XML Query Reformulation for XPath, XSLT and XQuery
Sven Groppe
(Sven.Groppe@deri.org, http://members.deri.at/~sveng/)
Tutorial at DBA 2006/Innsbruck

Structure of the Tutorial

- First 90 Minutes
  - Short Introductions
    - XML
    - XSLT including XPath
  - 3 Query Reformulation Methods including Performance Evaluation
- Second 90 Minutes
  - Intersection Test of XPath expressions
  - Reduction of Intersection Test to Satisfiability Test
  - Satisfiability Test without schema information
  - Satisfiability Test with schema information
  - Differences between XQuery and XSLT
  - 2 Caching Strategies for transformed XML data

Motivation – Intersection Test

- Static Analysis of XSLT Stylesheets

- Which templates <xsl:template match=M> could be called from an <xsl:apply-templates select=I/> XSLT instruction?

  - M intersect I ≠ {}
Logical Testers

- Incomplete Logical Testers
  - One direction uncertain
  - In our application scenarios: Only loss of accuracy, no loss of correctness

Intersection Test
- For all XML documents: \( \text{XP}_1 \cap \text{XP}_2 = \emptyset \) or maybe \( \text{XP}_1 \cap \text{XP}_2 \neq \emptyset \)

Satisfiability Test
- For all XML documents: \( \text{XP}_1 = \emptyset \) or maybe \( \text{XP}_1 \neq \emptyset \)

Schema Logical Tests:
- All XML Documents must be valid according to a given Schema
- For whole XPath, the satisfiability test is undecidable (Benedikt, Fan and Geerts, PODS 2005)
  - \( \Rightarrow \) Incomplete Testers are theoretically the best what we can achieve

Reduction of Intersection Test to Satisfiability Test

- Satisfiability test as more "basic" test
  - Less complicated than intersection test

Idea:
- Instead of \( \text{XP}_1 \cap \text{XP}_2 = \emptyset \)
- Test if nodes of \( \text{XP}_1 \) are not matched by \( \text{XP}_2 \)
  - by checking whether "all" is not satisfiable
- I.e. test \( \text{XP}_1[\overline{\text{XP}_2}] = \emptyset \),
  - where \( \overline{\text{XP}_2} \) is an expression (reverse pattern), which returns a non-empty result if the current context node is matched by \( \text{XP}_2 \)

Example:
- \( \text{XP}_1 = \text{idoc}:/\text{child::Maps}/\text{descendant::title} \)
- \( \text{XP}_2 \) is a reverse pattern

Remember: Evaluation of XPath expression \( E \)

- \( E = /\text{child::Maps}/\text{descendant::title} \)
- This node is a result of \( E \)
- This node is a result of \( E \)
- The current node set includes the element node and its children
- Parent node is the node above the current node
- Next sibling is the node after the current node
- Test Node
Reverse Pattern

- Opposite direction to evaluation of E:
  - Maps: descendant: title
  - Evaluation of E: starting at the context node
    - path: self: title

This node is matched by E:

- Reverse Pattern of
  
```
  [self::element()]
  
  ancestor::Maps
  [self::element()]
  
  parent::node()
  [self::node()] is root()
```

Example: reverse pattern of child: a[attribute::b-c] is

```
  self::a[attribute::b-c][self::instance of element()] / parent::node()
```

Reverse Pattern

- Reverse Pattern of
  
```
  [axis: test, F[1], F[2], ... / axis: test, F[1], F[2], ... ]
  
  self::test, F[1], F[2], ... / test, F[1], F[2], ... ]
```

where axis are reverse axes of axis, F[1] an additional test of axis, F[2] is [self::node() is root()] for absolute pattern and F[3] is the empty expression for relative pattern

Patterns containing disjunctions, "or" are first factored out and then we reverse each expression without "or".

Reverse Pattern

<table>
<thead>
<tr>
<th>Axis A</th>
<th>Reverse Axes of A</th>
<th>Additional Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestor</td>
<td>descendant</td>
<td>self instance of element()</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>descendant-or-self, node/attribute</td>
<td>self instance of element()</td>
</tr>
<tr>
<td>attribute</td>
<td>ancestor</td>
<td>self instance of attribute()</td>
</tr>
<tr>
<td>axis</td>
<td>ancestor</td>
<td>self instance of element()</td>
</tr>
<tr>
<td>descendant</td>
<td>ancestor</td>
<td>self instance of element()</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td>ancestor-or-self</td>
<td>self instance of element()</td>
</tr>
<tr>
<td>following</td>
<td>preceding</td>
<td>self instance of element()</td>
</tr>
</tbody>
</table>
Reverse Pattern

<table>
<thead>
<tr>
<th>Axis A</th>
<th>Reverse Axes of A</th>
<th>Additional Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>following-sibling</td>
<td>preceding-sibling</td>
<td></td>
</tr>
<tr>
<td>parent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parent</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Elimination of – operator and Reduction of Subsumption Test

- Elimination of except XPath operator (short: - operator)
  - XP1 except XP2
    - All nodes of XP1, which are not nodes of XP2
    - XP1[not(XP2-1)]

- Reduction of Subsumption Test to Satisfiability Test
  - XP1 ⊆ XP2
  - XP1 except XP2 = {} ⊆ XP1[not(XP2-1)]={}

Case A Case B Case C Case D

Other Application Scenarios for Logical Testers

<table>
<thead>
<tr>
<th>Application Scenario</th>
<th>Used Tester</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Analysis in XSLT</td>
<td>Tester</td>
<td>Used Tester</td>
</tr>
<tr>
<td>Querying a cache</td>
<td>Tester</td>
<td>Description</td>
</tr>
<tr>
<td>Update of the cache</td>
<td>Tester</td>
<td>Description</td>
</tr>
<tr>
<td>Querying published version</td>
<td>Tester</td>
<td>Description</td>
</tr>
<tr>
<td>Optimization of disjunctive queries</td>
<td>Tester</td>
<td>Description</td>
</tr>
<tr>
<td>Access control</td>
<td>Tester</td>
<td>Description</td>
</tr>
<tr>
<td>Optimization of matching relations</td>
<td>Tester</td>
<td>Description</td>
</tr>
</tbody>
</table>
Other Application Scenarios for the Elimination of \( \land \) and – operators

<table>
<thead>
<tr>
<th>Application Scenario</th>
<th>Used Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>( \land )</td>
<td>The except XPath 2.0 operator is used for caching when the cache loads all nodes of a given XPath query ( \land ) except those nodes, which are already in the cache, in order to answer ( \land ).</td>
</tr>
<tr>
<td>Optimization of applying multiple XPath queries ( \land ), ... ( \land )</td>
<td>( \land )</td>
<td>We can optimize the execution time of applying multiple XPath queries ( \land ), ... ( \land ) by using the approach of [TF04].</td>
</tr>
<tr>
<td>Access control</td>
<td>( \land )</td>
<td>We can check access rights of users by using the approach of [BS05].</td>
</tr>
</tbody>
</table>

Satisfiability Tester without using Schema Information

- (Incomplete) Satisfiability Tester
  - \( Q \) reduced to empty expression \( \Rightarrow Q = \{ \} \)
  - \( Q \) not reduced to empty expression \( \Rightarrow \text{maybe } Q \neq \{ \} \)

  The goals of our heuristic method are that
  - sub-expressions are reduced to the empty path, wherever possible.
  - the XPath expression does not contain a reverse axis so that the XPath evaluator processes only forward axes.
  - our approach eliminates the \( \text{not}(\ldots) \) operator, wherever possible.
  - our approach eliminates location steps with a \( \text{self} \) axis, wherever possible, in order to avoid unnecessary location steps.

Example

```
<descendant::Map/child::*[self::title[self instance of element()] and ancestor::Maps[self instance of element()]]
2*[p1[p2]] = p1[p2]
```

A node test \( t_1 \) is more restrictive than a given node test \( t_2 (t_1 = t_2) \)
  - \( t_1 \) is the name node test and \( t_2 \) is *
  - \( t_1 \) is attribute() and \( t_2 \) is *
  - \( t_1 \) is * and \( t_2 \) is element()
  - \( t_1 \) is \( \text{text()}, \text{comment()}, \ldots, \text{processing-instruction()} \) and \( t_2 \) is element()

\( a_1::t_1 F \) if \( t_1 \neq t_2 \)
\( a_1::t_1 F \) if \( t_1 = t_2 \)
\( a_1::t_2 F \) if \( t_2 \neq t_1 \)
\( a_1::t_2 F \) if \( t_2 = t_1 \)
otherwise
Example

/ancestor::Maps[ancestor::node()][self::node() is root()][child::title]

Example

/ancestor::Maps[ancestor::node()][self::node() is root()][child::title]

Example

/ancestor::Maps[ancestor::node()][self::node() is root()][child::title]
Example

```xml
/descendant::node()[self::node() is root()]
/child::Maps[self::node() is root()]/descendant::Map
/child::title | /self::node()[self::node() is root()]
/child::Maps[self::node() is root()]/descendant::Map
/child::title
```

```xml
/p/a1::t1
≡
⊥⊥ ⊥⊥ if a1 ≠ self
```

Speed Up Evaluation by Simplification of XPath Expressions

Saxon

Pi = //keyword(/../keyword)
E7, E15, E16: simplified to empty expression
Sven Groppe

DERI, University of Innsbruck

XML Query Reformulation for XPath, XSLT and XQuery

Saxon Qizx

\[ Si = //\text{keyword}(/self::\text{keyword}) \]

Elimination of many location steps

Speed Up Evaluation by Simplification of XPath Expressions

Saxon

Qizx

\[ Si = //\text{keyword}(/self::\text{keyword}) \]

Elimination of many location steps

Satisfiability Test using schema information

Precondition

- Given XPath query \( Q \) the satisfiability of which we test
- Schema available of the XML structure on which \( Q \) would apply

Motivation

- More "precise" test results
- Example:

\[
/descendant::\text{Map}/child::*[\text{self::title}[\text{self instance of element()}]/ancestor::\text{Maps}[\text{self instance of element()}]/parent::\text{node}()[\text{self::node()} \text{is root()}]]
\]

is maybe satisfiable according to satisfiability test without using schema information

- If \( \text{Map} \) cannot have \text{title} children according to given schema, then satisfiability test with using schema information returns not satisfiable

Approach for Schema Satisfiability Test

1. Rewrite \( Q \) into \( Q' \) according to schema information
   - Search \( Q \) in the schema information
   - If search is not successful, return not satisfiable
   - Otherwise return equivalent query \( Q' \)
   - determined by a pass through the result of the search
   - in order to encode schema information in \( Q' \)

2. Test \( Q' \) for satisfiability without using schema information
Example – Rewriting according to schema

<table>
<thead>
<tr>
<th>Current Entry</th>
<th>Construction Formulas for XP</th>
</tr>
</thead>
</table>
| <xsd:element name="N"> | If(B=forward) 
| | XP\_{\text{new}}/child::N 
| | else XP\_{\text{new}}/parent::N |
| <xsd:attribute name="N"> | If(B=forward) 
| | XP\_{\text{new}}/attribute::N 
| | else XP\_{\text{new}}/parent::N |

**Attached path**

<table>
<thead>
<tr>
<th>XP_{\text{init}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>XP_{\text{new}}/XP_{\text{init}}</td>
</tr>
</tbody>
</table>

**Entry representing an operator or function**

<table>
<thead>
<tr>
<th>op(XP_{\text{path1}}, \ldots, XP_{\text{pathm}})</th>
</tr>
</thead>
</table>

**Attached loop schema paths 1.m**

<table>
<thead>
<tr>
<th>XP_{\text{init}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>XP_{\text{new}}/XP_{\text{init}}</td>
</tr>
</tbody>
</table>

**Otherwise**

| XP_{\text{new}}/XP_{\text{init}} |

---

\[Q = /\text{descendant::Map/child::*}\]

\[<\text{xsd:schema}>\]

\[<\text{xsd:element name="Maps">}\]

\[<\text{xsd:complexType}>\]

\[<\text{xsd:group ref="r1"/>}\]

\[</\text{xsd:complexType}>\]

\[</\text{xsd:element}>\]

\[</\text{xsd:complexType}>\]

\[</\text{xsd:group}>\]

\[</\text{xsd:schema}>\]
XQuery

- functional language, which means that expressions can be nested with full generality
- strongly-typed language in which the operands of various expressions, operators, and functions must conform to the expected types
- XQuery embeds XPath as path language to locate XML nodes in XML structures
- an XPath expression itself is a simple XQuery expression
- XQuery extends the XPath language by
  - constructors for XML structures like elements and attributes,
  - For-Let-Where-Order By-Return (FLWOR) expressions, which can combine and restructure information from XML documents,
  - user-defined functions and many more language elements

Differences between XQuery and XSLT

<table>
<thead>
<tr>
<th>XQuery</th>
<th>XSLT</th>
</tr>
</thead>
<tbody>
<tr>
<td>No template model, instead user-defined functions</td>
<td>Template model</td>
</tr>
<tr>
<td>Parameters of functions are referred to by order</td>
<td>Parameters of templates are referred to by name</td>
</tr>
<tr>
<td>variables can store every type of XML node</td>
<td>variables can only store element nodes</td>
</tr>
<tr>
<td>we can nest expressions with full generality</td>
<td>use of intermediate variables for nested expressions</td>
</tr>
<tr>
<td>many operations (e.g. sorting) do not affect the identity of XML nodes</td>
<td>nodes must be copied in variables for sorting</td>
</tr>
</tbody>
</table>

Example – XQuery and XSLT

XQuery

```xml
<table>
{ For $it in /table/row
  Return
  <address id="{$it/id}"
    firstname="{$it/firstname}"
    lastname="{$it/lastname}"
    streets="{$it/street}"
    city="{$it/city}"
    state="{$it/state}"
    zip="{$it/zip}"/>
}
</table>
```

XSLT

```xml
<xsl:stylesheet>
  <xsl:template match="table">
    <table>
      <xsl:apply-templates/>
    </table>
  </xsl:template>
</xsl:stylesheet>
```
Caching of XSLT transformed XML data

<table>
<thead>
<tr>
<th>Motivation</th>
<th>Optimizations for XML Query Reformulation</th>
<th>Caching of XSLT transformed XML data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform only necessary section of XML data</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reuse of results of already answered queries</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>

Different Cache Strategies

- We assume no updates of original data, but the strategies could be extended to handle updates
- 2 different caching strategies
  - Typical cache
  - Combining results
Typical Cache

- **New Query Q can be directly answered from cache**

Client

Query Q

Answer Q(C_i) if Q \subseteq C_i

Typical Cache

- **New query Q cannot be directly answered from cache**

Client

Query Q

Cache entry (Qi, Ci)

Answer Q(C_i) if Q \subseteq C_i

Typical Cache

- **New query Q cannot be directly answered from cache**

Client

Query Q

Cache entry (Qi, Ci)

Answer Q(C_i) if Q \subseteq C_i

Typical Cache

**Different Cache Strategies**

- 2 different caching strategies
  - **Typical cache**
  - **Combining results**
Combining Results

• New Query $Q$ can be directly answered from cache

Client

\[ \text{Query} \hspace{1cm} Q \hspace{1cm} \text{Answer} \hspace{1cm} (C) \text{ if } Q \subseteq C \]

Cache

$\{Q(C), C\}$

Combing Results

• New query $Q$ cannot be directly answered from cache

Client

\[ \text{Query} \hspace{1cm} Q \hspace{1cm} \text{Cache} \hspace{1cm} (Q_{C}, C) \hspace{1cm} \text{Answer} \hspace{1cm} Q(C) \text{ if } Q \subseteq Q_{C} \]

Server

\[ \text{XML} \hspace{1cm} \text{Data-base} \hspace{1cm} (D) \]

Functionality on Client, Server or Proxy

\[ \text{Enhance XSLT view with numbering scheme} \]

\[ \text{Apply optimized XSLT view} \]

\[ \text{Combine V} \]

\[ \text{New cache} \]

$\{Q(C), Q_{C}, (C)\}$

Numbering Scheme

• Modify XSLT view so that
  • element nodes in the output are annotated by an identifier,
  • the identifier preserves document order even in the whole output of the XSLT view,
  • different output of different optimized XSLT views (of the same XSLT view) can be combined
Numbering Scheme – Example (1/2)

```xml
<xsl:stylesheet>
  <xsl:template match="table">
    <table>
      <xsl:apply-templates/>
    </table>
  </xsl:template>
  <xsl:template match="row">
    <address id="{id}" lastname="{lastname}"/>
  </xsl:template>
</xsl:stylesheet>
```

Numbering Scheme – Example (2/2)

```xml
<xsl:stylesheet>
  <xsl:template match="table">
    <table>
      <xsl:apply-templates/>
    </table>
  </xsl:template>
  <xsl:template match="row">
    <address id="{id}" lastname="{lastname}"/>
  </xsl:template>
</xsl:stylesheet>

Combination of the results - Example

Q1 = table/address[@id='1' or @id='3']/@*
Cache Result after Q2

Q2 = table/address[@id='2']/@*

Source =
<row id="1">
  <id>1</id>
  <lastname>Name1</lastname>
</row>
<row id="2">
  <id>2</id>
  <lastname>Name2</lastname>
</row>
<row id="3">
  <id>3</id>
  <lastname>Name3</lastname>
</row>
Performance Analysis 1/3

- Q1 = /table/address/attribute::firstname[self::node()='James']
- Q2 = /table/address/attribute::firstname[self::node()='Bob']
- Q3 = (/table/address/attribute::firstname[self::node()='James'] | /table/address/attribute::firstname[self::node()='Bob'])

Performance Analysis 2/3

- Q1 = /table/address/attribute::firstname
- Q2 = /table/address/attribute::lastname
- Q3 = Q1, Q4=Q2, ...

Performance Analysis 3/3

- QX = /table/address/attribute::id[self::node() > 'X']
- X=8000, 7000, 6000, 5000, 4000, 3000, 2000, 1000, 900, 800, 700, 600, 500, 400, 300, 200, 100, 90, 80, 70, 60, 50, 40, 30, 20, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
<table>
<thead>
<tr>
<th>Comparing the 2 different Caching Strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typical Cache</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Reuse results if new query is subsumed by previous query</td>
</tr>
<tr>
<td>Reuse results even if new query is not subsumed by previous query (and only intersects previous queries)</td>
</tr>
<tr>
<td>Load only missing section into cache if new query cannot be directly answered from the cache</td>
</tr>
<tr>
<td>No preprocessing step of XSLT view necessary</td>
</tr>
<tr>
<td>No combination of the results necessary</td>
</tr>
</tbody>
</table>

Questions / Remarks?

- Sven Groppe, XML Query Reformulation for XPath, XSLT and XQuery, Sierke-Verlag, Göttingen, 2005. ISBN 3-933893-24-0