Module 4

Implementation of XQuery

Part 3: Support for Streaming XML
Motivation

• XQuery used in very different environments:
  – XQuery implementations on XML stored in databases (with indexes).
  – **Main-memory XQuery implementations on XML in files, sent as streams, computed on the fly...**

• Example Applications:
  – Web Services (e.g., ActiveXML).
  – Telecommunication apps (XML messages).
  – XML documents.
  – Information Integration.
Challenges to Address

• Efficient Representation: Compression
• Matching Content/Message Brokering
• Discarding unneeded Data: Projection
Reducing the space overhead

• XML uses rather verbose syntax
  – High bandwidth overhead
  – Slow parsing speed

• Excludes usage in resource-constrained environments

• Compress XML to trade additional CPU time to storage/transfer cost
Classification of Compression

• XML knowledge
  – General Text Compression
  – Schema-dependent compression
  – Schema-independent compression

• Queryable
  – Archive-only
  – Homomorphic compression
  – Non-homomorphic compression
Compression

• Classic approaches: e.g., Lempel-Ziv, Huffman
  – decompress before queries
  – miss special opportunities to compress XML structure
  – Not Queryable at all
• XMill: Liefke & Suciu 2000
  – Idea: separate data and structure -> reduce entropy
  – separate data of different type -> reduce entropy
  – specialized compression algo for structure, data types
• Assessment
  – Very high compression rates for documents > 20 KB
  – Decompress before query processing (bad!)
  – Indexing the data not possible (or difficult)
XML Architecture

XML

Parser
Path Processor

Cont. 1
Cont. 2
Cont. 3
Cont. 4

Compr.
Compr.
Compr.
Compr.

Compressed XML
XMill Example

<book price="69.95">
  <title>Die wilde Wutz</title>
  <author>D.A.K.</author>
  <author>N.N.</author>
</book>

– Dictionary Compression for Tags:
  book = #1, @price = #2, title = #3, author = #4

– Containers for data types:
  ints in C1, strings in C2

– Encode structure (/ for end tags) - skeleton:
  gzip( #1 #2 C1 #3 C2 / #4 C2 / #4 C2 / / )
Querying Compressed Data
(Buneman, Grohe & Koch 2003)

- Idea:
  - extend Xmill
  - special compression of skeleton
  - lower compression rates,
  - but no decompression for XPath expressions
Compression

• XML-aware compressors outperform text compressors
• Queryable compressors show worse compression than archival
• Not much adoption outside research
• Binary XML
  – picks up many compression ideas
  – Now a W3C standard: EXI
Content Matching: XML Message Brokering

Filtering
Transformation
Routing
Message-based Middleware

• Publish/Subscribe
  – Subscribers express interests, later notified of relevant data from publishers.
  – Loose coupling at the communication level.

• XML, a de facto standard for online data exchange
  – Flexible, extensible, self-describing.
  – Enhanced functionality: XSLT, XQuery, ...
  – Loose coupling at the content level.

• XML message brokering
  – Publish/subscribe + XML = flexibility at communication and content levels.
  – Declarative XML queries provide high functionality.
New Applications

• Message brokering supports a large number of emerging distributed applications:
  – Application integration
  – Personalized newspaper generation
  – Stock tickers
  – Network monitoring
  – Mobile services
  …
Problem Statement

Inputs:
   (1) continuously arriving XML messages (usually small)
   (2) a set of XQuery queries representing client interests

Main functions of an XML message broker:
   – **Filtering**: matches messages to query predicates.
   – **Transformation**: restructures the matching messages.
   – **Routing**: directs messages to queries over a network of brokers.

Challenges: providing this functionality for
   – *large numbers of queries* (e.g., 10’s thousands of them)
   – *high volumes of XML messages* (e.g., tens or hundreds/sec)
Design Space

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Subject-based</th>
<th>Predicate-based</th>
<th>XML filtering</th>
<th>XML filtering &amp; transformation</th>
<th>Expressiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>TIBCO MQ Pub/Sub, JMS Pub/Sub</td>
<td>Siena Gryphon, xmlBlaster, Snoeren et al. [SOSP01]</td>
<td>ONYX [VLDB04]</td>
<td>YFilter [VLDB03]</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>Oracle Advanced Queuing, Le Subscribe, XFilter, XTrie, IndexFilter, XMLTK</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Subject = “Stock”

Yes No

(a1, v1)
(a2, v2)
(a3, v3)
...
(an, vn)

Yes No

Yes No

Yes No
YFilter & ONYX

- **YFilter**, a system for XML filtering and transformation.
- **Filtering** exploiting sharing:
  - Order-of-magnitude performance benefits over previous work.
  - Scalable to 100’s thousands of distinct queries.
  - YFilter 1.0 release: used in research projects and product development, being integrated into Apache Hermes for WS-Notification.
- **Transformation** exploiting sharing:
  - The first algorithm for transformation for a large set of queries.
  - Scalable up to 10’s of thousands of distinct queries.
- **Routing (ONYX)**: an overlay network of brokers with routing abilities, providing flexible, Internet-scale XML dissemination services.
The Filtering Problem

• Full XPath/XQuery too expensive 😞

• Query language: path expression =
  \((('/' | '//' \text{ElementName} | '*) \text{Predicate}*)^+\)

• The filtering problem:
  – Given (1) a set \(Q = Q_1, \ldots, Q_n\) of path queries, where each \(Q_i\) has an associated query identifier, and (2) a stream of XML documents.
  – Compute, for each document \(D\), the set of query identifiers corresponding to the XPath queries that match \(D\).
Constructing an FSM for a Query

Key Idea: represent query paths as state machine that are driven by the XML parser (SAX)

- Simple paths: ( ("/" | "//") (ElementName | "*") )+
- A finite state machine (FSM) for each path: mapping steps to machine states.

Map location steps to FSM fragments.

<table>
<thead>
<tr>
<th>Location steps</th>
<th>FSM fragments</th>
</tr>
</thead>
<tbody>
<tr>
<td>/a</td>
<td>![Diagram of FSM fragment for /a]</td>
</tr>
<tr>
<td>/*</td>
<td>![Diagram of FSM fragment for */]</td>
</tr>
<tr>
<td>//a</td>
<td>![Diagram of FSM fragment for //a]</td>
</tr>
</tbody>
</table>

Concatenate FSM fragments for location steps in a query.

Query "/a//b"

![Diagram of concatenated FSM fragments for query "/a//b"]
YFilter builds a single combined FSM for all paths!

- Complete prefix sharing among paths.
- Nondeterministic Finite Automaton (NFA)-based implementation: a small machine size, flexible, easy to maintain, etc.
- Output function (Moore machine): accepting states → partition of query ids.

Q1=/a/b   Q5=/a/*/b
Q2=/a/c   Q6=/a//c
Q3=/a/b/c  Q7=/a/*/*/c
Q4=/a//b/c Q8=/a/b/c
YFilter uses a stack mechanism to handle XML
• Backtracking in the NFA.
• No repeated work for the same element!
DFA vs. NFA

• DFA has exponential number of states
  – Large main-memory requirements
  – Or I/O needed in order to process messages

• DFA has high maintenance costs
  – Need to rerun Myhill/Büchi algorithm, everytime a new profile is posted or deleted

• NFA is slower than DFA

• NFA: entries in stack can grow exponentially
  – In practice, XML documents are fairly flat

• NFA is the clear winner (current trade-offs)!
Performance results for YFilter

- YFilter scales to 150,000 distinct path queries w/o predicates.
  - Consistently takes 30 msec or less.
  - Achieves a 25x performance improvement over previous approaches.
- Deep element nesting: No exponential blow-up of active states.
- Sensitivity to ‘*’ and ‘//’: Little, due to effective prefix sharing.
- NFA maintenance for query updates: Tens of milliseconds for inserting 1000 queries.
- YFilter handles 100’s thousands of queries with predicates.
  - No real competition before
  - Mechanism not shown here. What are the difficulties?
XML Projection
Memory Limitations

• Main-memory XQuery implementations cannot handle large documents.
• Complex XQuery expressions require materialization (DOM).
• DOM is the bottleneck.

<table>
<thead>
<tr>
<th>XQuery Processors</th>
<th>Maximum Document Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quip</td>
<td>7Mb</td>
</tr>
<tr>
<td>Kweelt</td>
<td>17Mb</td>
</tr>
<tr>
<td>Galax</td>
<td>33Mb</td>
</tr>
<tr>
<td>Xalan (XSLT)</td>
<td>75Mb</td>
</tr>
</tbody>
</table>

XMark Query 1 on an IBM laptop T23 (256Mb RAM)
Projection: Example

<site>
<regions>...
<people>
...
<person id="person120">
  <name>Wagar Bougaut</name>
  <emailaddress>mailto:Bougaut@wgt.edu</emailaddress>
</person>
<person id="person121">
  <name>Waheed Rando</name>
  <emailaddress>mailto:Rando@pitt.edu</emailaddress>
  <address>
    <street>32 Mallela St</street>
    <city>Tucson</city>
    <country>United States</country>
    <zipcode>37</zipcode>
  </address>
  <creditcard>7486 5185 1962 7735</creditcard>
  <profile income="59224.09">
  ...

XMark Query 1
for $b in /site/people/person[@id="person0"]
return $b/name

Less than 2% of original document!
Projection: Intuition

• Given a query:
  
  For $b$ in /site/people/person[@id="person0"]
  Return $b/name$

  – Most nodes in the input document(s) are not required.
  – Projection operation removes unnecessary nodes.
  – Evaluation of the query on projected document yields the same results as on the original document.

• How it works:
  – Projection defined by set of paths.
  – Static analysis infers sets of paths used within a query.

/site/people/person
/site/people/person/@id
/site/people/person/name
Projection: Challenges

• For an XQuery expression, compute all paths that allow to reach nodes required to evaluate the expression.

• XQuery is complex:
  – Variables
  – Composition
  – Syntactic Sugar
  – Complex expressions

• Have to be able to analyze all of XQuery.
XML Projection

• Similar to relational projection:
  – One key operation.
  – Prunes unnecessary part of the data.
  – Essential for memory management.

• Specific problems related to XML:
  – Projection must operate on trees.
  – Requires analysis of the query.
  – Need to address XQuery complexity.
Notation

• Projection Paths:
  – Path expressions are noted using XPath semantics
    (/site/people/person/@id)
  – “#” notation used when subtree should be kept
    (/site/people/person/name#)

• Static Analysis: inference rule notation
  \[ \text{Expr} \rightarrow \text{Paths} \]
Static Analysis: Variables

• Variables can be bound to nodes coming from different paths.
  
  for $b$ in /site/people/(teacher | student)
  return $b/name$

• Analysis must remember paths to which variable was bound
  /site/people/teacher
  /site/people/student

• Environment is maintained during path analysis:
  $Env | - Expr => Paths$
Static Analysis: Example

- Literals do not require any paths:

\[
Env |- Literal \Rightarrow \{\}
\]

32 \Rightarrow \{

- Paths are propagated in a sequence:

\[
\begin{align*}
Env & |- \text{Expr1} \Rightarrow \text{Paths1} \\
Env & |- \text{Expr2} \Rightarrow \text{Paths2} \\
Env & |- \text{Expr1,Expr2} \Rightarrow \text{Paths1 \cup Paths2}
\end{align*}
\]

\[
\begin{align*}
/a/b & \Rightarrow \{/a/b\} \\
/a/d & \Rightarrow \{/a/d\} \\
/a/b,/a/d & \Rightarrow \{/a/b,/a/d\}
\end{align*}
\]
(if (count (/site/regions/*) = 3)
  then /site/people/person
  else /site/open_auctions/open_auction)/@id

- @id does not apply to /site/regions/*
- Final set of paths should be
  /site/regions/*
  /site/people/person/@id
  /site/open_auctions/open_auction/@id

- Need to differentiate two sets of paths during analysis:
  - Returned Paths: returned by the expression, further path steps are applied on them.
  - Used Paths: used to compute the expression.

\[ Env \vdash \text{Expr} \Rightarrow \text{Paths} \text{ using UPaths} \]
XQuery Processing Architecture

- XQuery Expression
- XQuery Parser
- XQuery Abstract Syntax tree
- Path Analysis
- Projected Data Model
- XML Query Result
- XML Data Model
- Document Data Model
- XML Data Model Loader
- SAX Events
- SAX Parser
- Input XML Document
Loading Algorithm: Description

• Input:
  – Set of projection paths.
  – Document SAX events.

• Decide on action to apply on document nodes:
  – Skip: ignore node and its subtree.
  – KeepSubtree: keep node and its subtree.
  – Keep: keep node without its subtree.
  – Move: keep processing SAX events. Current node is only kept if some of its children are kept.

• Keep a set of current paths.
Loading Algorithm: Example

Projection Paths:
/a/b/c#
/a/d

Loaded Nodes:

Current Paths:
Loaded Nodes:
/b/c#
/d

Action:
Move
Skip
/c#
Keep Subtree
b
d
/a/b/c#
/a/d

Similar to XML filtering algorithms

Limitations:
- Backward Axis!
- Number of current paths can be huge (descendant axis)
Experiments: Settings

• XML Projection Evaluation:
  – **Effectiveness**: projection impact on different queries.
  – **Maximum document size**: largest document that can be processed.
  – **Processing time**: effect on processing time.

• Experimental Setup:
  – Default XMark document size: 50Mb.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>CPU</th>
<th>Cache Size</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1GHz</td>
<td>256Kb</td>
<td>256Mb</td>
</tr>
<tr>
<td>B</td>
<td>550MHz</td>
<td>512Kb</td>
<td>768Mb</td>
</tr>
<tr>
<td>C (default)</td>
<td>1.4GHz</td>
<td>256Kb</td>
<td>2Gb</td>
</tr>
</tbody>
</table>
All queries but one require less than 5% of the document.
# Experiments: Maximal Document Size

<table>
<thead>
<tr>
<th>Configuration</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMark Query 3 (simple selection with predicate)</td>
<td>No Projection</td>
<td>33Mb</td>
<td>220Mb</td>
</tr>
<tr>
<td></td>
<td>Optimized Projection</td>
<td>1Gb</td>
<td>1.5Gb</td>
</tr>
<tr>
<td>XMark Query 14 (Non-selective path query with predicates)</td>
<td>No Projection</td>
<td>20Mb</td>
<td>20Mb</td>
</tr>
<tr>
<td></td>
<td>Optimized Projection</td>
<td>100Mb</td>
<td>100Mb</td>
</tr>
<tr>
<td>XMark Query 15 (Long, very selective path query)</td>
<td>No Projection</td>
<td>33Mb</td>
<td>220Mb</td>
</tr>
<tr>
<td></td>
<td>Optimized Projection</td>
<td>1Gb</td>
<td>2Gb</td>
</tr>
</tbody>
</table>
Experiments: Query Execution Time

Projection significantly reduces query processing time
Next Bottleneck: Joins!
Hardware-based Projection

• Projection effective to reduce memory consumption, document processing cost
• Still bound by XML parsing speed
  – Best parsers on modern CPUs: 10-30 MB/s
• How can we do better:
  – Hardware/Software Co-Design!
  – Run Projection on an FPGA!
  – Parse and project on wire speed!
1. Extract Projection Path, load into FPGA
2. Request XML document
3. Send (regular) XML to FPGA
   Receive filtered XML from FPGA
FPGAs

• Field-Programmable Gate Arrays
• Reconfigurable Hardware
  – Memory
  – Logic Gates
  – Wires
• Massive parallelism possible
• „Create“ custom processor
• Slow to reprogram
Projection Processing on FPGAs

- FPGA very efficient in running automata
  \[\Rightarrow\text{Use automata-based path processing (see before)}\]
- Reprogramming Slow
  \[\Rightarrow\text{Provide general „skeleton“ path processor}\]
  \[\Rightarrow\text{Instantiate for specific projection paths}\]
Evaluation/Demo Setup

- Use FPGA boards with 1GB Ethernet
- Send XML document over network using UDP
- Run stock MXQuery with UDP receiver
Performance Results

- Performance gains of 1-2 orders of magnitude
- Many queries close to network limit
- Q15 slowed down by Gigabit Ethernet!