
5) Input for staircase join can be *any* node sequence

⇒ (3) + (4) ⇒ no post-processing (unique/sort) is needed to comply to XPath semantics
Example query: \(/\text{descendant::profile/descendant::education}\)
Consider ‘b’ and ‘f’: ‘f’ is a following node of ‘b’. Two following nodes can never have common descendants! So there is no reason to scan beyond the first following node encountered. ⇒ We are encoding trees; nodes are not arbitrarily spread out in the plane: there are empty regions.

\[
\text{scanpartition_desc}(\pre_1, \pre_2, \post) \\
\begin{align*}
\text{begin} & \text{ for } i \text{ from } \pre_1 \text{ to } \pre_2 \text{ do} \\
& \text{ if } \text{doc}[i].\post < \post \text{ then} \\
& \quad \text{append } \text{doc}[i] \text{ to } \result; \\
& \quad \text{else break; /* skip */}
\end{align*}
\end{align*}
\]

- Never touch more than \(|\text{context}| + |\text{result}|\) nodes
- A characterization of the location of ‘f’ can be found before scan starts (estimated skipping)
For any two nodes ‘a’ and ‘b’

(1) Nodes $a$ and $b$ relate to each other on the ancestor/descendant axis.

(2) Nodes $a$ and $b$ relate to each other on the preceding/following axis.
There is a very useful law that relates preorder rank, postorder rank, and the number of descendants of a node.

- **pre(v)** = |v/ancestor| + |v/preceding| + 1
  (in a preorder traversal of a tree, you encounter *all* ancestors and preceding nodes of a node v)

- **post(v)** = |v/descendant| + |v/preceding| + 1
  (in a postorder traversal of a tree, you encounter *all* descendants and preceding nodes of a node v)

- \[ \Rightarrow post(v) - pre(v) = |v/descendant| - |v/ancestor| \]

- |v/descendant| is often called **size(v)**
- |v/ancestor| is often called **level(v)**

- Conclusion: **size(v) = post(v) - pre(v) + level(v)**
Tree knowledge 4: Shrink-wrapping the // axis

For any v: \( \text{post}(v) - \text{pre}(v) + \text{level}(v) = \text{size}(v) \)

If \( \text{level}(v) \) is unavailable, we can estimate the number of descendants, since \( \text{level}(v) \leq \text{height} \):

\[ \text{post}(v) - \text{pre}(v) \leq \text{size}(v) \leq \text{post}(v) - \text{pre}(v) + \text{height} \]

Hence: \( \text{window}(\text{descendant}, v) = \)

\[ (\text{pre}(v), \text{post}(v)+\text{level}(v)], \]
\[ [\text{pre}(v)-\text{level}(v), \text{post}(v)) \]

or (estimated):

\[ (\text{pre}(v), \text{post}(v)+\text{height}], \]
\[ [\text{pre}(v)-\text{height}, \text{post}(v)) \]
\[ size(v) = \text{post}(v) - \text{pre}(v) + \text{level}(v) \]

\[ \Rightarrow \text{post}(v) - \text{pre}(v) \leq size(v) \leq \text{post}(v) - \text{pre}(v) + h \]

This removes an IF from the inner loop in scannapartition. This allows a CPU to accurately predict if branches are taken or not. Calculations with CPU cycles show that scan-loop takes 17 CPU cycles and copy-loop only 5! (Intel Pentium 4)
A stretched pre/post plane

- The exact pre and post ranks are unimportant, only the order
  - Couple pre and post

- For descendants:
  - \( \text{pre}(c) \leq \text{pre}(v) \leq \text{post}(c) \)
  - \( \text{pre}(c) \leq \text{post}(v) \leq \text{post}(c) \)

- We can choose query window:
  - pre between \( \text{pre}(c) \), \( \text{post}(c) \)
  - post between \( \text{pre}(c) \), \( \text{post}(c) \)

- Furthermore
  - \( \text{size}(v) = \frac{1}{2} (\text{post}(v) - \text{pre}(v) - 1) \)

Pre: 0<1<2<3<5<9<…
Post: 4<7<8<11<14<…
Example query: \( /\text{descendant::profile/descendant::education} \)

**Query plan 1**

- \( \sigma_{\text{education}} \)
- \( \sigma_{\text{profile}} \)
- \( \sigma_{\text{doc}} \)
- \( \sigma_{\text{root}} \)

**Push selection through staircase join**

**Query plan 2**

- \( \sigma_{\text{education}} \)
- \( \sigma_{\text{profile}} \)
- \( \sigma_{\text{root}} \)
- \( \sigma_{\text{doc}} \)
Query optimization

• How to choose among alternative query plans?
  – Cost estimation!

• Result size estimation:
  – Given context sequence CS
    Approximate |CS/axis| or |nodetest(CS)|

• Execution model of staircase join algorithm:
  – Context: main-memory access patterns
  – Sequential traversal of context sequence
  – Sequential traversal of document sequence with average-length skips
  – Sequential traversal of result sequence
  – Consider CPU cache misses as if all three happen in parallel
Experiments: effect of skipping

Query: `/ descendant:: profile/ descendant:: education`

Interm. results (1GB XMark) 47,015,212  127,984  1,849,360  63,793
Query 1: `/descendant::*profile/descendant::*education`

Query 2: `/descendant::*increase/ancestor::*bidder`
Conclusions

- XPath accelerator
  - Complete relational representation of XML
- Staircase join
  - Encapsulates “tree knowledge” by using pre/post encoding, pruning, partitioning, and skipping
  - Local change to RDBMS kernel
  - Provides efficient evaluation of XPath axis steps

Compilation of XQuery core to relational algebra
(wednesday morning by Torsten Grust)

Validation of encoded XML against schema definition
(wednesday morning by Torsten Grust)

- Staircase join √
- XPath accelerator √
- SQL, relational algebra