Purely Relational XQuery
Database-Supported XML Processors

Torsten Grust
Technische Universität München

http://www-db.in.tum.de/~grust/  LUG Erding, Oct 2007
A Database Guy’s View of XML, XPath, and XQuery
A Database Guy’s View of XML, XPath, and XQuery

• Build XML processors that do not stumble once XML input volumes become sizable.
A Database Guy’s View of XML, XPath, and XQuery

• Build XML processors that do not stumble once XML input volumes become sizable.

• Wait! Sizable data volume...? Use Relational Databases!
A Database Guy’s View of XML, XPath, and XQuery

- Build XML processors that do not stumble once XML input volumes become sizable.
- Wait! Sizable data volume...? Use Relational Databases!

```xml
<?xml version="1.0"?>
<root>
  <elem att="val">
    <!-- comment -->
    text  
  </elem>
</root>
```

```xml
let $doc := doc("in.xml")
for $t in $doc//text()
return
  count($t/../comment())
```
A Database Guy’s View of XML, XPath, and XQuery

• Build XML processors that do not stumble once XML input volumes become sizable.

• Wait! Sizable data volume...? Use Relational Databases!

```xml
<?xml version="1.0"?>
<root>
    <elem att="val">
        <!-- comment -->
        text
    </elem>
</root>
```

```xquery
let $doc := doc("in.xml")
for $t in $doc//text()
return
  count($t/../comment())
```
A Database Guy’s View of XML, XPath, and XQuery

• Build XML processors that do not stumble once XML input volumes become sizable.

• Wait! Sizable data volume...? Use Relational Databases!

```xml
let $doc := doc("in.xml")
for $t in $doc//text()
return
  count($t/../comment())
```
A Database Guy’s View of XML, XPath, and XQuery

- Build XML processors that do not stumble once XML input volumes become sizable.
- Wait! Sizable data volume...? Use Relational Databases!
A Database Guy’s View of XML, XPath, and XQuery

- Build XML processors that do not stumble once XML input volumes become sizable.
- Wait! Sizable data volume...? Use Relational Databases!

```
SELECT   pre, size,
         ROW_NUMBER (…)
FROM     t000
WHERE    value
GROUP BY iter, …
```
RDBMSs as Processors for Non-Relational Languages

- Relational databases contain the best understood and most scalable query processors available today.
- Can we use relational query engines *as-is* to efficiently process non-relational languages?
RDBMSs as Processors for Non-Relational Languages

- Relational databases contain the best understood and most scalable query processors available today.
- Can we use relational query engines *as-is* to efficiently process non-relational languages?
RDBMSs as Processors for Non-Relational Languages

- Relational databases contain the best understood and most scalable query processors available today.

- Can we use relational query engines *as-is* to efficiently process non-relational languages?

SQL / Relational Algebra

$R \xrightarrow{L} X \xrightarrow{R^{-1}} X$
RDBMSs as Processors for Non-Relational Languages

- Relational databases contain the best understood and most scalable query processors available today.
- Can we use relational query engines as-is to efficiently process non-relational languages?
Purely Relational XQuery

- Find a relational representation of XQuery (XML, XPath) that leverages the table-based processing model:
  - Exploit functionality already built into the database system (do not invade the database kernel):
    - SQL idioms, OLAP extensions, partitioned B-Trees.
  - Run on top off-the-shelf relational database systems:
Purely Relational XQuery

- Find a relational representation of XQuery (XML, XPath) that leverages the table-based processing model:

- Exploit functionality already built into the database system (do not invade the database kernel):
  
  SQL idioms, OLAP extensions, partitioned B-Trees.

- Run on top off-the-shelf relational database systems:

  IBM DB2 V9  SQL Server 2005  PostgreSQL  kdb+  MonetDB
Pathfinder
pathfinder-xquery.org

Relational Database System (Kernel)
Pathfinder
pathfinder-xquery.org

Pathfinder XQuery Compiler
XPath/XQuery Semantics
XML Encoding

Relational Database System (Kernel)
Pathfinder
pathfinder-xquery.org

Pathfinder XQuery Compiler

XPath/XQuery Semantics
XML Encoding

Relational Database System (Kernel)
Pathfinder
pathfinder-xquery.org

Pathfinder XQuery Compiler

XPath/XQuery Semantics
XML Encoding

SQL
RA

Relational Database System
(Kernel)
Pathfinder & DB2®
Against 110+ MB of XML
Pathfinder & DB2
Against 110+ MB of XML
Slow Motion Replay

Pathfinder

XQuery

SQL

DB2®
Slow Motion Replay

XQuery (XMark Q8)

```xml
let $a := doc("XMark-110mb.xml")
return
  <item person="{ $p/name/text() }">
  { count($ca) }
  </item>
```
Slow Motion Replay

SQL:1999 (no SQL/XML)
WITH
  t0000 (iter,pre) AS
    SELECT ... FROM ... WHERE ... 
  t0001 (iter) AS
    SELECT ... FROM ... WHERE ... 
SELECT pre,size,kind,...
FROM ...
WHERE ...
ORDER BY ...
Slow Motion Replay

Relational Encoding of Result

<table>
<thead>
<tr>
<th>pre</th>
<th>size</th>
<th>kind</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>5072509</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5072510</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5072511</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pathfinder

SQL

XQuery

DB2®
Slow Motion Replay

Serialized XML Text

```xml
<?xml version="1.0" encoding="UTF-8"?>
<XQuery>
  <item person="Jixiang ...">0</item>
  <item person="Harpreet ...">0</item>
  ...
</XQuery>
```
Table Access Modes
Table Access Modes

• Relational query engines derive much of their efficiency from the simplicity of the *table of tuples* model.
Table Access Modes

- Relational query engines derive much of their efficiency from the simplicity of the *table of tuples* model.

1. Sequential scans:
   - read-ahead on disks,
   - prefetching CPU caches.
Table Access Modes

• Relational query engines derive much of their efficiency from the simplicity of the *table of tuples* model.

1. Sequential scans: read-ahead on disks, prefetching CPU caches.
Table Access Modes

- Relational query engines derive much of their efficiency from the simplicity of the *table of tuples* model.

1. Sequential scans: read-ahead on disks, prefetching CPU caches.

2. Index-based access: B-Trees, range scans.
Table Access Modes

- Relational query engines derive much of their efficiency from the simplicity of the *table of tuples* model.

1. Sequential scans: read-ahead on disks, prefetching CPU caches.

2. Index-based access: B-Trees, range scans.
Table Access Modes

- Relational query engines derive much of their efficiency from the simplicity of the *table of tuples* model.

1. Sequential scans: read-ahead on disks, prefetching CPU caches.

2. Index-based access: B-Trees, range scans.

- But: We will benefit **only if** we actually operate the relational database in this bulk-oriented mode.
The Core of XQuery
ordered item sequences

\((e_1, \ldots, e_n)\)
The Core of XQuery

$(e_1, \ldots, e_n)$

for $x$ in $e_1$ return $e_2$

ordered item sequences

iteration
The Core of XQuery

\[(e_1, \ldots, e_n)\]

for \(x \in e_1\) return \(e_2\)

let \(x := e_1\) return \(e_2\)

ordered item sequences
iteration
variable binding
The Core of XQuery

\[(e_1, \ldots, e_n)\]

for \(x\) in \(e_1\) return \(e_2\)

let \(x := e_1\) return \(e_2\)

if \((e_1)\) then \(e_2\) else \(e_3\)

ordered item sequences
iteration
variable binding
conditionals
(e_1, \ldots, e_n)
for $x$ in e_1 return e_2
let $x := e_1$ return e_2
if (e_1) then e_2 else e_3
<t> \{ e_1 \} </t>

ordered item sequences
iteration
variable binding
conditionals
XML node construction
The Core of XQuery

\((e_1, \ldots, e_n)\)

- for $x$ in $e_1$ return $e_2$
- let $x := e_1$ return $e_2$
- if ($e_1$) then $e_2$ else $e_3$
- `<t> \{ e_1 \} </t>`
- $e_1$/child::t, $e_1$/following::t

ordered item sequences
iteration
variable binding
conditionals
XML node construction
XPath location steps
The Core of XQuery

\[(e_1,\ldots, e_n)\]

for $x$ in $e_1$ return $e_2$

let $x := e_1$ return $e_2$

if (e_1) then $e_2$ else $e_3$

\[<t> \{ e_1 \} </t>\]

e_1/child::*:t, e_1/following::*:t

unordered \{ e_1 \}

ordered item sequences
iteration
variable binding
conditionals
XML node construction
XPath location steps
local order indifference
The Core of XQuery

(e₁,..,eₙ)
for $x$ in e₁ return e₂
let $x := e₁$ return e₂
if (e₁) then e₂ else e₃
<t> { e₁ } </t>
e₁/child::t, e₁/following::t
unordered { e₁ }
fn:doc(e₁), fn:data(e₁)

+ Document order, node identity, dynamic typing, schema validation, user-defined functions, ...
The Core of XQuery

(e₁, .., eₙ)
for $x$ in e₁ return e₂
let $x := e₁$ return e₂
if (e₁) then e₂ else e₃
<t> { e₁ } </t>
e₁/child::*:t, e₁/following::*:t
unordered { e₁ }
fn:doc(e₁), fn:data(e₁)

ordered item sequences
iteration
variable binding
conditionals
XML node construction
XPath location steps
local order indifference
built-in functions

+ Document order, node identity, dynamic typing, schema validation, user-defined functions, ...

• The XQuery–SQL gap appears substantial.
FP-Style Iteration
FP-Style Iteration

- The XQuery `for ..in..return` construct performs side-effect free iteration:

  \[
  \text{for } x \text{ in } (e_1,..,e_n) \\
  \text{return } b
  \]
FP-Style Iteration

• The XQuery `for ..in..return` construct performs side-effect free iteration:

\[
\text{for } \$x \text{ in } (e_1, \ldots, e_n) \\
\text{return } b \\
\equiv \\
(b[e_1/$x$], \ldots, b[e_n/$x$])
\]

• Individual iterations cannot interfere and may be evaluated in any order (or in parallel).
Loop Lifting

• “Relational loop unrolling”: For any XQuery subexpression, generate a relational plan that evaluates the expression in all iterations.
Loop Lifting

- “Relational loop unrolling”: For any XQuery subexpression, generate a relational plan that evaluates the expression in all iterations.

```xml
for $x in (10,20,30)
return $x + 42
```
Loop Lifting

• “Relational loop unrolling”: For any XQuery subexpression, generate a relational plan that evaluates the expression in all iterations.

```
for $x in (10, 20, 30)
return $x + 42
```
Loop Lifting

• “Relational loop unrolling”: For any XQuery subexpression, generate a relational plan that evaluates the expression in all iterations.

```xquery
for $x$ in (10, 20, 30)
return $x + 42
```

<table>
<thead>
<tr>
<th>iter</th>
<th>pos</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>42</td>
</tr>
</tbody>
</table>
Loop Lifting

• “Relational loop unrolling”: For any XQuery subexpression, generate a relational plan that evaluates the expression in all iterations.

for $x$ in (10, 20, 30)
return $x + 42$

for $x$ in (10, 20, 30)
return $x + 42$
Loop Lifting

- “Relational loop unrolling”: For any XQuery subexpression, generate a relational plan that evaluates the expression in *all iterations*.

```xquery
for $x$ in (10, 20, 30)
return $x + 42
```

```xquery
for $x$ in (10, 20, 30)
return $x$ + 42
```

```xquery
let $x := (10, 20, 30)$
return $x$
```
Loop Lifting

“Relational loop unrolling”: For any XQuery subexpression, generate a relational plan that evaluates the expression in all iterations.

for $x$ in (10, 20, 30)
return $x + 42$

for $x$ in (10, 20, 30)
return $x + 42$

let $x := (10, 20, 30)$
return $x$
Loop Lifting
Loop Lifting

for $x$ in (10, 20, 30)
return $x + 42$
Loop Lifting

for $x$ in (10, 20, 30)
return $x + 42$

<table>
<thead>
<tr>
<th>iter</th>
<th>pos</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>iter$_1$</th>
<th>pos$_1$</th>
<th>item$_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>42</td>
</tr>
</tbody>
</table>
Loop Lifting

for $x$ in (10, 20, 30)
return $x + 42$
Loop Lifting

for $x$ in (10, 20, 30)
return $x + 42$
Loop Lifting

for $x$ in (10, 20, 30)
return $x + 42$
Loop Lifting

for $x$ in $(10, 20, 30)$
return $x + 42$

```
<table>
<thead>
<tr>
<th>iter</th>
<th>pos</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>30</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>iter1</th>
<th>pos1</th>
<th>item1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>42</td>
</tr>
</tbody>
</table>
```

```
itorio = iter1
```

```
item2: (item, item1)
```

```
π
```

```
<table>
<thead>
<tr>
<th>iter</th>
<th>pos</th>
<th>item2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>52</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>62</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>72</td>
</tr>
</tbody>
</table>
```
Loop Lifting
Loop Lifting

for $x$ in (1,2,3,4)
return if ($x$ mod 2 = 0) then "even" else "odd"
Loop Lifting

for $x$ in (1, 2, 3, 4)
return if ($x \mod 2 = 0$) then "even" else "odd"

<table>
<thead>
<tr>
<th>iter</th>
<th>pos</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>true</td>
</tr>
</tbody>
</table>
Loop Lifting

for $x$ in $(1,2,3,4)$
return if ($x \mod 2 = 0$) then "even" else "odd"
Loop Lifting

for $x$ in $(1,2,3,4)$
return if ($x \mod 2 = 0$) then "even" else "odd"
Loop Lifting

for $x$ in (1,2,3,4)
return if ($x \mod 2 = 0$) then "even" else "odd"
Loop Lifting

for $x$ in (1,2,3,4)
return if ($x \mod 2 = 0$) then "even" else "odd"
Loop Lifting

for $x$ in (1,2,3,4)
return if ($x \mod 2 = 0$) then "even" else "odd"
Intermediate Code: Relational Algebra

Pathfinder XQuery Compiler

Relational Algebra

\( \sigma \ \pi \ \bowtie \ \cup \ \cdots \)
Intermediate Code: Relational Algebra

- Compiles into RA that mimics capabilities of SQL:1999 query engines.
- RA dialect is ...
  1. primitive
  2. explicit
  3. designed to be easily analyzable
Intermediate Code: Relational Algebra

- Compiles into RA that mimics capabilities of SQL:1999 query engines.
- RA dialect is ...
  1. primitive
  2. explicit
  3. designed to be easily analyzable
Intermediate Code: Relational Algebra

- Compiles into RA that mimics capabilities of SQL:1999 query engines.
- RA dialect is ...
  1. primitive
  2. explicit
  3. designed to be easily analyzable
Typical Plans ...
Typical Plans ...
Relational XQuery Optimization

```
let $d := fn:doc(···)
return for $a in $d//a
  for $b in $d//b
  where $b/@c = $a/@d
  return $b
```
We use concepts in the *relational domain* to analyze plans and to steer simplification and optimization.

Detect join-like XQuery expressions. Let XQuery’s syntactic diversity not affect the detection:

```xml
let $d := fn:doc(⋯)
return for $a in $d//a
  for $b in $d//b
  where $b/@c = $a/@d
return $b
```
Relational XQuery Optimization

• We use concepts in the *relational domain* to analyze plans and to steer simplification and optimization.

• Detect join-like XQuery expressions. Let XQuery’s syntactic diversity not affect the detection:

\[
\begin{align*}
\text{let } & \$d := \text{fn:doc(⋯)} \\
\text{return } & \$d//b[@c = \$d//a/@c]
\end{align*}
\]
Relational XQuery Optimization

- We use concepts in the *relational domain* to analyze plans and to steer simplification and optimization.
- Detect join-like XQuery expressions. Let XQuery’s syntactic diversity not affect the detection:

  \[
  \text{let } d := \text{fn:doc}(\ldots) \\
  \text{return } d//b[@c = d//a/@c]
  \]

- For both queries, the value-based XQuery join surfaces as a multi-valued dependency in the algebraic plans.
Rewriting XMark Q8
Rewriting XMark Q8

- Rewriting phases:
  1. Constant propagation and projection pushdown,
Rewriting XMark Q8

- Rewriting phases:
  1. Constant propagation and projection pushdown,
  2. functional dependency and data flow analysis,
Rewriting XMark Q8

- Rewriting phases:
  1. Constant propagation and projection pushdown,
  2. functional dependency and data flow analysis,
  3. algebraic XQuery join detection,
Rewriting XMark Q8

- Rewriting phases:
  1. Constant propagation and projection pushdown,
  2. functional dependency and data flow analysis,
  3. algebraic XQuery join detection,
  4. XPath join graph isolation,
Rewriting XMark Q8

- Rewriting phases:
  1. Constant propagation and projection pushdown,
  2. functional dependency and data flow analysis,
  3. algebraic XQuery join detection,
  4. XPath join graph isolation,
  5. data guide exploitation ("Node GPS").

Pathfinder & IBM DB2 V9, 115 MB XML input data
< 1 sec
Rewriting XMark Q8

- Rewriting phases:
  1. Constant propagation and projection pushdown,
  2. functional dependency and data flow analysis,
  3. algebraic XQuery join detection,
  4. XPath join graph isolation,
  5. data guide exploitation ("Node GPS")
Can DB2® Cope?
Can DB2® Cope?

Prof. Dr. Torsten Grust

Technische Universität München
Can DB2® Cope?
Can DB2® Cope?

Prof. Dr. Torsten Grust

Technische Universität München
Can DB2® Cope?

Prof. Dr. Torsten Grust
Can DB2® Cope?
Can DB2® Cope?

Can           Cope?

SQL Reference
Volume 1
DB2®

DB2 Version 9
for Linux, UNIX, and Windows

SC10-4249-00  

/SV610000/SV620000/SV630000
Can **DB2** Cope?

*) Indexes avised by DB2 itself
Order Indifference

let $\text{warning} := \langle p \rangle \text{Do <em>not</em> press button, computer will <em>explode!</em>} \rangle

return \text{fn:distinct-doc-order}(\text{for } x \text{ in } \text{warning}/* \\
\text{return } x/\text{text}())
Order Indifference

• Order in XML and XQuery processing is essential:

```xml
let $warning := <p>Do <em>not</em> press button,
    computer will <em>explode!</em></p>
return fn:distinct-doc-order(
    for $x in $warning//*[@
        return $x/text()
    ]
    return $x/text() )
```
Order Indifference

- Order in XML and XQuery processing is essential:

```
let $warning := <p>Do <em>not</em> press button, computer will <em>explode</em>!</p>
return fn:distinct-doc-order( for $x in $warning//*
                          return $x/text() )
```

Do not press button, computer will explode!
Order Indifference

- Order in XML and XQuery processing is essential:

```xml
let $warning := <p>Do <em>not</em> press button, computer will <em>explode!</em></p>
return for $x in $warning//*
    return $x/text()
```

Do press button, computer will not explode!
Order Indifference

• Order in XML and XQuery processing is essential:

let $warning := <p>Do <em>not</em> press button, computer will <em>explode!</em></p>
return for $x in $warning//*[node()]
return $x/text()

Do press button, computer will not explode!

• Order-awareness thus is often **deeply wired** into XQuery processors.

• Supporting constructs like unordered `{ e₁ }` is challenging.
Order Indifference

- Order in XML and XQuery processing is essential:

```
let $warning := <p>Do <em>not</em> press button, computer will <em>explode!</em></p>
return for $x in $warning//@*
  return $x/text()
```

Do press button, computer will not explode!

- Order-awareness thus is often deeply wired into XQuery processors.
- Supporting constructs like unordered \{ e_1 \} is challenging.
Order Indifference

- Order in XML and XQuery processing is essential:

  let $warning := <p>Do <em>not</em> press button, 
  computer will <em>explode!</em></p>
  return for $x in $warning/*
  return $x/text()

Do press button, computer will not explode!

- Order-awareness thus is often deeply wired into XQuery processors.

- Supporting constructs like unordered \{ e_1 \} is challenging.
More Purely Relational XQuery

- Declarative XQuery debugging with “time travel”.
- Users observe expressions and may traverse iterations forward/backward.
More Purely Relational XQuery

- Declarative XQuery debugging with “time travel”.
- Users observe expressions and may traverse iterations forward/backward.
More Purely Relational XQuery

- Declarative XQuery debugging with “time travel”.
- Users observe expressions and may traverse iterations forward/backward.
- Dependable cardinality estimates at XQuery subexpression level (# items per iteration and overall contribution).

```xml
for $x in $doc//x
return if (e₁) then e₂ else e₃
```
More Purely Relational XQuery

• Declarative XQuery debugging with “time travel”.

• Users observe expressions and may traverse iterations forward/backward.

• Dependable cardinality estimates at XQuery subexpression level (# items per iteration and overall contribution).

\[
\text{for } x \text{ in } \text{doc} // x \\
\text{return if (e_1) then } e_2 \text{ else } e_3
\]
Atomization Indexes

<a>
  <b>
    <c>Same</c><d>shirt</d></c>
    different
  </b>
  day
</a>
Atomization Indexes

Same shirt</c> different day
</a>
Atomization Indexes

\[ \text{Same shirt} \]
\[ \text{different day} \]
Atomization Indexes

<table>
<thead>
<tr>
<th>node</th>
<th>atom1</th>
<th>atom2</th>
<th>atom3</th>
<th>atom4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(<a>\)\(<b>Same\(<d>shirt</d>\)</c> different\(</b>\)\(</a>\)

\(<c>Same\(<d>shirt</d>\)</c> \(<b>different\)</b> day \(</a>\)

Day different
### Atomization Indexes

<table>
<thead>
<tr>
<th>node</th>
<th>atom1</th>
<th>atom2</th>
<th>atom3</th>
<th>atom4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The diagram illustrates the relations between the nodes, with some nodes labeled as `Same` and others as `different`.

The table shows the distribution of `a`, `b`, `c`, `d`, and `shirt` across the nodes, with some entries marked as `Same` or `different`.
Atomization Indexes

\[
\begin{array}{cccc}
\text{node} & \text{atom1} & \text{atom2} & \text{atom3} & \text{atom4} \\
a & & & & \\
b & & & & \\
c & & & & \\
\circ & \text{Same} & & & \\
d & \circ & \text{shirt} & & \\
\circ & \text{different} & & & \\
\circ & \text{day} & & & \\
\end{array}
\]

\[
\begin{align*}
&lt;a&gt;
\circ \text{Same} &lt;d&gt; \text{shirt} \&lt;/d&gt;&lt;/c&gt; \\
\text{different} \\
&lt;/b&gt; \\
\text{day} \\
&lt;/a&gt;
\end{align*}
\]
Atomization Indexes

<\textit{a}> <\textit{b}> <\textit{c}> \textit{Same}<\textit{d}> \textit{shirt}</\textit{d}></\textit{c}> different </\textit{b}> day </\textit{a}>

<table>
<thead>
<tr>
<th>node</th>
<th>atom1</th>
<th>atom2</th>
<th>atom3</th>
<th>atom4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td></td>
<td>Same</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td>shirt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td>shirt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td>shirt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td>different</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td>day</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Atomization Indexes

- `<a>`
- `<b>`
- `<c>`
- `<d>`

Different

Same

<table>
<thead>
<tr>
<th>node</th>
<th>atom1</th>
<th>atom2</th>
<th>atom3</th>
<th>atom4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Same</td>
<td>shirt</td>
<td>different day</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Same</td>
<td>shirt</td>
<td>different</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>Same</td>
<td>shirt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>Same</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>shirt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>shirt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>different</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>day</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Atomization Indexes

<table>
<thead>
<tr>
<th>node</th>
<th>atom1</th>
<th>atom2</th>
<th>atom3</th>
<th>atom4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Same</td>
<td>shirt</td>
<td>different</td>
<td>day</td>
</tr>
<tr>
<td>b</td>
<td>Same</td>
<td>shirt</td>
<td>different</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>Same</td>
<td>shirt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Same</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>shirt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>shirt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>different</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>day</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Atomization Indexes

### Node Atomization

<table>
<thead>
<tr>
<th>node</th>
<th>atom1</th>
<th>atom2</th>
<th>atom3</th>
<th>atom4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$\delta_1$</td>
<td>shirt</td>
<td>different</td>
<td>day</td>
</tr>
<tr>
<td>b</td>
<td>$\delta_1$</td>
<td>shirt</td>
<td>different</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>$\delta_1$</td>
<td>shirt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>$\delta_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>shirt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>shirt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>different</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>day</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Dictionary

<table>
<thead>
<tr>
<th>dict</th>
<th>atom</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta_1$</td>
<td>Same</td>
</tr>
</tbody>
</table>

### Diagram

The diagram illustrates the atomization process with nodes representing different values and edges showing relationships. The node 'a' is connected to 'b', 'c', and 'd', with 'b', 'c', and 'd' each showing the attributes 'shirt', 'different', and 'day' respectively.
Atomization Indexes

\[
\begin{array}{cccc}
\text{node} & \text{atom1} & \text{atom2} & \text{atom3} \\
\hline
a & \delta_1 & \delta_2 & \delta_3 \\
b & \delta_1 & \delta_2 & \delta_3 \\
c & \delta_1 & \delta_2 & \\
d & \delta_2 & \\
o & \delta_1 & \\
o & \delta_2 & \\
o & \delta_3 & \\
o & \delta_4 & \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{dict} & \text{atom} \\
\hline
\delta_1 & \text{Same} \\
\delta_2 & \text{shirt} \\
\delta_3 & \text{different} \\
\delta_4 & \text{day} \\
\end{array}
\]
Atomization Indexes

<table>
<thead>
<tr>
<th>node</th>
<th>atom1</th>
<th>atom2</th>
<th>atom3</th>
<th>atom4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>δ₅</td>
<td></td>
<td>δ₃</td>
<td>δ₄</td>
</tr>
<tr>
<td>b</td>
<td>δ₅</td>
<td></td>
<td>δ₃</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>δ₅</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>δ₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>δ₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>δ₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>δ₃</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>δ₄</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- **dict**:
  - δ₁: Same
  - δ₂: shirt
  - δ₃: different
  - δ₄: day
  - δ₅: Same | shirt

- **atom**:

- node atom1 atom2 atom3 atom4
  - a δ₅
  - b δ₅
  - c δ₅
  - o δ₁
  - d δ₂
  - o δ₂
  - o δ₃
  - o δ₄
  - o δ₅

- **Day**
  - Same
  - different

- **Shirt**
  - Same

Example:

- `<a>`
- `<b>`
- `<c>` Same `<d>` shirt `</d>` `</c>`
- `different`
Atomization Indexes

\[ \begin{array}{|c|c|c|c|}
\hline
\text{node} & \text{atom1} & \text{atom2} & \text{atom3} & \text{atom4} \\
\hline
a & \delta_6 & & & \delta_4 \\
b & \delta_6 & & & \\
c & \delta_5 & & & \\
o & \delta_1 & & & \\
d & \delta_2 & & & \\
o & \delta_2 & & & \\
o & \delta_3 & & & \\
o & \delta_4 & & & \\
\hline
\end{array} \]

\[
\begin{array}{|c|c|c|}
\hline
\text{dict} & \text{atom} \\
\hline
\delta_1 & \text{Same} \\
\delta_2 & \text{shirt} \\
\delta_3 & \text{different} \\
\delta_4 & \text{day} \\
\delta_5 & \delta_1 & \delta_2 \\
\delta_6 & \delta_5 & \delta_3 \\
\hline
\end{array}
\]
Atomization Indexes

\[\begin{array}{|c|c|c|}
\hline
\text{node} & \text{atom1} & \text{atom2} \\
\hline
a & \delta_6 & \delta_4 \\
b & \delta_6 \\
c & \delta_5 \\
d & \delta_2 \\
\# & \delta_2 \\
\# & \delta_3 \\
\# & \delta_4 \\
\hline
\end{array}\]
Relational Encodings for XML

• XML documents represent ordered, unranked trees (nodes of 7 kinds).

• Relational XQuery processing calls for an adequate relational encoding of such trees:

1. Schema-oblivious (XQuery constructs arbitrary XML fragments),

2. accessible for the query processor and index support (≫10^7 nodes in typical GB-range XML instances).
Relational Encodings for XML

- XML documents represent ordered, unranked trees (nodes of 7 kinds).
- Relational XQuery processing calls for an adequate relational encoding of such trees:
  1. Schema-oblivious (XQuery constructs arbitrary XML fragments),
  2. accessible for the query processor and index support ($\gg 10^7$ nodes in typical GB-range XML instances).

<table>
<thead>
<tr>
<th>doc</th>
<th>XML Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>...</td>
</tr>
</tbody>
</table>
Pre/Postorder Encoding

```
<a>
    <b><c><d/>e</c></b>
    <f><!--g-->
        <h><i/><j/></h>
    </f>
</a>
```
Pre/Postorder Encoding

```
<a>
  <b><c><d/>e</c></b>
  <f><!--g--></f>
  <h><i/><j/></h>
</a>
```
Pre/Postorder Encoding

```
<a>
  <b><c><d/>e</c></b>
  <f><!--g--><h><i/><j/></h></f>
</a>
```
Pre/Postorder Encoding

\(<a>\\\n  \text{<b><c><d/>e</c></b>}\\n  \text{<f><!--g-->}\\n  \text{<h><i/><j/></h>}\\n</f>\\n</a>\n
<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>h</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>i</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>j</td>
</tr>
</tbody>
</table>
Pre/Postorder Encoding

\[
\begin{align*}
\langle a \rangle \\
\langle b \rangle \langle c \rangle \langle d \rangle \langle e \rangle \langle /c \rangle \langle /b \rangle \\
\langle f \rangle \langle !--g--\rangle \\
\langle /h \rangle \langle i \rangle \langle /j \rangle \langle /h \rangle \\
\langle /f \rangle \\
\langle /a \rangle 
\end{align*}
\]

Pre order: 0 1 2 3 4 5 6 7 8 9
Post order: a b c d e f g h i j
Node: a b c d e f g h i j

Document order:
0 9
1 3
2 2
3 0
4 1
5 8
6 4
7 7
8 5
9 6

Pre table:
<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>h</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>i</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>j</td>
</tr>
</tbody>
</table>
Pre/Postorder Encoding

\[
\begin{align*}
\text{pre} & \quad \text{post} & \quad \text{node} & \quad \text{kind} \\
0 & \quad 9 & \quad \text{a} & \quad \text{elem} \\
1 & \quad 3 & \quad \text{b} & \quad \text{elem} \\
2 & \quad 2 & \quad \text{c} & \quad \text{elem} \\
3 & \quad 0 & \quad \text{d} & \quad \text{elem} \\
4 & \quad 1 & \quad \text{e} & \quad \text{text} \\
5 & \quad 8 & \quad \text{f} & \quad \text{elem} \\
6 & \quad 4 & \quad \text{g} & \quad \text{comm} \\
7 & \quad 7 & \quad \text{h} & \quad \text{elem} \\
8 & \quad 5 & \quad \text{i} & \quad \text{elem} \\
9 & \quad 6 & \quad \text{j} & \quad \text{elem} \\
\end{align*}
\]

\[
\begin{align*}
\text{document order} \\
\text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} & \text{h} & \text{i} & \text{j} \\
\end{align*}
\]

\[
\begin{align*}
\text{pre} & \quad \text{post} & \quad \text{node} & \quad \text{kind} \\
9 & \quad 0 & \quad \text{a} & \quad \text{elem} \\
3 & \quad 1 & \quad \text{b} & \quad \text{elem} \\
2 & \quad 2 & \quad \text{c} & \quad \text{elem} \\
0 & \quad 3 & \quad \text{d} & \quad \text{elem} \\
1 & \quad 4 & \quad \text{e} & \quad \text{text} \\
8 & \quad 5 & \quad \text{f} & \quad \text{elem} \\
7 & \quad 6 & \quad \text{g} & \quad \text{comm} \\
6 & \quad 7 & \quad \text{h} & \quad \text{elem} \\
5 & \quad 8 & \quad \text{i} & \quad \text{elem} \\
4 & \quad 9 & \quad \text{j} & \quad \text{elem} \\
\end{align*}
\]

\[
\begin{align*}
\text{pre} & \quad \text{post} & \quad \text{node} & \quad \text{kind} \\
0 & \quad 9 & \quad \text{a} & \quad \text{elem} \\
1 & \quad 3 & \quad \text{b} & \quad \text{elem} \\
2 & \quad 2 & \quad \text{c} & \quad \text{elem} \\
3 & \quad 0 & \quad \text{d} & \quad \text{elem} \\
4 & \quad 1 & \quad \text{e} & \quad \text{text} \\
5 & \quad 8 & \quad \text{f} & \quad \text{elem} \\
6 & \quad 4 & \quad \text{g} & \quad \text{comm} \\
7 & \quad 7 & \quad \text{h} & \quad \text{elem} \\
8 & \quad 5 & \quad \text{i} & \quad \text{elem} \\
9 & \quad 6 & \quad \text{j} & \quad \text{elem} \\
\end{align*}
\]
Pre/Postorder Encoding

<pre>
&lt;a&gt;
 &lt;b&gt;&lt;c&gt;&lt;d/&gt;&lt;/c&gt;&lt;/b&gt;
 &lt;f&gt;&lt;!--g--&gt;
 &lt;h&gt;&lt;i/&gt;&lt;j/&gt;&lt;/h&gt;
&lt;/f&gt;
&lt;/a&gt;
</pre>
Pre/Postorder Encoding

```
<a>
  <b>&lt;c&gt;&lt;d/&gt;&lt;/b&gt;
  &lt;f&gt;&lt;!--g-->&lt;/f&gt;
</a>
```

```
<table>
<thead>
<tr>
<th>node</th>
<th>kind</th>
<th>size</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>elem</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>elem</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>elem</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>elem</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>text</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>f</td>
<td>elem</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>comm</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>h</td>
<td>elem</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>i</td>
<td>elem</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>elem</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
```

Pre/Postorder Encoding Diagram
Pre/Postorder Encoding

\[
\langle a \rangle \\
\langle b \rangle \langle c \rangle \langle d / \rangle e \langle / c \rangle \langle / b \rangle \\
\langle f \rangle \langle ! \-- g \-- \rangle \\
\langle h \rangle \langle i \rangle \langle j \rangle \langle / h \rangle \\
\langle / f \rangle \\
\langle / a \rangle \\
\]

Postorder:

Preorder:

1b3
2c2
3d0
4e1
5f8
6g4
7h7
8i5
9j6

```
<table>
<thead>
<tr>
<th>node</th>
<th>kind</th>
<th>size</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>elem</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>elem</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>elem</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>elem</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>text</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>f</td>
<td>elem</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>comm</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>h</td>
<td>elem</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>i</td>
<td>elem</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>elem</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
```
Pre/Postorder Encoding

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

```
  0a9
   1b3
    2c2
    3d0

  4e1
  5f8
   6g4
   7h7

  8i5
  9j6
```

```
<table>
<thead>
<tr>
<th>node</th>
<th>kind</th>
<th>size</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>elem</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>elem</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>elem</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>elem</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>text</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>f</td>
<td>elem</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>comm</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>h</td>
<td>elem</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>i</td>
<td>elem</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>elem</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
```
B-Trees Accelerate
XPath Location Steps

- The XPath axes *ancestor*, *descendant*, *preceding*, *following* partition any XML document:
B-Trees Accelerate XPath Location Steps

- The XPath axes *ancestor*, *descendant*, *preceding*, *following* partition any XML document:
B-Trees Accelerate

XPath Location Steps

- The XPath axes *ancestor*, *descendant*, *preceeding*, *following* partition any XML document:
B-Trees Accelerate XPath Location Steps

- The XPath axes \textit{ancestor, descendant, preceding, following} partition any XML document:
B-Trees Accelerate XPath Location Steps

- The XPath axes `ancestor`, `descendant`, `preceding`, `following` partition any XML document:
The XPath axes ancestor, descendant, preceding, following partition any XML document:
B-Trees Accelerate XPath Location Steps

• The XPath axes ancestor, descendant, preceding, following partition any XML document:
B-Trees Accelerate XPath Location Steps

- The XPath axes *ancestor*, *descendant*, *preceding*, *following* partition any XML document:
Partitioned B-Trees
Partitioned B-Trees

- Use low selectivity key prefixes to partition the XML nodes in a B-Tree according to various criteria:
Partitioned B-Trees

- Use low selectivity key prefixes to partition the XML nodes in a B-Tree according to various criteria:

  1. level (support XPath child axis)
Partitioned B-Trees

- Use low selectivity key prefixes to partition the XML nodes in a B-Tree according to various criteria:
  1. level (support XPath child axis)
Partitioned B-Trees

- Use low selectivity key prefixes to partition the XML nodes in a B-Tree according to various criteria:
  1. level (support XPath child axis)
  2. element tag name
  3. path to node (“Node GPS”)

- Given a Pathfinder-generated workload, the index advisor of IBM DB2 V9 proposes such partitionings automatically.
Injecting More Tree Awareness

- In XQuery, an XPath location step originates in a sequence of context nodes.

Duplicate nodes, out of order results (XPath semantics!), wasted work.
Injecting More Tree Awareness

- In XQuery, an XPath location step originates in a sequence of context nodes.

Duplicate nodes, out of order results (XPath semantics!), wasted work.
Injecting More Tree Awareness

- In XQuery, an XPath location step originates in a sequence of context nodes.

Duplicate nodes, out of order results (XPath semantics!), wasted work.
Injecting More Tree Awareness

- In XQuery, an XPath location step originates in a sequence of context nodes.

Duplicate nodes, out of order results (XPath semantics!), wasted work.
Injecting More Tree Awareness

- In XQuery, an XPath location step originates in a sequence of context nodes.

Duplicate nodes, out of order results (XPath semantics!), wasted work.
Injecting More Tree Awareness

• In XQuery, an XPath location step originates in a sequence of context nodes.

Duplicate nodes, out of order results (XPath semantics!), wasted work.
Injecting More Tree Awareness

- In XQuery, an XPath location step originates in a sequence of context nodes.

Duplicate nodes, out of order results (XPath semantics!), wasted work.
Injecting More Tree Awareness

- In XQuery, an XPath location step originates in a sequence of context nodes.

Duplicate nodes, out of order results (XPath semantics!), wasted work.
Staircase Join: Context Pruning

- XPath following axis:
Staircase Join: Context Pruning

- XPath following axis:
Staircase Join: Context Pruning

- XPath following axis:
Staircase Join: Context Pruning

- **XPath following axis:**

- Index scan yields duplicate free result in document order.
Staircase Join: Skipping

• XPath descendant axis:
Staircase Join: Skipping

• XPath descendant axis:

\[ c_1 \rightarrow c_3 \]
Staircase Join: Skipping

- XPath descendant axis:
Staircase Join: Skipping

- XPath descendant axis:
Staircase Join: Skipping

- XPath descendant axis:

- Index scan yields duplicate free result in document order.
MonetDB/XQuery
MonetDB/XQuery

- MonetDB: Extensible relational database kernel, optimized for query processing close to the CPU.
- Full vertical fragmentation (column store).
MonetDB/XQuery

- MonetDB: Extensible relational database kernel, optimized for query processing close to the CPU.
- Full vertical fragmentation (column store).
MonetDB/XQuery

- MonetDB: Extensible relational database kernel, optimized for query processing close to the CPU.

- Full vertical fragmentation (column store).
MonetDB/XQuery

- MonetDB: Extensible relational database kernel, optimized for query processing close to the CPU.
- Full vertical fragmentation (column store).
- Pathfinder + MonetDB = MonetDB/XQuery.
- Implementation of XQuery 1.0.
- Evaluates queries against GB-range XML instances in interactive time.
- XQuery Update, XQuery RPC (execute at uri `{ e }`), support for multi-dimensional XML, ...
“DB2’s XML support doesn’t beat its relational self.”
“DB2’s XML support doesn’t beat its relational self.”

- Pathfinder builds on 30+ years of development of relational database technology.
“DB2’s XML support doesn’t beat its relational self.”

- Pathfinder builds on 30+ years of development of relational database technology.

- Outperforms the built-in DB2 V9 pureXML® engine, especially for large XML input instances.
“DB2’s XML support doesn’t beat its relational self.”

- Pathfinder builds on 30+ years of development of relational database technology.

- Outperforms the built-in DB2 V9 pureXML® engine, especially for large XML input instances.

- Covers other data-intensive languages:
  1. SQL/XML
“DB2’s XML support doesn’t beat its relational self”.

• Pathfinder builds on 30+ years of development of relational database technology.

• Outperforms the built-in DB2 V9 pureXML® engine, especially for large XML input instances.

• Covers other data-intensive languages:
  1. SQL/XML
  2. LINQ + other “Nested Loop” languages.
END SLIDE AREA

grust@in.tum.de
http://www-db.in.tum.de/~grust/