### XPath Query Evaluation: Improving Time and Space Efficiency

Georg Gottlob, Christoph Koch and Reinhard Pichler
Technische Universität Wien, Austria

* US Patent Pending. Work supported by Austrian Science Fund (Pj.-Nr. Z29-INF)

### Motivation
- XPath used in many XML-related standards (XSLT, XPointer, XQuery, ...).
- Current systems: evaluation of XPath 1.0 in exponential time
- Feasible in polynomial time: VLDB’02
- Improvements: ICDE’03

### Overview of this Talk
- XPath Evaluation: Running Example
- Polynomial-Time XPath Evaluation via the “Context-value Table Principle”
- Improvements
- Further Improvements for fragments of XPath

### XPath Evaluation: Running Example

#### Document and Query
Sample document $D$:
```
<a> <b/> <c/> <b/> <c/> </a>
```

Sample query $Q$:
```
child::b/following::*[position() != last() and self::b]
```

#### XPath Evaluation Problem
Example:
Evaluate the XPath query $Q$ on the XML document $D$ for the context node $a$. 

Parse Tree of the Query

Query: child::b/following::*[position() != last() and self::b]

Query Tree:
- N1: child::b/N2
- N2: following::*[N1]
- N3: N4 and N5
- N4: position() != N7
- N5: boolean(N8)
- N6: last()
- N7: self::b

Straightforward XPath Evaluation

evaluation of child::b/N2 for cn = a
(with N2 = following::*[position() != last() and self::b])
intermediate result (for child::b): {b1, b2}
final result: recursive evaluation of N2 for cn = b1
recursive evaluation of N2 for cn = b2

evaluation of following::*[N3] for cn = b1
(with N3 = position() != last() and self::b)
intermediate result (for following::*): {c1, b2, c2}
final result: recursive evaluation of N3 for cn = b2, cp = 2, cs = 3
recursive evaluation of N3 for cn = c2, cp = 3, cs = 3
only the evaluation of N3 for cn = b2, cp = 2, cs = 3 yields “true”

Intermediate Result: [b2]

only the evaluation of N3 for cn = b2, cp = 2, cs = 3 yields “true”
=> result of following::*[N3] for cn = b1: [b2]
evaluation of following::*[N3] for cn = b1
(with N3 = position() != last() and self::b)

intermediate result (for following::*):
{c1, b2, c2}

final result: recursive evaluation of N3 for cn = c1, cp = 1, cs = 3
recursive evaluation of N3 for cn = b2, cp = 2, cs = 3
recursive evaluation of N3 for cn = c2, cp = 3, cs = 3

only the evaluation of N3 for cn = b2, cp = 2, cs = 3 yields “true”

=> result of following::*[N3] for cn = b1:
{b2}

=> overall result of Q = child::b/following::*[N3] for cn = a:
{b2}

Summary

Time complexity \( T(|D|, |Q|) \) of this algorithm:

\[
T(|D|, |Q|) = |D| \cdot T(|D|, |Q| - 1)
\]

\[
T(|D|, |Q|) = O(|D|^{|Q|})
\]

Observation

• CURRENT XPATH PROCESSORS REQUIRE TIME EXPONENTIAL IN THE SIZE OF THE XPATH QUERY!!!

• Reason: Evaluation of an XPath query Q via recursive calls of the evaluation procedure.

Polynomial-Time XPath Evaluation via Context-Value Tables (CVTs)
Documents and Contexts

Sample document $D$:

```
<q> <h> <c> <h> <c> </h> </c> </h> </q>
```

Contexts:

\[ C \subseteq \{ (cn, cp, cs) \mid cn \in \{ a, b_1, c_1, b_2, c_2 \}, 1 \leq cp \leq cs \leq 5 \} \]

Context-value Tables (CVT)

- Given an XPath expression $e$ and a context $c$, the result of evaluating $e$ on the document $D$ is uniquely determined.
- Four types of values ($nset, num, str, bool$)
- The CVT of $e$ is a relation

\[ R \subseteq C \times \{ nset \cup num \cup str \cup bool \} \]

(In fact, this is only a relevant subset of the full tables.)
null
Context-Value Table Principle

if CVT for each operation $Op(e_1, ..., e_n)$ can be computed in polynomial time given the CVTs for sub-expressions $e_1, ..., e_n$,

then CVT of overall query can be computed (bottom-up) in polynomial time.

Efficiency of the CVT-Algorithm

Space Complexity: $O(|D|^t |Q|^t)$
- number of tables: $O(|Q|)$
- number of rows per table: $O(|D|^t)$
- size of each row: $O(|D|^t |Q|)$

Time Complexity: $O(|D|^t |Q|^t)$
- time to compute each row: $O(|D|^t |Q|)$

Improvements

Basic Ideas:

1. Top-down Evaluation (of location paths):
2. Restriction to the relevant context
3. Treating “outermost” location paths as sets
4. Treating context-position/size in a loop

Basic Ideas:

1. Top-down Evaluation (of location paths):
2. Restriction to the relevant context
3. Treating “outermost” location paths as sets
4. Treating context-position/size in a loop
### Diagram and Table Representation

#### Diagram 1
- **Node a**: A root node with children b1, c1, b2, c2.
- **Node b**: A node with children b1, c1, b2, c2.
- **Node c**: A node with children b1, c1, b2, c2.
- **Node d**: A node with children b1, c1, b2, c2.

#### Table 1

<table>
<thead>
<tr>
<th>Node</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Diagram 2
- **Node e**: A root node with children b1, c1, b2, c2.
- **Node f**: A node with children b1, c1, b2, c2.
- **Node g**: A node with children b1, c1, b2, c2.
- **Node h**: A node with children b1, c1, b2, c2.

#### Table 2

<table>
<thead>
<tr>
<th>Node</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Diagram 3
- **Node i**: A root node with children b1, c1, b2, c2.
- **Node j**: A node with children b1, c1, b2, c2.
- **Node k**: A node with children b1, c1, b2, c2.
- **Node l**: A node with children b1, c1, b2, c2.

#### Table 3

<table>
<thead>
<tr>
<th>Node</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Diagram 4
- **Node m**: A root node with children b1, c1, b2, c2.
- **Node n**: A node with children b1, c1, b2, c2.
- **Node o**: A node with children b1, c1, b2, c2.
- **Node p**: A node with children b1, c1, b2, c2.

#### Table 4

<table>
<thead>
<tr>
<th>Node</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Basic Ideas:

1. Top-down Evaluation (of location paths):
2. Restriction to the relevant context
3. Treating “outermost” location paths as sets
4. Treating context-position/size in a loop
Basic Ideas:

1. Top-down Evaluation (of location paths):
2. Restriction to the relevant context
3. Treating “outermost” location paths as sets
4. Treating context-position/size in a loop

Observation

• The CVTs for these location steps have size $O(D^i)$ (i.e., set of nodes reachable from each $cn$)

• Sufficient: compute for each $i$ the set of nodes reachable from the input $cn$ via the first $i$ location steps

=> CVTs are of size $O(D^i)$

Basic Ideas:

1. Top-down Evaluation (of location paths):
2. Restriction to the relevant context
3. Treating “outermost” location paths as sets
4. Treating context-position/size in a loop

Observation

• Principal feature of the CVT method: simultaneous evaluation of every subexpression of $Q$ for every possible/relevant context tuple in a single CVT

• Sufficient (for polynomial time): relevant context $cn$: simultaneous evaluation otherwise: evaluation in a loop over $cp / cs$

=> maximum table size: $O(|D|^i*|Q|)$
Improved Complexity Bounds

**Time Complexity:** $O(|D|^2 * |Q|)$
- Number of tables: $O(|Q|)$
- Number of rows per table: $O(|D|^2)$
- Time to compute a row: $O(|D|^2 * |Q|)$

**Space Complexity:** $O(|D|^2 * |Q|)$
- (For each CVT)
- Information held simultaneously: $O(|D|^2 * |Q|)$

Further Improvements for Fragments of XPath

Definition of a Linear Time Fragment

Core XPath:
Location paths and conditions consisting of boolean combinations of path conditions.

Example:
```
//descendant::a/parent::b[child::c/child::d or not(following::*)]
```

Definition of a Linear Space Fragment

Goal: Store each CVT in space $O(|D|^2 * |Q|)$

Strategy (“Extended Wadler Fragment”)
- Restrictions to ensure that scalar values fit into constant space
- Restrictions / Evaluation strategy: location paths inside predicates must not be computed explicitly

Conclusion

- Main-memory XPath processing algorithm with polynomial-time combined complexity.
- Further improvements for full XPath 1.0
- Definition of XPath fragments:
  - Linear time / Linear space
- Prototype implementation available

Further resources at http://www.xmltaskforce.com

- Errata, updates etc.
- Future papers
- Implementation