XQueC: embedding compression into XML databases

Tralala, Marseille, 03/2005

Andrei Arion$^{1,2}$  Angela Bonifati$^3$
Ioana Manolescu$^1$    Andrea Pugliese$^4$

$^1$ INRIA Futurs – LRI, PCRI, France
$^2$ University of Paris XI
$^3$ ICAR CNR, Italy
$^4$ DEIS, University of Calabria, Italy
XQueC: a (compressed) XML database

Storage and compression principles
  - Storage vs. compression vs. encoding

Path sequence-based XML storage model

The search for a compression configuration

Query processing in XQueC
  - Path sequence-based XQuery optimization
  - Physical operators
  - Putting it all together

Some experiments

Conclusion and future work
XQueC storage and compression principles
Compressing XML: lots of work to do!

XML document = structure + contents

First principle: *compress* the contents and *encode* the structure
   Encoding does shrink the structure a lot

Second principle: *lazy decompression*
   Query as much as possible in the compressed domain

Third principle: stay compatible with general "database approach"
   1. *Selective data access; low memory needs*
   2. *Access path selection; physical / logical data independence*
   3. *Set-at-a-time query processing / logical & physical operator algebras*
   4. *Trade compression ratio for "full database approach"*
The XQueC logical storage model: path sequences
The logical path sequences storage model

Split structure and content **by the incoming path**

One *path sequence* of [pre, post, depth] element IDs for each root-to-element path

One sequence of [pre, value] pairs for each root-to-leaf path: *container* (à la XMill)

Store each ID sequence and container separately, in document order
Structural identifier sequences

12 paths = 12 structural ID sequences
Structural identifier sequences

[1, 8, 1] <item @id=""item0"">  
  [2, 1, 2] <description> Beany Crocodile </description>  
  [3, 3, 2] <seller>  
    [4, 2, 3] <personRef>Joe</personRef>  
  </seller>  
  [5, 4, 2] <payment>Visa</payment>  
  [6, 5, 2] <payment>MasterCard</payment>  
  [7, 6, 2] <mailbox> </mailbox>  
  [8, 7, 2] <shipping> UPS in US only </shipping>  
</item>

Path sequence for /site/auctions/samerica/item: { [1, 8, 1] }  
Path sequence for ../item/payment: { [5, 4, 2], [6, 5, 2] }
Structural identifier sequences

[1, 8, 1]  <item @id=""item0"">
[2, 1, 2]  <description> Beany Crocodile </description>
[3, 3, 2]  <seller>
[4, 2, 3]   <personRef>Joe</personRef>
[5, 4, 2]  </seller>
[6, 5, 2]  <payment> Visa </payment>
[7, 6, 2]  <payment> MasterCard </payment>
[8, 7, 2]  <mailbox> </mailbox>
[8, 7, 2]  <shipping> UPS in US only </shipping>

</item>

Path sequence for /site/auctions/samerica/item:  { [1, 8] }
Path sequence for ../item/payment:  { [5, 4], [6, 5] }
Storing document content in path sequences

For each root-to-leaf path in the XML document, store a container: a sequence of (ID, value) pairs

Text nodes: IDs of the parent elements
Attributes: IDs of the enclosing elements

Container for /site/samerica/item/payment:
{ (5, "Visa")
  (6, "MasterCard") }  

Better idea:
{ (1, "Visa")
  (2, "MasterCard") }
XQueC data catalog and other structures

Catalog = path summary (= DataGuide)
  Each path summary node corresponds to a path sequence or a container
  Each path summary node contains the database location where the sequence / container starts

Other required structures
  Compression dictionaries (to be seen)

Other (optional) structures
  Simple cardinality information on path summary [AAN01]
  Container indexes
Physical storage for path sequences

ID sequences: fixed-length items
Containers: varying-length items
Ordered sequences (document order):
  - B⁺ - trees
  - Compact (continuous) sequences
Trade-off: compactness vs. suitability to updates
Implementation supports both, we mostly used sequences
The XQueC compression model
Compressing data values in XQueC

Lazy decompression entails

Fine-grained compression: access to each individual compressed value

Compressed container:

\{(n_1, \text{comp}(v_1)), (n_2, \text{compr}(v_2)), \ldots (n_k, \text{compr}(v_k))\}

Compression respectful of comparisons in a query workload, if one is known

Compatibility with database techniques entails

Ability to index directly compressed values
Algorithmic properties of value compression algorithms

Existing algorithms exhibit different algorithmic properties

**Equality-preserving**: if \( v_1 = v_2 \), then \( \text{compr}(v_1) = \text{compr}(v_2) \)

+ 2-pass Huffman, Antoshenkov-Murray-Lomet (ALM)...

– 1-pass Huffman and others

**Inequality-preserving**: if \( v_1 < v_2 \), then \( \text{compr}(v_1) < \text{compr}(v_2) \)

+ ALM, some arithmetic encodings

– Huffman

**Regexp-transparent**: \( v_1 \) like \( \text{regex} \) can be tested on \( \text{compr}(v_1) \).
Entails \( v_1 < \text{const} \) can also be tested on \( \text{compr}(v_1) \).
Is not inequality-preserving.

+ Moura et al.

– Huffman and others
Impact of compressor algorithmic properties

Existing algorithms exhibit different algorithmic properties

*Equality-preserving*: if $v_1 = v_2$, then $\text{compr}(v_1) = \text{compr}(v_2)$

  Allows evaluating "=" selections/joins in the compressed domain, hash-based indexing

*Inequality-preserving*: if $v_1 < v_2$, then $\text{compr}(v_1) < \text{compr}(v_2)$

  Allows evaluating "<" selections/joins in the compressed domain, B+-tree indexing

*Regexp-transparent*: $v_1$ like $\text{regexp}$ can be tested on $\text{compr}(v_1)$.
Entails $v_1 < \text{const}$ can also be tested on $\text{compr}(v_1)$.

  Allows evaluating "=" , "<", "like" selections in the compressed domain.
Performance properties of a compression algorithm

Compression speed
  Impacts loading & compression time

Compression ratio
  Impacts storage size

Decompression speed
  Impacts query & decompression time

Compression dictionary size
  Impacts storage size, memory needs for decompression
Choosing a compression configuration (1/2)

Configuration = partition over C, the set of containers

Containers in a partition group are compressed with the same compression algorithm and dictionary

[W: workload of XQuery queries → container comparisons ]

Cost of a configuration:

\[ \alpha \cdot \text{compr. time} + \beta \cdot \text{compressed data & dict. size} + \gamma \cdot \text{time of decompression due to compression mismatch} + \delta \cdot \text{time of decompression due to result construction} \]

Conflicting factors, weighted cost
Choosing a compression configuration (2/2)

Configuration = partition of the containers set + 1 algo per group

\[ W: \text{workload of XQuery queries} \rightarrow \text{container comparisons} \]

Start with initial configuration: each group contains 1 container, use e.g. Huffman

Greedily apply the most cost-effective move

- Join two groups
  - May reduce \( W \)-incurred decompression time
  - May increase / decrease storage size

- Change group compression algorithm
  - May reduce \( W \)- or return-incurred decompression time
  - May increase / decrease storage size
Choosing a compression configuration, example

C: /root/a/#text
/root/b/#text
/root/c/#text
/root/d/e/#text
/root/d/f/#text

W: //a[text()="a1"]
for $x$ in //b, $y$ in //c
where $x$/text()<$y$/text()
return <d>{$x}/d</d>

\[ H \]

a/#text
b/#text
c/#text
e/#text
f/#text
Choosing a compression configuration, example

C: /root/a/#text
   /root/b/#text
   /root/c/#text
   /root/d/e/#text
   /root/d/f/#text

W: //a[text()="a1"]
   for $x$ in //b, $y$ in //c
   where $x/text()<y/text()$
   return <d>{$x/d}</d>
Choosing a compression configuration, example

C: /root/a/#text
   /root/b/#text
   /root/c/#text
   /root/d/e/#text
   /root/d/f/#text

W: //a[text()="a1"]

for $x$ in //b, $y$ in //c
where $x/text() < y/text()$
return <d>{$x/d}</d>
Choosing a compression configuration, example

```
C: //root/a/#text
   //root/b/#text
   //root/c/#text
   //root/d/e/#text
   //root/d/f/#text

W: //a[text()="a1"]
  for $x$ in //b, $y$ in //c
  where $x/text()<$y/text()
  return <d>{$x/d}</d>
```
XML query processing based on path sequences
XQuery optimization approach

```
for $i in //samérica/item
    $k in $i//@keyword where $i//@id="item101"
return <article>
    <numero> $i//@id/value() </numero>
    <mot-cle> $k/text() </mot-cle>
</article>
```

1. Bind variables and return clause expressions
   Identify paths and containers to which they refer

2. Construct access plans:
   Scan path sequence(s)
   Scan container(s)
   Index access on container(s)

Very selective data access due to vertical fragmentation
XQuery optimization approach

```xquery
for $i in //samerica/item
    $k in $i//keyword where $i/@id="item101"
return <article>
    <numero> $i/@id/value() </numero>
    <mot-cle> $k/text() </mot-cle>
</article>
```

1. Bind variables and return clause expressions
   Identify paths and containers to which they refer
2. Construct access plans:
3. Construct QEP as if uncompressed XML
   Structural joins with two-ways skipping
4. Insert decompress operators prior to XMLConstruct
XQuery optimization approach

```xml
for $i in //samerica/item
  $k in $i//keyword where $i/@id="item101"
return <article>
  <numero> $i/@id/value() </numero>
  <mot-cle> $k/text() </mot-cle>
</article>
```

XML Construct

```
($i.ID, @id/value)
```

R

```
($i.ID [ @id="item101" ], $k.ID)
```

F-W

```
($k.ID, $k/text())
```

R
Variable path inference

**Input**: query pattern; path summary

```
is site
  $i$ item [ @ID="item0"]
  $k$ keyword
```

```
site
  asia
  $i$ item [ @ID="item0"]
  $k$ keyword
```

**Output**: *path list pList of each variable*

- `$i.pList` = { `/site/regions/africa/item`, `/site/regions/asia/item`, `/site/regions/europe/item`, `/site/regions/namerica/item`, `/site/regions/samerica/item` }
- `$k.pList` is too large 😊
Variable path inference

for $i$ in //item[location],
$p$ in $i$/listitem,
$k$ in $p$/keyword

$\text{i.pList} /site/samerica/item$

$\text{p.pList} /site/sam/item/descr/listitem$
/site/sam/item/descr/listitem/parlist/listitem$

$\text{k.pList} /site/sam/item/descr/listitem/plist/item/kwd$
Variable path inference

for $i$ in //item[location],
$p$ in $i$/listitem,
$k$ in $p$/keyword

/site/samerica/item
/site/sam/item/descr/Item/parlist/listitem
/site/sam/item/descr/Item/pList/listitem/kwds
Variable path inference

Traverse in parallel the path summary and the query
  Find the minimal set of paths for each variable
  Eliminate redundant branches
  Infer new structural relationships between variables
Implementation: query evaluation on the path summary
  1. In memory: recursive traversals
  2. On disk-resident path summary: streaming evaluation
Physical operators

Iterator interface
Flat tuple algebra

PathScan: returns all IDs on a given path
ContainerScan: returns all (ID, value) pairs on a given path
Merge: sequence fusion in document order
StackTreeDesc (structural join), outer- and semi-join variants
SortedOuterUnion and Merge for reconstruction
Other XML compressors
Other XML compressors

XMill [SIGMOD2000]: non-queriable compression
XGrind [ICDE2003]: homomorphic compression

Compressed XML document is almost an XML document

XQZip [EDBT2004]: structural summary + blocks of compressed values
XPress [SIGMOD2003]: reverse arithmetic encoding of paths
MacMill [VLDB2003]: document tree compressed by bisimulation, no value compression

All are based in memory and potentially require exhaustive traversal for querying
Performance evaluation
Plan

XQueC compression:
- Various compressed data structures
- Impact of compression configuration

XQueC storage model & query processing:
- Comparison baseline: **TP**: Tag partitioning (Natix, Timber, Niagara)

Compression performance:
- Comparison baseline: XMill (unqueryable compressor), XGrind, XPRESS, XQZip
### Some sample documents

<table>
<thead>
<tr>
<th>Document</th>
<th>Size</th>
<th>#elems</th>
<th>#paths</th>
<th>#containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeBank</td>
<td>82 Mb</td>
<td>2,437,665</td>
<td>338,748</td>
<td>220,818</td>
</tr>
<tr>
<td>INEX</td>
<td>500 Mb</td>
<td>8,091,779</td>
<td>10,478</td>
<td>12,380</td>
</tr>
<tr>
<td>XMark</td>
<td>111 Mb</td>
<td>1,666,310</td>
<td>514</td>
<td>444</td>
</tr>
<tr>
<td>SwissProt</td>
<td>112 Mb</td>
<td>2,977,030</td>
<td>117</td>
<td>191</td>
</tr>
<tr>
<td>Shakespeare</td>
<td>7 Mb</td>
<td>359,380</td>
<td>58</td>
<td>191</td>
</tr>
<tr>
<td>DBLP</td>
<td>121 Mb</td>
<td>3,332,129</td>
<td>125</td>
<td>136</td>
</tr>
<tr>
<td>NASA</td>
<td>24 Mb</td>
<td>476,645</td>
<td>95</td>
<td>70</td>
</tr>
<tr>
<td>UW course</td>
<td>3 Mb</td>
<td>84,051</td>
<td>18</td>
<td>12</td>
</tr>
</tbody>
</table>
Parenthesis: of the importance of using *varied* XML documents
End of paranthesis
Compressed data structure sizes (small docs)
Compressed data structure sizes (large docs)
Compression dictionary sizes for various compression configurations

Size (MB)

- NaiveHuffman1
- NaiveHuffman2
- NaiveHuffman3
- NaiveALM1
- NaiveALM2
- NaiveALM3
- CH
- GH
- G2H

UW course | Shakespeare | Nasa
Compressed containers size for various compression configurations

![Bar chart showing size (MB) for different compression configurations across SwissProt, Xmark111, and DBLP datasets.]
Compression dictionary sizes for various compression configurations

Actual value: 11.7Mb

Size (MB)

SwissProt  Xmark111  DBLP

NaiveHuffman1  NaiveHuffman2  NaiveHuffman3  NaiveALM1  NaiveALM2  NaiveALM3  CH  GH  G2H
Compression factor comparison with other systems

XQueC CF w.r.t. competitors (1)
Comparing compression performance

Other compressors achieve better compression rate
  XMill beats all
Queryable compressors tend to have less query functionalities
  (Some) XPath...
  "Query answer is: yes"
  "Sorry, your document has lost its whitespaces"
Extensions to capture complex queries not clear
XQueC adopts a database-biased view
Storage model evaluation

Implemented based on BerkeleyDB

<table>
<thead>
<tr>
<th></th>
<th>B+-trees</th>
<th>Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path partitioning</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Tag partitioning</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(No compression): sequence-based storage 4 times more compact than TP

No persistent tree

Sequences are compact; B+-trees better for updates

Tags are stored only once per path
Path partitioning vs. tag partitioning: does it make a difference?

Tag fan-in: number of paths ending in the tag
Median fan-in: weighted average of tags fan-in

<table>
<thead>
<tr>
<th>Document</th>
<th>Size</th>
<th>Recursive?</th>
<th>Max fan-in</th>
<th>Median fan-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeBank</td>
<td>82 Mb</td>
<td>yes</td>
<td>49.901</td>
<td>19.187</td>
</tr>
<tr>
<td>INEX</td>
<td>500 Mb</td>
<td>yes</td>
<td>1.722</td>
<td>485</td>
</tr>
<tr>
<td>XMark</td>
<td>111 Mb</td>
<td>yes</td>
<td>99</td>
<td>15</td>
</tr>
<tr>
<td>SwissProt</td>
<td>112 Mb</td>
<td>no</td>
<td>39</td>
<td>11</td>
</tr>
<tr>
<td>Shakespeare</td>
<td>7 Mb</td>
<td>no</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>DBLP</td>
<td>121 Mb</td>
<td>no</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>NASA</td>
<td>24 Mb</td>
<td>yes</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>UW course</td>
<td>3 Mb</td>
<td>no</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Query processing performance

Comparison baseline:

**TP**: Tag partitioning (Natix, Timber, Niagara)
Base storage: persistent tree
Tag-partitioned index of [pre, post, depth] IDs
**Needs sort and duplicate elimination in many cases**

**OTP**: optimized TP
Storage just like TP
**Use path inference to find which sort and dup-elim are unnecessary**
Variable binding performance

ID access time (s) for "//tag" linear path expressions
Variable binding performance

ID access time (seconds), linear path expressions of length 2

- PS
- TP
- OTP

P8, P9, P10, P11, P12, P13, P14, P15, P16, P17
Variable binding performance

ID access time (seconds), linear path expressions of length 3
Variable binding performance

ID access time (seconds), two-variable bindings

- PS
- TP

P8' P9' P10' P11' P12' P13' P14' P15' P16' P17'
Summary and perspectives
Summary

XQueC: an XML database endowed with data compression
[VLDB'03, EDBT'04, TODS in progress]
Path sequence storage: vertical fragmentation, ordered storage

Compression strategy: cost-based search for a compression configuration, taking into account several concerns

Query processing on a path sequence storage
Variable binding (i.e. logical navigation) a big winner, for any path
Perspectives

Various improvements to the code, hopefully some release as a persistent XML database

Needs faster loading, better integrated optimizer

Streaming-based path summary matching: implemented as a standalone tool (XSum)

Path summary-based query minimization
Merci