Part XII

Mapping XML to Databases
Outline of this part

1. Mapping XML to Databases
   - Introduction

2. Relational Tree Encoding
   - Dead Ends

3. XPath Accelerator Encoding
   - Tree Partitions and XPath Axes
   - Pre-Order and Post-Order Traversal Ranks
   - Relational Evaluation of XPath Location Steps
Mapping XML to Databases

We now start to look at our preferred mapping direction:
- How do we put XML data into a database?
- ... and how do we get it back efficiently?
- ... and how do we run (XQuery) queries on them?

We will call the mapping \( \mathcal{E} \) an *encoding* in the sequel.
Exploiting DB technology

In doing so, our main objective is to use as much of existing DB technology as possible (so as to avoid having to re-invent the wheel).

- **XQuery operations** on trees, XPath traversals and node construction in particular, should be **mapped into operations over the encoded database:**

\[
\begin{array}{c}
\text{XPath/construction} \\
\downarrow \varepsilon \\
\text{relational query} \\
\downarrow \\
\text{relational query} \\
\end{array} 
\]

Our goal: let the database do the work!

- Obviously, \( \varepsilon \) needs to be chosen judiciously. In particular, a faithful **back-mapping** \( \varepsilon^{-1} \) is absolutely required.
How can we exploit DB technology?

1. Reuse knowledge gained by the DB community while you implement a "native" XML database management system from scratch.
   - It is often argued that, if you want to implement a new data model efficiently, there's no other choice.

2. Reuse existing DB technology and systems by defining an appropriate mapping of data structures and operations.
   - Often, relational DBMS technology is most promising, since it is most advanced and mature.
   - The challenge is to gain efficiency and not lose benchmarks against "native" systems!
Native XML processors

... need external memory representations of XML documents, too!

- Main-memory representations, such as a DOM tree, are insufficient, since they are only suited for “toy” examples (even with today’s huge main memories, you want persistent storage).
- Obviously, native XML databases have more choices than those offered on top of a relational DBMS.
- We will have to see whether this additional freedom buys us significant performance gains, and
- what price is incurred for “replicating” RDBMS functionality.
Relational XML processors (1)

Recall our principal mission in this course:

**Database-supported XML processors**

We will use *relational database technology* to develop a highly efficient, scalable processor for **XML** languages like XPath, XQuery, and XML Schema.

We aim at a **truly (or purely) relational approach** here:

- Re-use existing relational database infrastructure—table storage layer and indexes (*e.g.*, B-trees), SQL or algebraic query engine and optimizer—and invade the database kernel in a very limited fashion (or, ideally, not at all).
Relational XML processors (2)

Our approach to **relational XQuery processing**:

- The XQuery data model—ordered, unranked trees and ordered item sequences—is, in a sense, alien to a relational database kernel.
- A **relational tree encoding** $\mathcal{E}$ is required to map trees into the relational domain, *i.e.*, tables.
What makes a good (relational) (XML) tree encoding?

**Hard requirements:**

1. \( \mathcal{E} \) is required to reflect **document order** and **node identity**.
   - Otherwise: cannot enforce XPath semantics, cannot support \(<\) and \(\leq\), cannot support node construction.

2. \( \mathcal{E} \) is required to encode the **XQuery DM node properties**.
   - Otherwise: cannot support XPath axes, cannot support XPath node tests, cannot support atomization, cannot support validation.

3. \( \mathcal{E} \) is able to encode any well-formed **schema-less** XML fragment (i.e., \( \mathcal{E} \) is “**schema-oblivious”**, see below).
   - Otherwise: cannot process non-validated XML documents, cannot support arbitrary node construction.
What makes a good (relational) (XML) tree encoding?

Soft requirements (primarily motivated by performance concerns):

4. **Data-bound operations** on trees (potentially delivering/copying lots of nodes) should map into efficient database operations.
   - *XPath location steps* (12 axes)

5. Principal, recurring **operations imposed by the XQuery semantics** should map into efficient database operations.
   - *Subtree traversal* (atomization, element construction, serialization).

For a relational encoding, “database operations” always mean “table operations” . . .
Dead end #1: Large object blocks

- Import **serialized XML fragment as-is** into tuple fields of type CLOB or BLOB:

<table>
<thead>
<tr>
<th>uri</th>
<th>xml</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;foo.xml&quot;</td>
<td>&lt;a id=&quot;0&quot;&gt;b&gt;fo&lt;/b&gt;o...&lt;/a&gt;</td>
</tr>
</tbody>
</table>

- The CLOB column content is monolithic and **opaque with respect to the relational query engine**: a relational query cannot inspect the fragment (but extract and reproduce it).
- The database kernel needs to incorporate (or communicate with) an **extra XML/XPath/XQuery processor** ⇒ frequent re-parsing will occur.
- This is *not* a relational encoding in our sense.
- But: see SQL/XML functionality mentioned earlier!
Dead end #2: Schema-based encoding

XML address database (excerpt)

```xml
<person>
  <name><first>John</first><last>Foo</last></name>
  <address><street>13 Main St</street>
    <zip>12345</zip><city>Miami</city>
  </address>
</person>

<person>
  <name><first>Erik</first><last>Bar</last></name>
  <address><street>42 Kings Rd</street>
    <zip>54321</zip><city>New York</city>
  </address>
</person>
```

Schema-based relational encoding: table person

<table>
<thead>
<tr>
<th>id</th>
<th>first</th>
<th>last</th>
<th>street</th>
<th>zip</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>John</td>
<td>Foo</td>
<td>13 Main St</td>
<td>12345</td>
<td>Miami</td>
</tr>
<tr>
<td>1</td>
<td>Erik</td>
<td>Bar</td>
<td>42 Kings Rd</td>
<td>54321</td>
<td>New York</td>
</tr>
</tbody>
</table>
Dead end #2: Schema-based encoding

- Note that the schema of the “encoding” relation assumes a quite regular element nesting in the source XML fragment.
  - This regularity either needs to be discovered (during XML encoding) or read off a **DTD** or **XML Schema description**.
  - Relation person is **tailored to capture the specific regularities** found in the fragment.

**Further issues:**

- This encodes **element-only content** only (i.e., content of type element(*)*) or text() and fails for **mixed content**.
- Lack of any support for the XPath horizontal axes (e.g., following, preceding-sibling).
# Dead end #2: Schema-based encoding

## Irregular hierarchy

```xml
<a no="0">
  <b><c>X</c><c/></b>
</a>
<a no="1">
  <b><c>Y</c><c/></b>
</a>
<a><b/></a>
<a no="3"/>
```

## A relational encoding

<table>
<thead>
<tr>
<th>id</th>
<th>@no</th>
<th>b</th>
<th></th>
<th>id</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>α</td>
<td></td>
<td>1</td>
<td>α</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>β</td>
<td></td>
<td>2</td>
<td>α</td>
<td>NULL&lt;sup&gt;c&lt;/sup&gt;</td>
</tr>
<tr>
<td>5</td>
<td>NULL&lt;sup&gt;a&lt;/sup&gt;</td>
<td>γ</td>
<td>NULL&lt;sup&gt;b&lt;/sup&gt;</td>
<td>4</td>
<td>β</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>NULL&lt;sup&gt;b&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Issues:

- Number of encoding tables depends on nesting depth.
- Empty element c encoded by NULL<sup>c</sup>, empty element b encoded by absence of γ (will need *outer join* on column b).
- NULL<sup>a</sup> encodes absence of attribute, NULL<sup>b</sup> encodes absence of element.
- Document order/identity of b elements only implicit.
Dead end #3: Adjacency-based encoding

Adjacency-based encoding of XML fragments

```xml
<a id="0">
  <b>fo</b><o>
  <c>
    <d>b</d><e>ar</e>
  </c>
</a>
```

Resulting relational encoding

<table>
<thead>
<tr>
<th>id</th>
<th>parent</th>
<th>tag</th>
<th>text</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>a</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>@id</td>
<td>NULL</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>b</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>NULL</td>
<td>&quot;fo&quot;</td>
<td>NULL</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>NULL</td>
<td>&quot;o&quot;</td>
<td>NULL</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>c</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Torsten Grust (WSI) Database-Supported XML Processors Winter 2008/09 335
Dead end #3: Adjacency-based encoding

- **Pro:**
  - Since this captures all adjacency, kind, and content information, we can—in principle—**serialize the original XML fragment.**
  - **Node identity** and **document order** is adequately represented.

- **Contra:**
  - The XQuery processing model is not well-supported: subtree traversals require **extra-relational** queries (**recursion**).
  - This is completely parent–child centric. How to support descendant, ancestor, following, or preceding?
Tree partitions and XPath axes

Axes: descendant, ancestor, preceding, following

Given an arbitrary context node $\circ$, the XPath axes descendant, ancestor, preceding, following cover and partition the tree containing $\circ$. 
Tree partitions and XPath axes

Context node (here: f) is arbitrary

\[
\{a \ldots j\} = \{f\} \cup \bigcup_{\alpha \in \{\text{preceding, descendant, ancestor, following}\}} f/\alpha::\text{node}()
\]

**NB:** Here we assume that no node is an attribute node. Attributes treated separately (recall the XPath semantics).
The XPath Accelerator tree encoding

We will now introduce the **XPath Accelerator**, a relational tree encoding based on this observation.

- If we can exploit the partitioning property, the encoding will represent each tree node exactly once.
- In a sense, the semantics of the XPath axes descendant, ancestor, preceding, and following will be “built into” the encoding ⇒ “XPath awareness”.
- XPath accelerator is **schema-oblivious** and **node-based**: each node maps into a row in the relational encoding.
Pre-order and post-order traversal ranks

Pre-order/post-order traversal

(During a single scan through the document:) To each node \( v \), assign its pre-order and post-order traversal ranks \( \langle \text{pre}(v), \text{post}(v) \rangle \).

Pre-order/post-order traversal rank assignment

```
0 a 9
1 b 1
2 c 0
3 d 2
4 e 8
5 f 5
6 g 3
7 h 4
8 i 7
9 j 6
```

Torsten Grust (WSI)  Database-Supported XML Processors  Winter 2008/09  340
Pre-order/post-order: Tree isomorphism

\[ pre(v_1) \lessdot v_2 \iff pre(v_1) < pre(v_2) \]
\[ v_1 \text{ is } v_2 \iff pre(v_1) = pre(v_2) \]
XPath axes in the pre/post plane

Plane partitions \( \equiv \) XPath axes, \( \circ \) is arbitrary!

Pre/post plane regions \( \equiv \) major XPath axes

The major XPath axes descendant, ancestor, following, preceding correspond to rectangular pre/post plane windows.
XPath Accelerator encoding

XML fragment $f$ and its skeleton tree

```xml
<a>
  <b>c</b>
  <!--d-->
  <e><f><g/></f><?h?/><f/>
  <i>j</i>
</e>
</a>
```

Pre/post encoding of $f$: table `accel`

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>par</th>
<th>kind</th>
<th>tag</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>NULL</td>
<td>elem</td>
<td>a</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>elem</td>
<td>b</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>text</td>
<td>c</td>
<td>NULL</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>com</td>
<td>d</td>
<td>NULL</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0</td>
<td>elem</td>
<td>e-f</td>
<td>NULL</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>elem</td>
<td>g</td>
<td>NULL</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5</td>
<td>elem</td>
<td>h</td>
<td>NULL</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>5</td>
<td>pi</td>
<td>i</td>
<td>NULL</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>4</td>
<td>elem</td>
<td>i</td>
<td>NULL</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
<td>text</td>
<td>j</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Relational evaluation of XPath location steps

Evaluate an XPath location step by means of a window query on the pre/post plane.

1. Table accel encodes an XML fragment,
2. table context encodes the context node sequence (in XPath accelerator encoding).

### XPath location step (axis $\alpha$) $\Rightarrow$ SQL window query

```sql
SELECT DISTINCT v'.*
FROM context v, accel v'
WHERE v' INSIDE window($\alpha$, v)
ORDER BY v'.pre
```
10 XPath axes and *pre/post* plane windows

Window def’s for axis $\alpha$, name test $t$ ($* = \text{don’t care}$)

<table>
<thead>
<tr>
<th><strong>Axis $\alpha$</strong></th>
<th><strong>Query window window($\alpha::t, v$)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>child</td>
<td>$\langle (v.pre, <em>), (</em>, v.post), v.pre, elem, t \rangle$</td>
</tr>
<tr>
<td>descendant</td>
<td>$\langle (v.pre, <em>), (</em>, v.post), *, elem, t \rangle$</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td>$\langle [v.pre, <em>], (</em>, v.post], *, elem, t \rangle$</td>
</tr>
<tr>
<td>parent</td>
<td>$\langle v.par, (v.post,*), *, elem, t \rangle$</td>
</tr>
<tr>
<td>ancestor</td>
<td>$\langle (<em>, v.pre), (v.post,</em>), *, elem, t \rangle$</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>$\langle (<em>, v.pre], [v.post,</em>], *, elem, t \rangle$</td>
</tr>
<tr>
<td>following</td>
<td>$\langle (v.pre,<em>), (v.post,</em>), *, elem, t \rangle$</td>
</tr>
<tr>
<td>preceding</td>
<td>$\langle (<em>, v.pre), (</em>, v.post), *, elem, t \rangle$</td>
</tr>
<tr>
<td>following-sibling</td>
<td>$\langle (v.pre,<em>), (v.post,</em>), v.par, elem, t \rangle$</td>
</tr>
<tr>
<td>preceding-sibling</td>
<td>$\langle (<em>, v.pre), (</em>, v.post), v.par, elem, t \rangle$</td>
</tr>
</tbody>
</table>

\[41\text{Missing axes in this definition: self and attribute.}\]
**Pre/post plane window ⇒ SQL predicate**

**descendant::foo, context node v**

\[ v' \text{ INSIDE } \langle(v.pre, *), (*, v.post), *, elem, foo\rangle \]

\[ \equiv \]

\[ v'.pre > v.pre \text{ AND } v'.post < v.post \text{ AND } v'.kind = elem \text{ AND } v'.tag = foo \]

**ancestor-or-self::*, context node v**

\[ v' \text{ INSIDE } \langle(*, v.pre], [v.post, *), *, elem, *\rangle \]

\[ \equiv \]

\[ v'.pre <= v.pre \text{ AND } v'.post >= v.post \text{ AND } v'.kind = elem \]
\((e,f)/\text{descendant::node}()\)

### Context & frag. encodings

```
\begin{array}{llll}
\text{context} & \text{accel} \\
\hline
\text{pre} & \text{post} & \cdots \\
5 & 5 & \cdots \\
4 & 8 & \cdots \\
\end{array}
```

### SQL query with expanded `window()` predicate

```sql
SELECT DISTINCT v1.*
FROM context v, accel v1
WHERE v1.pre > v.pre AND v1.post < v.post
ORDER BY v1.pre
```
Compiling XPath into SQL

**path**: an XPath to SQL compilation scheme (sketch)

\[
\text{path}(\text{fn:root()} ) = \quad \text{SELECT} \quad v'.* \\
\text{FROM} \quad \text{accel} \ v' \\
\text{WHERE} \quad v'.pre = 0
\]

\[
\text{path}(c/\alpha) \quad = \quad \text{SELECT} \quad \text{DISTINCT} \quad v'.* \\
\text{FROM} \quad \text{path}(c) \ v, \text{accel} \ v' \\
\text{WHERE} \quad v' \text{ INSIDE window}(\alpha, v) \\
\text{ORDER BY} \quad v'.pre
\]

\[
\text{path}(c[\alpha]) \quad = \quad \text{SELECT} \quad \text{DISTINCT} \quad v.* \\
\text{FROM} \quad \text{path}(c) \ v, \text{accel} \ v' \\
\text{WHERE} \quad v' \text{ INSIDE window}(\alpha, v) \\
\text{ORDER BY} \quad v.pre
\]
An example: Compiling XPath into SQL

Compile \( \text{fn:root()}/\text{descendant::a/child::text()} \)

\[
\begin{align*}
\text{path(fn:root()}/\text{descendant::a/child::text()} & = \\
\text{SELECT DISTINCT } v_1.* & \text{ FROM } \text{path(fn:root/descendant::a)} v, \text{accel } v_1 \\
\text{WHERE } v_1 \text{ INSIDE window(child::text(), } v) & \\
\text{ORDER BY } v_1.\text{pre} & = \\
\text{SELECT DISTINCT } v_1.* & \\
\text{FROM } \left( \begin{array}{l}
\text{SELECT DISTINCT } v_2.* \\
\text{ FROM } \text{path(fn:root)} v, \text{accel } v_2 \\
\text{WHERE } v_2 \text{ INSIDE window(descendant::a, } v) \\
\text{ORDER BY } v_2.\text{pre} \\
\text{accel } v_1 \\
\text{WHERE } v_1 \text{ INSIDE window(child::text(), } v) \\
\text{ORDER BY } v_1.\text{pre}
\end{array} \right) v,
\end{align*}
\]
Does this lead to efficient SQL? Yes!

- Compilation scheme $path(\cdot)$ yields an SQL query of nesting depth $n$ for an XPath location path of $n$ steps.
  - On each nesting level, apply ORDER BY and DISTINCT.

**Observations:**

1. All but the outermost ORDER BY and DISTINCT clauses may be safely removed.
2. The nested SELECT-FROM-WHERE blocks may be *unnested* without any effect on the query semantics.
Result of \(\text{path}(\cdot)\) simplified and unnested

\[\text{path}(\text{fn:root()}/\text{descendant::a}/\text{child::text()})\]

\[
\begin{align*}
\text{SELECT} & \quad \text{DISTINCT } v_1.\ast \\
\text{FROM} & \quad \text{accel } v_3, \text{accel } v_2, \text{accel } v_1 \\
\text{WHERE} & \quad v_1 \text{ INSIDE } \text{window}(\text{child::text()}, v_2) \\
\text{AND} & \quad v_2 \text{ INSIDE } \text{window}(\text{descendant::a}, v_3) \\
\text{AND} & \quad v_3.pre = 0 \\
\text{ORDER BY} & \quad v_1.pre
\end{align*}
\]

- An XPath location path of \(n\) steps leads to an \(n\)-fold **self join** of encoding table \(\text{accel}\).

- The join conditions are
  - **conjunctions** ✓ of
  - **range** or **equality predicates** ✓.

\text{multi-dimensional window!}