XQueC: XQuery processor and Compressor

Speaker: Angela Bonifati, Icar CNR, Italy
MailTo: bonifati@icar.cnr.it

Joint work with: A. Arion (INRIA Futurs, France),
G. Costa (DEIS, Unical, Italy)
S. D’Aguanno (INRIA Futurs, France)
I. Manolescu (INRIA Futurs, France)
A. Pugliese (DEIS, Unical, Italy)

Motivations

- XML is redundant and akin to compression
- As in the spirit of [WK00, CG01], compression influences the query evaluator, the storage manager and the query optimizer
- Properly suited compression can drastically reduce I/O accesses, CPU effort, network bandwidth, thus QET
- Compression of query results is attractive in many settings (e.g. mobile devices, distributed query processors,...)
State of the art

- **Query-unaware systems:**
  - XMill: exploits data semantics
  - XMLPPM: exploits classical PPM algorithms
- **Query-aware systems:**
  - XGrind: an “homomorphic” compressor/basic query processor
  - XPress: an “homomorphic” compressor/slightly extended query processor
- XQueC improves over previous works, by covering arbitrarily complex queries in the compressed domain
- XQueC is ongoing work:
  - Demo accepted at VLDB 2003
  - We are currently setting up an experimental evaluation session

**XQueC w.r.t. XMill**

- XMill aims to obtain high compression ratios, disregarding query capabilities
- XMill splits data into containers
- XMill uses the best semantic encoding for data types and later applies zlib
- XQueC inherits data splitting, using fixed root-to-leaf navigation
XQueC w.r.t. XGrind/XPress

- XQueC is the first query processor for real-life complex queries in the compressed domain ("lazy decompression")
- XQueC is the first query processor to use a dictionary-based order-preserving algorithm, i.e. ALM [Antoshenkov97]
- XQueC is the first query processor to devise a set of access support structures for efficient query evaluation
- None of the previous features is handled in XGrind/XPress

XQueC principles

- Associate a container to each <type, root-to-leaf PE>
- Keep the content separated from the structure by means of pointers
Disobeying “homomorphism” is not a drawback:

- consider //A/C in XGrind

\[
\begin{array}{c|c|c|c}
T1 & T2 & \text{nahuff(ciao)} & /\\
\end{array}
\]
Clever Structures

- Disobeying “homomorphism” is not a drawback:
  - in XGrind, same parsing as in decompressed data

Structure Summary

Containers

Structure Tree
Clever Structures

○ Disobeying “homomorphism” is not a drawback:
  ● XQueC keeps it simple!

Structure Summary

match(//a/c, /e/g/a/c) match(//a/c, /e/h/a/c) match(//a/c, /e/y/z/a/c)

ciao hello salut

Containers

More on Containers

○ They are kept lexicographically ordered

○ Document order can be easily reconstructed looking at the structure summary

○ Containers happen to be efficient pre-cooked indexes (resemble B+trees on value columns)

○ Binary search on containers: logarithmic cost
Talk hot points

- XQueC Architecture
- Basics on ALM
  - Early experiments
  - Occupancy of XQueC structures
- Which is the best algorithm?
  - Cost model
- XQueC storage in brief
- Future promising developments:
  - soon figures on QETs

XQueC architecture
Basics on ALM

- Dictionary-based order preserving algorithm
  - Dictionary without the prefix property
  - The same token may have different encodings

<table>
<thead>
<tr>
<th>token</th>
<th>code</th>
<th>interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>c.</td>
<td>[theaa, therd]</td>
</tr>
<tr>
<td>the</td>
<td>d</td>
<td>[theaa, thezz]</td>
</tr>
<tr>
<td>...</td>
<td>e.</td>
<td>[therf, thezz]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Occupancy of XQueC structures

![Graph showing Compression Factor (%) for different datasets: Shakespeare, Baseball, Course]
Average Compression Factor

![Bar chart showing average compression factors for XGrind, XPress, and XQueC.]

- XGrind: 67%
- XPress: 71%
- XQueC: 72%

More experiments

![Bar chart showing compression factors for xmark15 and Religion.]

- xmark15:
  - Xmill: 67%
  - XGrind: 63
  - XQueC: 50

- Religion:
  - Xmill: 60.38
  - XGrind: 67.54
  - XQueC: 73
Memory issues

- Consider a query, like:
  - FOR $c$ IN /A/B/C,
    $b$ IN /A/B
  WHERE $c$/text() > $b$/ text()
  RETURN $c$

- In XQueC, you have direct access to the corresponding containers
  - You pay the effort of fetching in memory only two containers plus (part of) the structure summary

- In XGrind and XPress, you have to fetch in memory all the XML file, keep part of it and decompress

---

Memory issues

- Consider a query, like:
  - FOR $a$ IN /A,
    $c$ IN $a$/B/C,
    $b$ IN $a$/B
  WHERE $c$/text() > $b$/ text()
  RETURN $a$

- In XQueC, you take at worst all the structure tree plus the $a$-related containers and the structure summary

- In XGrind and XPress, you have to fetch in memory all the XML file, keep part of it and decompress
Need of a cost model

- XQueC may employ the query workload W, if available:
  - To choose the best compression algorithm
  - To choose how to share the source models (SM) among containers in order to fully exploit their similarities

- Take for instance Huffman and ALM:
  - ALM supports (in)equality predicates
  - Huffman supports equality/wildcards predicates
    - /A/B contains ‘ang*’

- Mind that XGrind naively builds a separate SM for each tag name (i.e. /a/b and /c/d/b share the same SM for b’s)

Cost Model excursus

- Cost Model Input: configuration \( s(P, alg) \), partition \( P \) of the containers \( C \), available algorithms \( A \) and a function \( alg: P \rightarrow A \)
- Search space: exponential in \(|C|\)
- Cost function:
  - Combines decompression time, compressed container storage cost, source model storage cost
  - Bases on the properties of comparison predicates in \( W \)
- Search strategy:
  - Greedy with ‘moves’ in only three directions
  - Complexity: linear w.r.t. the number of predicates in \( W \), quadratic w.r.t. \(|C|\)
Examples of partitions

- ALM
- Huffman
- ZIP2

Homogeneous containers

Heterogeneous containers

Heterogeneous Containers, Unbalanced workload

XQueC Storage

- The problem of storing XML data strikes back when data is compressed
- File systems, current native (uncompressed) XML repositories are not sufficient
- We choose our own solution based on Berkeley DB, an embedded database/high-speed library for record storage
Conclusions

- XQueC uses data fragmentation for advantageous query evaluation
- Implementation is ongoing for the demo (come to visit our demo stand at VLDB 2003 in Berlin!)
- Future studies are devoted to
  - improve the query optimizer
  - refine the cost model:
    - better exploiting schema information to assess container similarities
- XQueC is online:
  http://dns.isi.cs.cnr.it/isi/bonifati/xquec/

References

Thank you!

Extra slides
**XQueC w.r.t. XGrind (details)**

- **XGrind** (ICDE 2002) is an extended SAX parser
  - Encoding is “homomorphic”
  - Uses:
    - Huffman/Arithmetic for text/numbers encoding
    - \( \log_2 K \) scheme for tags/numerated domains
  - Handles limited number of queries (Prefix/Exact Match queries) in the compressed domain
  - Inefficient evaluation of path expressions
- **XQueC** is the first that aims to cover full XQuery in an efficient way

**XQueC w.r.t. XPress (details)**

- **XPress** (Sigmod 03) extends XGrind:
  - Obey the homomorphism
  - Uses:
    - Reverse Arithmetic Encoding for path expressions (PE->an interval within \([0.0, 1.0]\))
    - Huffman for textual data
    - 4 Types of encoding for numbers
    - Scheme \( \log_2 K \) for enumerated domains
  - Handles limited number of queries in the compressed domain (Prefix/Exact Match queries, Range queries only with numbers)
- **XQueC** extends the set of queries addressable in the compressed domain:
  - We treat textual data with order-preserving ALM [Antoshenkov97]
  - We use simple binary encoding for numbers and \( \log_2 K \) scheme for tags
**ALM encoding by example**

<table>
<thead>
<tr>
<th>token</th>
<th>code</th>
<th>interval</th>
<th>token</th>
<th>code</th>
<th>interval</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the</td>
<td>c</td>
<td>[there, there]</td>
<td>ir</td>
<td>b</td>
<td>[ir, ir]</td>
</tr>
<tr>
<td>there</td>
<td>d</td>
<td>[there, there]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the</td>
<td>e</td>
<td>[therf, thezz]</td>
<td>se</td>
<td>v</td>
<td>[se, se]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>there : there</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>their : the</td>
<td>c</td>
<td>ir : ir</td>
<td>b</td>
<td>their : cb</td>
<td></td>
</tr>
<tr>
<td>these : the</td>
<td>e</td>
<td>se : se</td>
<td>v</td>
<td>these : ev</td>
<td></td>
</tr>
</tbody>
</table>

**Occupancy of our structures**

<table>
<thead>
<tr>
<th></th>
<th>Structure Tree</th>
<th>Dictionaries</th>
<th>Compressed containers</th>
<th>Structure Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literature</td>
<td>65%</td>
<td>12%</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>Course</td>
<td>49%</td>
<td>8%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Shakespeare</td>
<td>42%</td>
<td>6%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>smart 10</td>
<td>39%</td>
<td>7%</td>
<td>80%</td>
<td></td>
</tr>
<tr>
<td>smart 10</td>
<td>30%</td>
<td>6%</td>
<td>80%</td>
<td></td>
</tr>
</tbody>
</table>
XQueC Query Evaluator

- An algebra has been devised to work on XQueC structures:
  - Operators for data access
  - Operators to compress/decompress
  - Comparison operators on compressed data ($\sigma, \bowtie$)

Q1: FOR $b$ IN document(``auction.xml'')/site/people/person[ID='''person0''']
RETURN $b$/name/text()

*Diagram showing the query execution process with nodes and edges.*