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

- Here: **Incomplete** Logical Testers
  - One direction uncertain
  - In our application scenarios: Only loss of accuracy, no loss of correctness
- Intersection Test
  - For all XML documents: **XP1 intersect XP2 = {}** or **maybe XP1 intersect XP2 ≠ {}**
- Satisfiability Test
  - For all XML documents: **XP1 = {}** or **maybe XP1 ≠ {}**
- **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)
  - => Incomplete Testers are theoretically the best what we can achieve

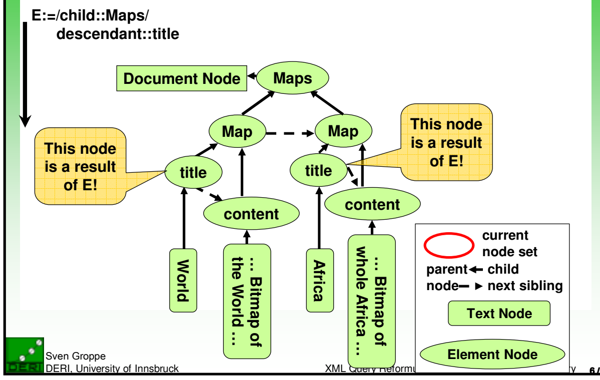
Reduction of Intersection Test to Satisfiability Test

- Satisfiability test as more „basic“ test
  - „less complicated“ than intersection test
- Idea:
  - Instead of **XP1 intersect XP2 = {}**
    - test if nodes of **XP1** are not matched by **XP2**
    - by checking whether „all“ is not satisfiable
  - i.e. test **XP1[XP2^{-1}] = {}**,
    - where **XP2^{-1}** is an expression (*reverse pattern*), which returns a non-empty result if the current context node is matched by **XP2**
- Example:

```
/descendant::Map/child::* intersect
/child::Maps/descendant::title = {}
⇔ /descendant::Map/child::*[self::title[self instance of
element()*]/ancestor::Maps[self instance of
element()*]/parent::node()[self::node() is root()]] = {}
```


Remember: Evaluation of XPath expression E

- Evaluation of E:=/child::Maps/descendant::title



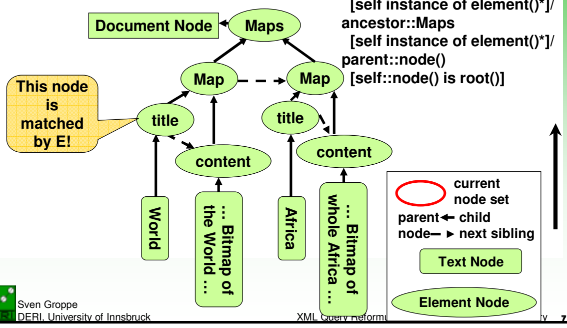
Reverse Pattern

- Opposite direction to evaluation of E:=/Maps/descendant::title

Test if a context node is matched by E

Evaluation of E^{-1} starting at the context node

E^{-1}:=self::title
[self instance of element()*]/
ancestor::Maps
[self instance of element()*]/
parent::node()
[self::node() is root()]



Reverse Pattern

- Reverse Pattern of

[/]**axis**$_1$**::test**$_1$**[F**$_{11}$**]…[F**$_{1n1}$**]/…/axis**$_m$**::test**$_m$**[F**$_{m1}$**]…[F**$_{mn_m}$**]**

is

self::test$_m$**[F**$_{m1}$**]…[F**$_{mn_m}$**]T**$_m$**/**
(raxis$_{m1}$**::test**$_{m-1}$**|…|raxis**$_{mp_m}$**::test**$_{m-1}$**) [F**$_{(m-1)1}$**]…[F**$_{(m-1)n_{m-1}}$**]T**$_{m-1}$**/…/**
(raxis$_{21}$**::test**$_1$**|…|raxis**$_{2p_2}$**::test**$_1$**) [F**$_{11}$**]…[F**$_{1n1}$**]T**$_1$**/**
(raxis$_{11}$**::node()|…|raxis**$_{1p1}$**::node())T**$_{root}$

where **raxis**$_{ij}$ are reverse axes of **axis**$_i$, **T**$_i$ additional test of **axis**$_i$,
T$_{root}$ is **[self::node() is root()]** for absolute pattern and
T$_{root}$ is the empty expression for relative pattern

Patterns containing disjunctions „|“ are first factored out and then we reverse each expression without „|“

Example: reverse pattern of **child::a[attribute::b=„c“]** **is**
self::a[attribute::b=„c“][self instance of element()*]/parent::node()

Reverse Pattern

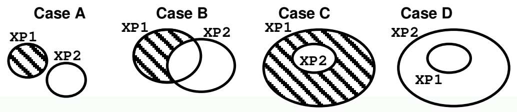
Axis A	Reverse Axes of A	Additional Test
ancestor	1) descendant 2) descendant-or-self::node()/attribute 3) descendant-or-self::node()/namespace	[self **instance of** element()*]
ancestor-or-self	1) descendant-or-self 2) descendant-or-self::node()/attribute 3) descendant-or-self::node()/namespace	
attribute	parent	[self **instance of** attribute()*]
child	parent	[self **instance of** element()*]
descendant	ancestor	[self **instance of** element()*]
descendant-or-self	ancestor-or-self	
following	preceding	[self **instance of** element()*]

Reverse Pattern

Axis A	Reverse Axes of A	Additional Test
following-sibling	preceding-sibling	
namespace	parent	[**not**(self **instance of** element()*) **and** **not**(self **instance of** attribute()*)]
parent	1) child 2) attribute 3) namespace	[self **instance of** element()*]
preceding	following	[self **instance of** element()*]
preceding-sibling	following-sibling	
self	self	

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)]={}**



XP1 except XP2

Other Application Scenarios for Logical Testers

Application Scenario	Used Tester			Description
	?	∩	⊆	
Static Analysis in XSLT Stylesheets		x		Template `<xsl:template match=M>` is called by `<xsl:apply-templates select=S>`, if M ∩ S ≠ ≠ {}.
Querying a cache		x	x	Caches are designed to return answers faster compared to querying the original source so that answer times are reduced whenever results can be retrieved from the cache. Given a query Q and the content of the cache described by a query C. Q can be partially answered by the cache if proving Q∩C≠{} is successful. Q can be fully answered by the cache if proving Q⊆C is successful.
Update of the data in a cache		x		Given an update query U and the content of the cache described by a query C. U describes the part of the original data that has been updated. We can check whether the content of the cache is valid by proving Q∩C={}.
Querying distributed sources		x		Given a query Q and the content of a source described by a query C. We can avoid querying the source if Q∩C={} can be proved, which saves transportation costs and reduces processor load of the processor where the source is located.
Optimization of disjunctive queries			x	We can optimize a disjunctive query Q1|Q2 to Q2 if Q1⊆Q2. We can save processing costs by applying this optimization.
Access control		x		We can check access rights of users by using the approach of [BS05].
Optimization of answering queries	x			Given a query Q. We can avoid querying a source if we can prove Q={}. Particularly in the case of querying views with cost intensive operations, where the output schema of the view is given, we can save processing costs.

Other Application Scenarios for the Elimination of ∩ and – operators

Application Scenario	Used Operation ∩	Used Operation –	Description
Cache		x	The `except` XPath 2.0 operator is used for caching when the cache loads all nodes of a given XPath query `Q` except those nodes, which are already in the cache, in order to answer `Q`.
Optimization of applying multiple XPath queries `XP1`, …, `XPn`	x		We can optimize the execution time of applying multiple XPath queries `XP1`, …, `XPn` by using the approach of [TF04].
Access control	x		We can check access rights of users by using the approach of [BS05].

Satisfiability Tester without using Schema Information

- (Incomplete) Satisfiability Tester
  - `Q` reduced to empty expression => `Q = {}`
  - `Q` not reduced to empty expression => `maybe Q ≠ {}`
- 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 `not(…)` operator, wherever possible.
  - our approach eliminates location steps with a `self` axis, wherever possible, in order to avoid unnecessary location steps.

Example

- `/descendant::Map/child::*[self::title[self instance of element()*]/ancestor::Maps[self instance of element()*]/parent::node()[self::node() is root()]]`

2X `p1[p2/p3] ≡ p1[p2[p3]]`

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

A node test `t1` is *more restrictive* than a given node test `t2` (`t1 « t2`)

- if `t1` is the name node test and `t2` is `*`
- if `t1` is `attribute()` and `t2` is `*`
- if `t1` is `*` and `t2` is `element()`
- if `t1∈{text(),comment(),document-node(),processing-instruction()}` and `t2` is `element()`
- if `t1` is the name node test or `t1∈{*,element(),attribute(),text(),`

`a1::t1[self::t2 F] ≡`	`a1::t1 F`	if t1«t2 or t1=t2
	`a1::t2 F`	if t2«t1
	`a1::t1[self::t2 F]`	if t1 ⊙ t2
	⊥	otherwise

Example

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

```
p/a1::t1[self instance of element()*] ≡
  p/a1::element()                        if element()«t1 or t1=element()
  p[self instance of element()*]/a1::t1  if a1=self
  p/a1::t1       if t1«element(), or a1∈{child, descendant,
                 following, following-sibling}
  ⊥              otherwise
```

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

```
p/child::t1[ancestor::t2] ≡ p[ancestor-or-self::t2]/child::t1
```

(Olteanu et al., XMLDM 2002)

```
/descendant::Map[ancestor-or-self::Maps[self instance of
element()*][parent::node()[self::node() is root()]]]
/child::title
```

```
ancestor-or-self::t1 ≡ ancestor::t1 | self::t1
```

(Olteanu et al., XMLDM 2002)

Example

```
/descendant::Map[ancestor::Maps[self instance of
element()*][parent::node()[self::node() is root()]] |
self::Maps[self instance of element()*][parent::node()
[self::node() is root()]]]/child::title
```

```
                           a1::t1 F              if t1«t2 or t1=t2
a1::t1[self::t2 F] ≡       a1::t2 F              if t2«t1
                           a1::t1[self::t2 F]    if t1⊙t2
                           ⊥                     otherwise
```

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

```
/descendant::t1[ancestor::t2] ≡
/descendant-or-self::t2/descendant::t1
```

(Olteanu et al., XMLDM 2002)

```
/descendant-or-self::Maps[self instance of element()*]
[parent::node()[self::node() is root()]]/descendant::Map
/child::title
```

```
descendant-or-self::t1 ≡ descendant::t1 | self::t1
```

(Olteanu et al., XMLDM 2002)

Example

```
/descendant::Maps[self instance of element()*]
[parent::node()[self::node() is root()]]/descendant::Map
/child::title | /self::Maps[self instance of element()*]
[parent::node()[self::node() is root()]]/descendant::Map
/child::title
```

```
/a1::t1 ≡ / if a1=self and t1=node()
          ⊥ otherwise
```

(Olteanu et al., XMLDM 2002)

```
/descendant::Maps[self instance of element()*]
[parent::node()[self::node() is root()]]/descendant::Map
/child::title
```

```
/descendant::t1[parent::t2] ≡
/descendant-or-self::t2/child::t1
```

(Olteanu et al., XMLDM 2002)

```
/descendant-or-self::node()[self::node() is root()]
/child::Maps[self instance of element()*]/descendant::Map
/child::title
```

```
descendant-or-self::t1 ≡ descendant::t1 | self::t1
```

(Olteanu et al., XMLDM 2002)

Example

```
/descendant::node()[self::node() is root()]
/child::Maps[self instance of element()*]/descendant::Map
/child::title | /self:::node()[self::node() is root()]
/child::Maps[self instance of element()*]/descendant::Map
/child::title
```

p/a1::t1[self::node() is root()] ≡ ⊥ if a1≠self

```
/self:::node()[self::node() is root()]
/child::Maps[self instance of element()*]/descendant::Map
/child::title
```

/self::t1[self::node() is root()] ≡ /self::t1

```
/self:::node()/child::Maps[self instance of element()*]
/descendant::Map/child::title
```

/self::node()/p ≡ /p

```
/child::Maps[self instance of element()*]
/descendant::Map/child::title
```

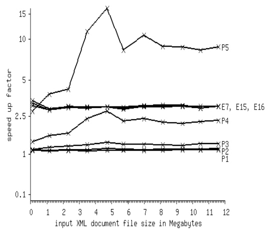

Example

```
/child::Maps[self instance of element()*]
/descendant::Map/child::title
```

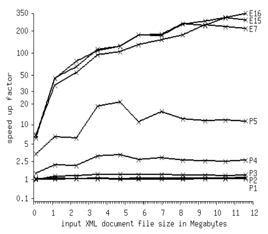
p/a1::t1[self instance of element()*] ≡
- p/a1::element() if element()«t1 or t1=element()
- p[self instance of element()*]/a1::t1 if a1=self
- p/a1::t1 if t1«element(), or a1∈{child, descendant, following, following-sibling}
- ⊥ otherwise

```
/child::Maps/descendant::Map/child::title
```


Speed Up Evaluation by Simplification of XPath Expressions



Saxon

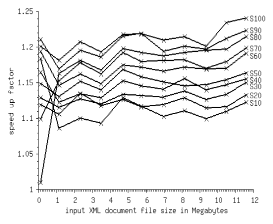


Qizx

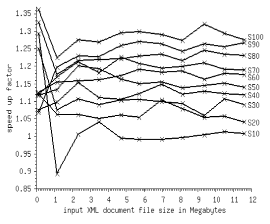
`Pi = //keyword(/../keyword)`^{i}: Elimination of reverse axes

`E7, E15, E16`: simplified to empty expression

Speed Up Evaluation by Simplification of XPath Expressions



Saxon



Qizx

`Si = //keyword(/self::keyword)`^{i}: 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::Map/child::*[self::title[self instance of element()*]/ancestor::Maps[self instance of element()*]/parent::node()[self::node() is root()]]`** **is `maybe satisfiable` according to satisfiability test without using schema information**
    - **If `Map` cannot have `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 - Search

- Q = /descendant::Map/child::*

```
<xsd:schema>
 <xsd:element name="Maps">
  <xsd:complexType>
   <xsd:group ref="r1"/>
  </xsd:complexType>
 </xsd:element>

 <xsd:group name="r1">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
   <xsd:element name="Map">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="title" type="xsd:string"
       minOccurs="0"/>
      <xsd:element name="content" type="xsd:string"
       minOccurs="0"/>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:group>
</xsd:schema>
```


Example – Rewriting according to schema

Current Entry	Construction Formulas for XP
`<xsd:element name=N>`	If (B=forward) XP_{new}=XP_{old}/child::N else XP_{new}=XP_{old}/parent::N
`<xsd:attribute name=N>`	If (B=forward) XP_{new}=XP_{old}/attribute::N else XP_{new}=XP_{old}/parent::N
Attached path	XP_{init}= XP_{new}=XP_{old}[XP_{path}]
Entry representing an operator or function op like or, and and not with m operands	XP_{init}=XP_{old} XP_{new}=op(XP_{path1}, …, XP_{pathm})
Attached loop schema paths 1…m	XP_{init}= XP_{new}=XP_{old}(XP_{path1}|…|XP_{pathm})^n
Otherwise	XP_{new}=XP_{old}

Example – Rewriting according to schema

- Q = /descendant::Map/child::*

XP=/

XP=/Maps

```
<xsd:schema>
 <xsd:element name="Maps">
  <xsd:complexType>
   <xsd:group ref="r1"/>
  </xsd:complexType>
 </xsd:element>

 <xsd:group name="r1">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
   <xsd:element name="Map">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="title" type="xsd:string"
       minOccurs="0"/>
      <xsd:element name="content" type="xsd:string"
       minOccurs="0"/>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:group>
</xsd:schema>
```

XP=/Maps/Map

XP1=/Maps/Map/title

XP2=/Maps/Map/content

Q`=/Maps/Map/title | /Maps/Map/content

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

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

Example – XQuery and XSLT

XQuery

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

XSLT

```
<xsl:stylesheet>

<xsl:template match="table">
  <table>
    <xsl:apply-templates/>
  </table>
</xsl:template>

<xsl:template match="row">
  <address id="{id}"
          firstname="{firstname}"
          lastname="{lastname}"
          street="{street}"
          city="{city}"
          state="{state}"
          zip="{zip}"/>
</xsl:template>
</xsl:stylesheet>
```

Caching of XSLT transformed XML data

- Motivation

	Optimizations for XML Query Reformulation	Caching of XSLT transformed XML data
Transform only necessary section of XML data	✔	✔
Reuse of results of already answered queries	-	✔

Different Cache Strategies

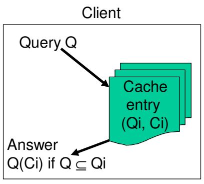
- We assume no updates of original data, but the strategies could be extended to handle updates
- 2 different caching strategies
  - Typical cache
  - Combining results

Different Cache Strategies

- 2 different caching strategies
  - Typical cache
  - Combining results

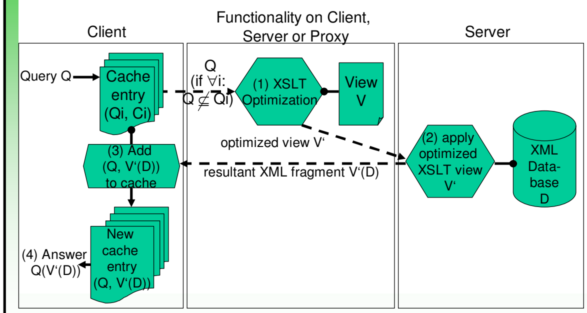
Typical Cache

- New Query Q can be directly answered from cache



Typical Cache

- New query Q cannot be directly answered from cache

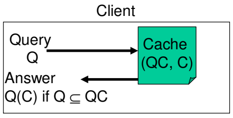


Different Cache Strategies

- 2 different caching strategies
  - Typical cache
  - Combining results

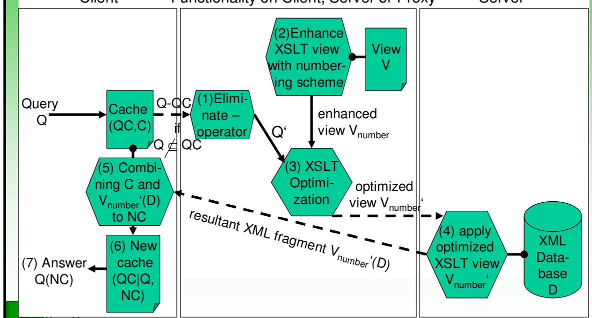
Combining Results

- New Query Q can be directly answered from cache



Combining Results

- New query Q cannot be directly answered from cache



Numbering Scheme

- Modify XSLT view so that
  - element nodes in the output are annoted 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)

```
<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>
```

```
<xsl:stylesheet>
<xsl:template match='/'>
  <xsl:apply-templates  select=‚*'>
    <xsl:with-param
     name='id_param'>1</xsl:with-param>
  </xsl:apply-templates>
</xsl:template>

<xsl:template match="table">
  <xsl:param name='id_param'/>
  <xsl:variable name='id_1'>
    <xsl:value-of select='$id_param'/>.
    <xsl:value-of
      select=‚count(preceding::node())'/>.1
  </xsl:variable>
  <table identifier="{$id_1}">
    <xsl:apply-templates>
      <xsl:with-param name="id_param">
        <xsl:value-of select='$id_1'/>.1
      </xsl:with-param>
    </xsl:apply-templates>
  </table>
</xsl:template>
...
```


Numbering Scheme – Example (2/2)

```
<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>
```

```
...

<xsl:template match="row">
  <xsl:param name='id_param'/>
  <xsl:variable name='id_2'>
    <xsl:value-of select='$id_param'/>.
    <xsl:value-of
      select=‚count(preceding::node())'/>.1
  </xsl:variable>
  <address identifier="{$id_2}"
           id="{id}"
           lastname="{lastname}"/>
</xsl:template>

</xsl:stylesheet>
```


Combination of the results - Example

Source

```
<table>
 <row>
  <id>1</id>
  <lastname>Name1
  </lastname>
 </row>
 <row>
  <id>2</id>
  <lastname>Name2
  </lastname>
 </row>
 <row>
  <id>3</id>
  <lastname>Name3
  </lastname>
 </row>
<table>
```

Q1=/table/address
[@id=‘1‘ or @id=‘3‘]/@*

```
<table identifier="1.0.1">
 <address identifier="1.0.1.1.2.1"
  id="1" lastname="Name1"/>
 <address identifier="1.0.1.1.50.1"
  id="3" lastname="Name3"/>
</table>
```

Q2=/table/address
[@id=‘2‘]/@*

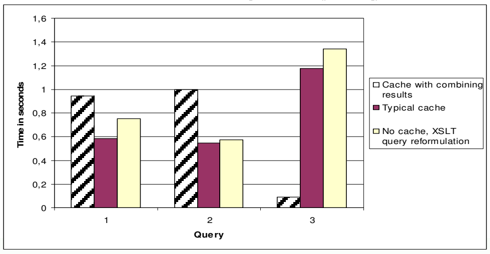
```
<table identifier="1.0.1">
 <address identifier="1.0.1.1.26.1"
  id="2" lastname="Name2"/>
</table>
```

Cache Result after Q2

```
<table identifier="1.0.1">
 <address identifier="1.0.1.1.2.1"
  id="1" lastname="Name1"/>
 <address identifier="1.0.1.1.26.1"
  id="2" lastname="Name2"/>
 <address identifier="1.0.1.1.50.1"
  id="3" lastname="Name3"/>
</table>
```

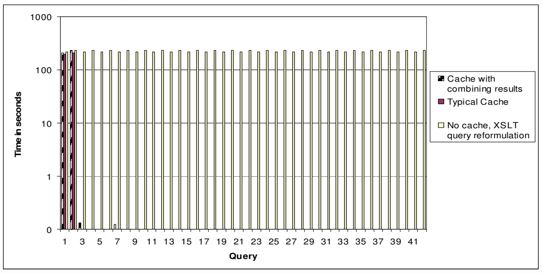
Performance Analysis 1/3

- 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'])



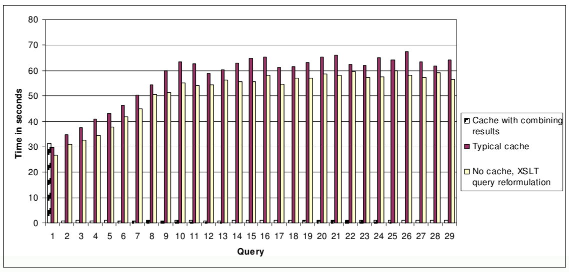
Performance Analysis 2/3

- 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

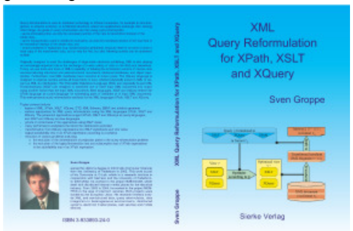


Comparing the 2 different Caching Strategies

	Typical Cache	Combining Results
Reuse results if new query is subsumed by previous query	✓	✓
Reuse results even if new query is not subsumed by previous query (and only intersects previous queries)	-	✓
Load only missing section into cache if new query cannot be directly answered from the cache	-	✓
No preprocessing step of XSLT view necessary	✓	-
No combination of the results necessary	✓	-

Questions / Remarks?

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



Sven Groppe
sven.groppe@deri.org
http://members.deri.at/~sveng/