Other Approaches to XQuery Processing

Stefan Manegold
Stefan.Manegold@cwi.nl
http://www.cwi.nl/~manegold/
Schedule

• 12.11.2009:
  • RDBMS back-end support for XML/XQuery (1/2):
    • Document Representation (XPath Accelerator, Pre/Post plane)
  • XPath navigation (Staircase Join)

• 19.11.2009:
  • XQuery to Relational Algebra Compiler:
    • Item- & Sequence- Representation
    • Efficient FLWoR Evaluation (Loop-Lifting)
    • Optimization

• 26.11.2009:
  • RDBMS back-end support for XML/XQuery (2/2):
    • Updateable Document Representation

• 03.12.2009:
  • Other (DB-) approaches to XML/XQuery processing
Topics

• Other approaches & techniques (*selection, far from complete!*)
  • Document storage / tree encoding:
    • ORDPATH
    • DataGuides
  • XPath processing:
    • Tree patterns, holistic twig joins
**Fixed-Width Tree Encodings & Updates**

- **Fixed-width** tree encoding (like XPath Accelerator) are
  - Good for read-only processing
    - small footprint, positional lookup, staircase join
  - But inherently **static**
- **Milo et al., PODS 2002:**

  “There is a sequence of updates (subtree insertions) for any persistent tree encoding scheme $E$ (where each node keeps its initial encoding label even under updates), such that $E$ needs labels of length $\Omega(N)$ to encode the resulting tree of $N$ nodes.”
XML/XQuery

Updates

do insert <k><l/><m/></k> as first into /a/f/g
XML/XQuery Updates

MonetDB/XQuery

hack:

exploit paging & mmap trick

but:

updating pg|off is still $O(N)$

Read–Only vs. Updatable Representation

<table>
<thead>
<tr>
<th>pre</th>
<th>size</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rid</th>
<th>size</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>null</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>null</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rid</th>
<th>size</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>null</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>null</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>null</td>
</tr>
<tr>
<td>19</td>
<td>4</td>
<td>null</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>null</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pre</th>
<th>size</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>null</td>
</tr>
</tbody>
</table>

here: pagesize = 8 unused space:
- level = NULL
- size set to unite consecutive space

do insert `<k>` `<l/>` `<m/>` `</k>` as first into `/a/f/g`

- first try to handle the insert inside a page
- if full, append pages (NULL padded)

- pre|size|level is table VIEW with pages in logical order
- pre is a ROWNUM in the view, therefore it adapts automatically
Fixed-Width Tree Encodings & Updates

- **Fixed-width** tree encoding (like XPath Accelerator) are
  - Good for read(-only) processing
    - small footprint, positional lookup, staircase join
  - But inherently **static**

- **Non-solutions:**
  - **Gaps** in the encoding (*never large enough*)
  - Encoding based on **decimal fractions** (*limited precision*)

- **Possible solution:**
  - **Variable-width** tree encodings:
    - Cheaper updates
    - At the expense of more expensive read(-only) processing
A Variable-Width Tree Encoding: ORDPATH

- The ORDPATH encoding (used in MS SQL Server™) assigns node labels of variable length.

**ORDPATH labels for an XML fragment**

1. The fragment root receives label 1.

2. The \( n \)th \((n = 1, 2, \ldots)\) child of a parent node labelled \( p \) receives label \( p \cdot (2 \cdot n - 1) \).

- Internally, ORDPATH labels are not stored as \( . \)-separated ordinals but using a prefix-encoding (similarities with Unicode).

- O'Neil et al., SIGMOD 2004.
ORDPATH Encoding: Example

ORDPATH encoding of a sample XML fragment

```
<a>
  <b/>
  <c>
    <d/><e/>
    <f><g/></f>
  </c>
  <h>
    <i/><j/><k/>
  </h>
</a>
```

• Note:
  ▶ **Lexicographic** order of ORDPATH labels \(\equiv\) document order
  \(\Rightarrow\) Clustered index on ORDPATH labels will be helpful.
In ORDPATH, the insertion of new nodes between two existing sibling nodes is referred to as “caring in” (caret \(\wedge\) insertion mark, \(<\)).

**ORDPATH: node insertion**

Let \((v_1, \ldots, v_n)\) denote a sequence of nodes to be inserted between two existing sibling nodes with labels \(p \cdot s\) and \(p \cdot (s + 2)\), \(s\) odd. After insertion, the new label of \(v_i\) is

\[
p \cdot (s + 1) \cdot (2 \cdot i - 1).
\]

Label \(p \cdot (s + 1)\) is referred to as a caret.
**ORDPATH: Insertion Between Siblings**

Insertion of \(<l/>\), \(<m/>\) between \(<j/>\) and \(<k/>\)
Determine ORDPATH label of new node \( v \) inserted

1. to the right of \(<k/>\),
2. to the left of \(<i/>\),
3. between \(<j/>\) and \(<l/>\),
4. between \(<l/>\) and \(<m/>\).
Is ORDPATH suitable for XQuery?

- Mapping core operations of the XQuery processing model to operations on ORDPATH labels:

\[ v / \text{parent}:: \text{node}() \]

1. Let \( p \cdot m \cdot n \) denote \( v \)'s label (\( n \) is odd).
2. If the rightmost ordinal (\( m \)) is even, remove it. Goto 2.

In other words: the carets (\( \wedge \)) do not count for ancestry.

\[ v / \text{descendant}:: \text{node}() \]

1. Let \( p \cdot n \) denote \( v \)'s label (\( n \) is odd).
2. Perform a lexicographic index range scan from \( p \cdot n \) to \( p \cdot (n + 1) \)-the virtual following sibling of \( v \).
• For a 10 MB XML sample document, the authors of ORDPATH observed label lengths between 6 and 12 bytes.

• ORDPATH labels encode root-to-node paths => common prefixes. => Label comparisons often need to inspect encoding bits at the far right.

• MS SQL Server employs further path encodings organized in reverse (node-to-root) order.

• Note: - Preorder ranks fit into CPU registers.
  - 4 byte pre's sufficient for $2^{32} = 4G$ nodes (11 GB XMark fits easily).
  - 8 byte pre's sufficient for $2^{64}$ nodes, i.e., “the universe”...
Topics

- Other approaches & techniques (*selection, far from complete*)
  - Document storage / tree encoding:
    - ORDPATH
    - DataGuides
  - XPath processing:
    - Tree patterns, holistic twig joins
DataGuides

- XPath Accelerator, ORDPATH & similar encoding schemes
  - encode the document's tree structure in the node ranks/labels they assign

DataGuides

- Developed in the context of Lore project (DBMS for semi-structured data)
  - Stanford University, Goldman & Widom, VLDB 1997
- encode the document's tree structure in relation names
- Observation:
  - Each node is uniquely identified by its path from the root
  - Paths of siblings with equal tag names can be unified,
  - Provided we keep their relative order (rank) explicitly
Definition

given a semistructured data instance DB, a *DataGuide* for DB is a graph G s.t.:

- every path in DB also occurs in G
- every path in G occurs in DB
- every path in G is unique
Example:
DataGuides

- Multiple DataGuides for the same data:
Definition

Let \( p, p' \) be two path expressions and \( G \) a graph; we define
\[
\equiv^G_{\text{p}} \quad \text{if} \quad p(G) = p'(G)
\]
i.e., \( p \) and \( p' \) are indistinguishable on \( G \).

Definition

\( G \) is a **strong** dataguide for a database \( DB \) if \( \equiv^G \equiv^DB \) is the same as
\[
\equiv^DB
\]

Example:

- \( G1 \) is a strong dataguide
- \( G2 \) is not strong

\[
\begin{align*}
\text{person.project} & \not\equiv^DB \text{dept.project} \\
\text{person.project} & \equiv^G_{\text{G1}} \text{dept.project} \\
\text{person.project} & \equiv^G_{\text{G2}} \text{dept.project}
\end{align*}
\]
Constructing the strong DataGuide $G$:

Nodes($G$)=$\{\text{root}\}$

Edges($G$)=$\emptyset$

while changes do

choose $s$ in Nodes($G$), $a$ in Labels

add $s'$=$\{y|\exists x \in s, (x \rightarrow a \rightarrow y) \in \text{Edges(DB)}\}$ to Nodes($G$)

add $(x \rightarrow a \rightarrow y)$ to Edges($G$)

• Use hash table for Nodes($G$)
• This is precisely the powerset automaton construction.
Monet XML approach

- Early attempt to store and query XML data in MonetDB
- By Albrecht Schmidt
- Not related to Pathfinder & MonetDB/XQuery
Monet XML approach

**Definition 1.** An XML document is a rooted tree \(d = (V, E, r, \text{label}_E, \text{label}_A, \text{rank})\) with nodes \(V\) and edges \(E \subseteq V \times V\) and a distinguished node \(r \in V\), the root node. The function \(\text{label}_E : V \rightarrow \text{string}\) assigns labels to nodes, i.e., elements; \(\text{label}_A : V \rightarrow \text{string} \rightarrow \text{string}\) assigns pairs of strings, attributes and their values, to nodes. Character Data (CDATA) are modeled as a special ‘string’ attribute of data nodes, \(\text{rank} : V \rightarrow \text{int}\) establishes a ranking to allow for an order among nodes with the same parent node. For elements without any attributes \(\text{label}_A\) maps to the empty set.

**Definition 2.** A pair \((o, \cdot) \in \text{oid} \times (\text{oid} \cup \text{int} \cup \text{string})\) is called an association.

**Definition 3.** For a node \(o\) in the syntax tree, we denote the sequence of labels along the path (vertex and edge labels) from the root to \(o\) with \(\text{path}(o)\).
Monet XML approach

Definition 4. Given an XML document \(d\), the Monet transform is a quadruple \(M_t(d) = (r, R, A, T)\) where:

\(R\) is the set of binary relations that contain all associations between nodes;

\(A\) is the set of binary relations that contain all associations between nodes and their attribute values, including character data;

\(T\) is the set of binary relations that contain all pairs of nodes and their rank;

\(r\) remains the root of the document.
Monet XML approach

\[
\begin{align*}
\text{bibliography, } o_1 & \leftarrow \text{key} \rightarrow \text{article, } o_2 \\
& \downarrow \text{key} \rightarrow \text{article, } o_7 \\
& \downarrow \text{key} \rightarrow \text{article, } o_7 \\
\end{align*}
\]

\[
\begin{align*}
\text{author, } o_3 & \quad \text{title, } o_5 \\
\text{editor, } o_8 & \quad \text{author, } o_{10} \\
\text{author, } o_{12} & \quad \text{title, } o_{14} \\
\text{cdata, } o_4 & \quad \text{cdata, } o_6 \\
\text{cdata, } o_9 & \quad \text{cdata, } o_{11} \\
\text{cdata, } o_{13} & \quad \text{cdata, } o_{15} \\
\text{string} & \quad \text{string} \\
\text{string} & \quad \text{string} \\
\text{string} & \quad \text{string} \\
\end{align*}
\]

\[
\begin{align*}
\text{"BB88"} & \leftarrow \text{key} \rightarrow \text{article, } o_2 \\
\text{"How to Hack"} & \leftarrow \text{key} \rightarrow \text{article, } o_7 \\
\text{"Bob Byte"} & \leftarrow \text{key} \rightarrow \text{article, } o_7 \\
\text{"Ken Key"} & \leftarrow \text{key} \rightarrow \text{article, } o_7 \\
\text{"Hacking & RSI"} & \leftarrow \text{key} \rightarrow \text{article, } o_7 \\
\end{align*}
\]

\[
\begin{align*}
\text{bibliography} \xrightarrow{\varepsilon} \text{article} \rightarrow \{\langle o_2, o_3 \rangle, \langle o_4, o_6 \rangle\}, \\
\text{bibliography} \xrightarrow{\varepsilon} \text{article} \xrightarrow{\varepsilon} \text{author} \rightarrow \{\langle o_2, o_3 \rangle, \langle o_7, o_{10} \rangle, \langle o_7, o_{12} \rangle\}, \\
\text{bibliography} \xrightarrow{\varepsilon} \text{article} \xrightarrow{\varepsilon} \text{author} \xrightarrow{\varepsilon} \text{cdata} \rightarrow \{\langle o_3, o_4 \rangle, \langle o_{10}, o_{11} \rangle, \langle o_{12}, o_{13} \rangle\}, \\
\text{bibliography} \xrightarrow{\varepsilon} \text{article} \xrightarrow{\varepsilon} \text{author} \xrightarrow{\varepsilon} \text{cdata} \xrightarrow{\varepsilon} \text{string} \rightarrow \{\langle o_4, \text{"Ben Bit"} \rangle, \langle o_{11}, \text{"Bob Byte"} \rangle, \langle o_{13}, \text{"Ken Key"} \rangle\}, \\
\text{bibliography} \xrightarrow{\varepsilon} \text{article} \xrightarrow{\varepsilon} \text{title} \rightarrow \{\langle o_2, o_5 \rangle, \langle o_7, o_{14} \rangle\}, \\
\text{bibliography} \xrightarrow{\varepsilon} \text{article} \xrightarrow{\varepsilon} \text{title} \xrightarrow{\varepsilon} \text{cdata} \rightarrow \{\langle o_5, o_6 \rangle, \langle o_{14}, o_{15} \rangle\}, \\
\text{bibliography} \xrightarrow{\varepsilon} \text{article} \xrightarrow{\varepsilon} \text{title} \xrightarrow{\varepsilon} \text{cdata} \xrightarrow{\varepsilon} \text{string} \rightarrow \{\langle o_6, \text{"How to Hack"} \rangle, \langle o_{15}, \text{"Hacking & RSI"} \rangle\}, \\
\text{bibliography} \xrightarrow{\varepsilon} \text{article} \xrightarrow{\varepsilon} \text{editor} \rightarrow \{\langle o_7, o_{10} \rangle\}, \\
\text{bibliography} \xrightarrow{\varepsilon} \text{article} \xrightarrow{\varepsilon} \text{editor} \xrightarrow{\varepsilon} \text{cdata} \rightarrow \{\langle o_8, o_9 \rangle\}, \\
\text{bibliography} \xrightarrow{\varepsilon} \text{article} \xrightarrow{\varepsilon} \text{editor} \xrightarrow{\varepsilon} \text{cdata} \xrightarrow{\varepsilon} \text{string} \rightarrow \{\langle o_9, \text{"Ed Itor"} \rangle\}, \\
\text{bibliography} \xrightarrow{\varepsilon} \text{article} \xrightarrow{\varepsilon} \text{key} \rightarrow \{\langle o_2, \text{"BB88"} \rangle, \langle o_7, \text{"BK99"} \rangle\}.
\end{align*}
\]
Monet XML approach

- Early attempt to store and query XML data in MonetDB
- By Albrecht Schmidt
- Not related to Pathfinder & MonetDB/XQuery
- No XQuery compiler
  - XMark queries are hand-crafted and -optimized in MIL
- Child, Descendant, Parent & Ancestor steps become regular expressions on the relation names (i.e., catalog)
- Open: preceeding & following steps?
Topics

- Other approaches & techniques (*selection, far from complete!*)
  - Document storage / tree encoding:
    - ORDPATH
    - DataGuides
  - XPath processing:
    - Tree patterns, holistic twig joins
Twig Join Algorithms

So far: interpreted XPath expressions in an **imperative** manner

- Evaluated XPath expressions **step-by-step**, as stated in the query
- Given /α₁::ν₁/α₂::ν₂/…/αₙ::νₙ,
- we first evaluated /, then XPath step α₁::ν₁, then step α₂::ν₂, ...

This may not always be the best choice:

- **Intermediate results** can get very large, even if the final result is small:

```
  a
 /|
/ | \
 b   ... b
 / |
/  | \
 c   ... c
```

▷ /a/b/d produces many intermediate b nodes, but only a single result node.

Database context => think in a **declarative** manner

- DBMS optimizer / engine can evaluate query in “best” order
Tree Patterns

• In fact, XPath is a **declarative language**.

  > /descendant::*:timeline/child::*:event

  “Find all nodes $v_1$, $v_2$, and $v_3$, such that

  $v_1$ is a document root,

  $v_2$ is a descendant element of $v_1$ and is named timeline, and

  $v_3$ is a child element of $v_2$ and named event.

  All nodes of type $v_3$ form the query result.

• Observe the combination of

  (a) predicates **on single nodes**, and

  (b) structural conditions **between these nodes**.
Tree Patterns

- Structural conditions: Intuitively expressed as tree patterns:
  - $p_1$
    - Nodes labeled with node predicates
  - $p_2$
    - Structural conditions:
      - **Double line:** ancestor/descendant relationships
      - **Single line:** parent/child relationships
  - $p_3$

- Arbitrary predicates are allowed, but typical are predicate on tag names:
  - /$p$
    - Nodes labeled with requested tag name
  - $p$
    - Document root: label /
      - If not /-node specified:
        - search for pattern anywhere in the document
Tree Patterns

- Given such a tree pattern, ‘query evaluation’ means
  
  “Find all bindings of nodes in the document to nodes in the tree pattern, such that all structural and node constraints are fulfilled.”

- Compare this to the tuple relational calculus:

  \[ \{ t \mid \exists r, \exists s : R(r) \land S(s) \land r[a] = s[a] \land t[a] = r[a] \land t[b] = s[b] \} \]

  We search for bindings for \( r \) and \( s \) that satisfy the given predicate.

- We have not, however, specified which of the pattern nodes to be the **query result**.

  - Either return **tuples** of nodes, as binding to all the pattern nodes,
  
  - or **mark** a specific node in the query as the result node.

  - What is the XPath query for the tree pattern on the right?
Tree Patterns

- Not limited to **path patterns**
- May also be **twig patterns**
- Mapping between tree patterns and XPath is in general not trivial
- Examples:

```
  a
 / \
 b   d
  |   |   \
 c   e   g
    |   |
    h   i
    |   |
    j   |
```
PathStack Algorithm


- Answer queries for path patterns.

- Idea:
  - Path patterns contain the forward axes child and descendant only.
  - To evaluate forward axes, it is sufficient to scan forward in preorder only.
  - Can we evaluate path queries in a single document scan?
PathStack Algorithm: Path Patterns

- During a sequential table read, maintain the path from the root to the current node with the help of a stack:

  For each node $n$
  - Remove all nodes $v$ from the stack that are not ancestors of $n$ ($v.post < n.post$).
  - Push $n$ onto the stack.

  (This is similar to the stack we used to generate the pre/post encoding.)

- For any node check if we can match the stack against the query pattern.

  - **Example:** Stack
  - For descendant axes, we allow gaps for the match.

  ➔ We **can** find path patterns in a single sequential read.

```plaintext
/  /  
|  |  
|  |  
|  |  

/  /  
|  |  
|  |  
|  |  

stack  pattern
```
PathStack Algorithm: Path Patterns

- The task is now to match the ancestor stack against the query pattern.
  - This requires **regular expression** matching.
  - Matching has to be triggered for each document node.
  - Regular expression matching is **expensive**.

- It is not sufficient to find **some** match, we need to find **all** query results.
  - There may be multiple matches on the same stack.
    (E.g., if the same tag name appears more than once on the stack.)

- Although we meet the **single scan** constraint, path evaluation is **tedious**.

**Idea:**

- While scanning, only put **interesting** nodes on the stack.
- Add some more **structural information** to the stack.
PathStack Algorithm: Path Patterns

① Test the predicates before pushing nodes on the stack.
   ▶ Save work when evaluating the stack.

② Keep separate stacks for each node in the query pattern.
   ▶ We know which predicate each node belongs to afterwards.
   ▶ Each of the stacks contains the ancestor/descendant relationship of nodes satisfying the same predicate.

③ Link nodes in different stacks to represent their ancestor/descendant relationship.
   ▶ Recover the information we lost in ②.
PathStack Algorithm: Path Patterns

- When a node is pushed onto the stack $S_i$, it is linked to the current top of $S_{i-1}$.
  - The pointer starting from node $v$ always points to an ancestor of $v$.

- We insert a node into Stack $S_i$ only if
  - the parent stack $S_{i-1}$ is not empty, or
  - $S_i$ is the stack of the query root, i.e. $i = 0$.

- Nodes within one stack are always in ancestor/descendant relationship.
  - From stack-bottom to top, all nodes are on a root-to-leaf path in the XML tree.

- For descendant-only patterns we have found an answer, as soon as there is a node in the leaf stack.
  - The child relationship has to be checked separately.

- The tree of stacks encodes all (partial) answers to the query pattern.
  - We will shortly see how to retrieve them.
PathStack Algorithm: Path Patterns

Example:

```
/  
|___ timeline
    |___ event

all stacks initially empty
```

```
/  
|___ S_0

document root visited
```

```
/  
|___ S_0

first timeline node visited
```

```
/  
|___ S_0

first event node visited
```

```
/  
|___ S_0

second timeline node visited
```

...
**PathStack Algorithm: Path Patterns**

Example: Recursive XML

- Document: `<a1> <b1> <a2> <b2> <c1> </c1> </b2> <c2> </c2> </a2> </b1> </a1>`
- Query: `a b c`

PathStack Algorithm: Path Patterns

- Stacks initially empty: `S0`
- `a1` visited: `S1`
- `b1` visited: `S1`
- `a2` visited: `S2`
- `b2` visited: `S2`
- `c1` visited: `S2`
- `c2` visited: `S2`
PathStack Algorithm: Path Patterns

- For each tuple $t$ in the document relation, the PathStack algorithm performs three steps:

  1. Clean stacks.
     - Remove all nodes in all stacks that precede the current node $t$.
       $(v \in t/\text{preceeding} \Rightarrow v.pre < t.pre \land v.post < t.post)$

  2. Push $t$ on the appropriate stack.
     - Push if $t$ matches a predicate in $q$.
     - Only push if $t$ matches the query root, or the parent stack is not empty.

  3. If $t$ matches the query leaf, output all solutions.
     - We are then sure to find a path from the root to $t$ that contains a match for each query predicate.

- If overlapping predicates are required, i.e., a node can satisfy more than one of the predicates, the algorithm needs to be rewritten slightly.
PathStack Algorithm: Path Patterns

Function PathStack (q : query pattern, doc : table (pre, post))

foreach t ∈ doc in pre-order do
    foreach n_i ∈ q do
        while ¬ empty(S_i) ∧ S_i.top().post < t.post do
            S_i.pop(); /* clean stacks */

    if t matches a predicate p_i in q then
        if i = 0 then
            S_0.push(t, nil); /* deal with query root node */
        else if ¬ empty(S_{i-1}) then
            S_i.push(t, stack position of S_{i-1}.top());
        if q_i is a leaf in the query pattern and t has been pushed onto a stack then
            showSolutions(i, stack position of S_i.top());
            S_i.pop();
PathStack Algorithm: Path Patterns

Back-tracing the solutions

- We are now left with the output of the actual query solution.
- Without the request for a specific binding in the query pattern, we return all bindings to all query nodes.

Idea:

- From each node $v$ in each stack $S_i$, we find its ancestors
  - below $v$ in stack $S_i$, and
  - in stack $S_{i-1}$, if we follow the parent pointer of $v$.
- We find all solutions by following all these ancestors until the root stack.
PathStack Algorithm: Path Patterns

Example: Recursive XML document

```
<a1>
  <b1>
    <a2>
      <b2>
        <c1> </c1>
      </b2>
      <c2> </c2>
    </a2>
  </b1>
</a1>
```

Document

```
a
  b
  c
```

Query

```
a1
  a2

b1
  b2

(c1)
```

```
(a2, b2, c1)
```

```
a1
  a2

b1
  b2

(c1)
```

```
(a1, b2, c1)
```

```
a1
  a2

b1
  b2

(c1)
```

```
(a1, b1, c1)
```
PathStack Algorithm: Path Patterns

Function showSolutions(stackno : int, slotno : int)

positions[stackno] ← slotno;
if stackno = 0 then
  output \( S_0[positions[0]], \ldots, S_{n-1}[positions[n-1]] \);
else
  foreach \( j < S_{stackno[slotno]}.parent \) do
    showSolutions(stackno - 1, j);

- \( n \) is the number of nodes in the query pattern.
- \( positions \) is an array of length \( n \) that holds the current position within all stacks traversed so far.
- We assume that we can reach an entry within a stack by an index, starting from 0.
- If we reach the query root stack \( S_0 \), we output the node in each stack we traversed to reach the root stack.
- Otherwise we follow the parent pointer (the parent field is the index within the parent stack) and recurse for that parent and all its ancestors in the parent stack.
PathStack Algorithm: Path Patterns

- `showSolutions()` returns all query answers for `descendant-only` queries.

- To support the `child` axis, we additionally need to test the `level` properties.

- How can we rewrite `showSolutions()` to support the child axis?
The showSolutions() algorithm with support for the child axis:

```plaintext
Function showSolutions(stackno : int, slotno : int)

positions[stackno] ← slotno;
if stackno = 0 then
    output (S_0[positions[0]], ..., S_{n-1}[positions[n-1]]);
else
    if stackno - 1 → stackno is a descendant axis then
        foreach j < S_{stackno}[slotno].parent do
            showSolutions(stackno - 1, j);
    else
        foreach j < S_{stackno}[slotno].parent do
            if S_{stackno-1}[j].level = S_{stackno}[slotno].level - 1 then
                showSolutions(stackno - 1, j);
```

PathStack Algorithm: Path Patterns
PathStack Algorithm: Path Patterns

- showSolutions() returns nodes in **leaf-to-root order**.
  - If another order is desired, we need to **block** processing.

- No duplicate elimination is performed.
  - If we **remove** each leaf node from the stack, as soon as its results are returned, we can avoid duplicates with respect to **all bindings**.
  - If only some bindings are requested, explicit **duplicate elimination** must be performed.

- PathStack does evaluate any **path pattern** in a single sequential read.
  - We touch at most |document| nodes.
  - Sequential access is (again) cache efficient.
PathStack Algorithm: Twig Patterns

So far we only considered path patterns

Can we extend our ideas for efficient twig pattern evaluation?

Idea:

Decompose twig patterns into multiple path patterns.

All path patterns start from the same root.

Use PathStack for each of them and merge their results.
PathStack Algorithm: Twig Patterns

- **Example**: Decompose twig pattern into path patterns

Original twig query $q_0$:

```
q0
  a
  b
  c  d
    e
```

Decompose twig pattern into path patterns.
PathStack Algorithm: Twig Patterns

- **Example:** Decompose twig pattern into path patterns

Original twig query $q_0$:

```
q0
  \|-- a
  \   \|-- b
  \     \|-- c
  \       \|-- d
  \         \|-- e
```

Split into path patterns $q_1$ and $q_2$:

```
q1
  \|-- a
  \   \|-- b
  \     \|-- c
  \       \|-- d
  \         \|-- e
```

```
q2
  \|-- a
  \   \|-- b
  \     \|-- c
  \       \|-- d
  \         \|-- e
```
PathStack Algorithm: Twig Patterns

- We’re now back at our original problem:
  - To evaluate twig patterns, we first produce **intermediate results**.
  - These intermediate results may get **huge**, even if the final result is **small**.

- Can we **avoid** some of the intermediate results that won’t contribute anyway?

- **Idea:**
  - Before pushing a node onto a stack, **peek** at each descendant tuple stream.
  - Only push a node, if we can find nodes in the stream heads that allow the creation of **at least** one twig solution.

- This way the **TwigStack** algorithm **skips** irrelevant intermediate results.
  - The stream processing model allows this “peeking forward”.
  - For the sequential document read, we need to **materialize** intermediate results.
PathStack Algorithm: Twig Patterns

PathStack performance

- The graphic shows the performance of PathStack, compared to a simple evaluation strategy, similar to a nested loop ("PathMPMJ").
- The time needed for a sequential read of the data is labeled "SS".
Summary (1/5)

- XML
  - Document markup
  - Data exchange
  - Semi-structured
  - Tree model
  - DTDs
  - XML Schema

- XPath
  - Navigation, location steps, axes, node tests, predicates, functions

- XQuery
  - Sequences & Iterations (FLWoR expressions)
Summary (2/5)

- XML Data Management
  - XML file processors
  - XML databases
  - XML integration platforms
  - RDBMS with XML functionality, SQL/XML
  - Relational XML storage: schema-based vs. schema-oblivious
Summary (3/5)

- Purely Relational XML/XQuery processing: MonetDB/XQuery
  - Document encoding: XPath Accelerator (pre/post plane)
  - XPath navigation: Staircase Join
  - XQuery to Relational Algebra translation
    - Item- & Sequence-representation
    - Iterations: Loop-lifting
    - Loop-lifted staircase join
    - Peephole Optimization
    - Order-awareness, sort avoidance
  - XML/XQuery Update Support
Summary (4/5)

• Other approaches & techniques
  • Document storage/encoding:
    • ORDPATH
    • DataGuides
  • XPath processing:
    • Tree patterns, holistic twig joins
Summary (5/5)

• Literature
  • Slides
  • Literature references in slides
  • Literature references on website:

• Tentamen / Exam:
  • Monday December 14 2009
  • 14:00 – 16:00
  • Zaal / Room: REC-P 0.14
Projects: Join the MonetDB Team!

• Own ideas, suggestions, initiative welcome!

• Master Student Projects (6 Months)
  • Various projects, each consisting of both research & implementation
  • See monetdb.cwi.nl/Development/Research/Projects/ for a sample list
  • Feel free to come with your own idea(s)!

• Implementation Projects
  • Both short-term & long-term
  • E.g. open feature requests: sf.net/tracker/?group_id=56967
  • Become owner/maintainer of some (new) part of MonetDB
  • We are (desperately) looking for Windows SW-development & system experts!
We Offer...

• 24x7x365 support & advice
• Membership in a kind & friendly Family-Team of Experts
• Chance to participate in & contribute to a large & successful open-source research project
• Lots of experiences, exiting research & fun
• Desk & workstation at CWI
  • Fridge, micro-wave, free coffee, free soup, free cake (occasionally)
  • Master Students only (possibly part-time)
  • Limited availability => FCFS!
• Some pocket money (stage vergoeding)
  • Master Students only
  • Limited availability => FCFS!
• ...

Stefan.Manegold@CWI.nl

Other Xquery Processing Approaches

ADT 2009
Exam / Tentamen

Monday December 14 2009
14:00 – 16:00
REC-P 0.14