Tree-Pattern Queries on a Lightweight XML Processor

MIRELLA M. MORO
Zografoula Vagena
Vassilis J. Tsotras

Research partially supported by CAPES, NSF grant IIS 0339032, UC Micro, and Lotus Interworks
Outline

- Motivation and Contributions
- Background
- Method Categorization
- Experimental Evaluation
- Conclusions
Motivation

- XML query languages: selection on both value and structure
  - “Tree-pattern” queries (TPQ) very common in XML
- Many promising holistic solutions
- None in lightweight XML engines
  - Without optimization module (e.g. eXist, Galax)
  - → Effective, robust processing method
- Reasons:
  - No systematic comparison of query methods under a common storage model
  - No integration of all methods under such storage model
- Context: XPath semantics, stored data (indexed at will)
Contributions

- TPQ methods over unified environment
- Method Categorization: data access patterns and matching algorithm
- Common storage model + integration of all methods
  - Capture the access features
  - Permit clustering data with off-the-shelf access methods (e.g. B^+tree)
- Novel variations of methods using index structures + Handle TPQ
- Extensive comparative study
  - Synthetic, benchmark and real datasets
  - Decision in the applicability, robustness and efficiency
XML database = forest of unranked, ordered, node-labeled trees, one tree per document
Common Storage Model

- Input = sequence (list) of elements
- One list per document tag = *element list*
  - Node clustering by index structures
- Numbering scheme

B+ Tree on ( tag, initial )

**UC Riverside**

**Tree-Pattern Queries on a Lightweight XML Processor**
Method Categorization

- Parameters: access pattern and matching algorithm
  - (1) set based techniques
  - (2) query driven
  - (3) input driven
  - (4) structural summaries
### Cat 1: Set-based Techniques

<table>
<thead>
<tr>
<th>Access Pattern</th>
<th>Matching Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted/indexed</td>
<td>Join sets, merge individual paths</td>
</tr>
</tbody>
</table>

- **Input**: sequences of elements, one list per query node element, possibly indexed (set-based)
- **Major representative**: TwigStack
  - Optimal XML pattern matching algorithm (ancestor/descendant)
- **Stack-based processing**
  - Set of stacks = compact encoding of partial and total results in linear space (possibly exponential number of answers)
TwigStack + Indexes

- B+tree, built on the left attribute
  - From ancestor: probe descendants: skip initial nodes
  - Ancestor skipping not effective (up to 1st element that follows)
- XB-tree: on (left,right) bounding segment
- XR-tree: on (left,right), B+tree with complex index key + stab lists

- A comparative study* shows that
  - Skipping ancestors: XBTree better (XBTree size is smaller)
  - Recursive level of ancestors: XBTree better again
    - Searching on stab lists of XR-tree is less efficient
  - Plain B+tree: skips descendants, BUT not ancestors
  - XBTwigStack is our choice

Cat 2: Query Driven Techniques

<table>
<thead>
<tr>
<th>Access Pattern</th>
<th>Matching Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed/random</td>
<td>Incremental construction of each result instance</td>
</tr>
</tbody>
</table>

- Processing: the query defines the way input is probed
- Major representatives: ViST and PRIX
- Specific details: significantly different
- Same strategy
  - Convert both document and query to sequences
  - Processing query = subsequence matching
ViST and PRIX

- Recursively identify matches = quadratic time
- Optimize the naïve solution:
  - Identify candidate nodes for each matching step
  - Index structures to cluster those candidates
- Subsequence matching process = a plan consisting of INLJ among relations, each of which groups document nodes with the same label
- For a given query, joins sequence \textit{statically} defined by the sequencing of the query
- INLJ plans are a superset of the static plans that PRIX and VIST use
### ViST x PRIX x INLJ

<table>
<thead>
<tr>
<th>Dataset #nodes</th>
<th>VIST</th>
<th>PRIX</th>
<th>INLJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>LEAVES: 80%</td>
<td>100</td>
<td>84.23</td>
<td>84.20</td>
</tr>
<tr>
<td>LEAVES: 1%</td>
<td>100</td>
<td>1.33</td>
<td>1.32</td>
</tr>
<tr>
<td>ROOT: 80%</td>
<td>84.22</td>
<td>100</td>
<td>84.18</td>
</tr>
<tr>
<td>ROOT: 1%</td>
<td>1.33</td>
<td>100</td>
<td>1.33</td>
</tr>
<tr>
<td>INTERNAL: 80%</td>
<td>89.48</td>
<td>89.49</td>
<td>84.20</td>
</tr>
<tr>
<td>INTERNAL: 1%</td>
<td>34.24</td>
<td>34.22</td>
<td>1.64</td>
</tr>
</tbody>
</table>

- Percentage of nodes processed by each algorithm
- INLJ: best plan
INLJ : improved B⁺-tree

- TPQ → evaluation of relational plan
- Independence of the ordered XML model
- Total avoidance of false positives

Consider b//c
Starting from c

b elem. list
Cat 3: Input Driven Techniques

<table>
<thead>
<tr>
<th>Access Pattern</th>
<th>Matching Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Input drives computation, merge individual paths</td>
</tr>
</tbody>
</table>

- **Processing**: at each point, the flow of computation is guided entirely by the input through a Finite State Machine (DFA/NFA)
- **Advantages**
  - Each node processed only once
  - Simplicity, sequential access pattern
- **Problem**: skipping elements
SingleDFA and IdxDFA

- **SingleDFA**
  - `<element>` triggers the DFA, choosing next state
  - `</element>`: execution backtracks to when start processed
  - TPQ matching: intermediate results compacted on stacks

- **Experiments show reading whole input = not enough**

- **Speeding up navigation: IdxDFA**
  - Instead of reading sequentially: use indexes and skip descendants
IdxDFA: example
IdxDFA: example
Cat 4: Graph Summary Evaluation

<table>
<thead>
<tr>
<th>Access Pattern</th>
<th>Matching Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed/Random</td>
<td>Merge-join partitioned input, merge individual paths</td>
</tr>
</tbody>
</table>

- **Structural summary**: index node identifies a group of nodes in the document
- **Processing**: identify index nodes that satisfy the query + post processing filtering
- **Beneficial**: when there is a reasonable structural index, much smaller than document
- **Problem**: graph size comparable/larger than original document
## Categories Summary

<table>
<thead>
<tr>
<th></th>
<th>Access Pattern</th>
<th>Matching Process</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set Based</strong></td>
<td>Sorted/ Indexed</td>
<td>Join sets, merge individual paths</td>
<td>Twigstack /XB, B⁺tree, XR-tree</td>
</tr>
<tr>
<td><strong>Query Driven</strong></td>
<td>Indexed/random</td>
<td>Incremental construction of each result instance</td>
<td>(ViST, PRIX) INLJ</td>
</tr>
<tr>
<td><strong>Input Driven</strong></td>
<td>Sequential</td>
<td>Input drives computation, merge individual paths</td>
<td>SingleDFA, IdxDFA</td>
</tr>
<tr>
<td><strong>Structural Summary</strong></td>
<td>Indexed/random</td>
<td>Merge-join partitioned input, merge individual paths</td>
<td>Structural indexes</td>
</tr>
</tbody>
</table>
Experimental Evaluation

1. Experiments with real datasets
2. Experiments with synthetic datasets
   - Further analyze each method
   - Characterize the methods according to specific features available in each custom dataset
3. More sets of experiments
   - Closely verify XBTWIGSTACK versus INLJ
Setup

- Algorithms using the same API
- Analysis varying structure and selectivity
- **Performance measure** = total time required to compute a query
  - Number of nodes as secondary information
- Intel Pentium 4 2.6GHz, 1Gb ram
- **Berkeley DB**: 100 buffers, page size 8Kb, B+ tree
- Real/benchmark datasets
  - XMark (Internet auction, 1.4 GB raw data, ± 17 million nodes), Protein Sequence Database
XMark

![Graph showing time (sec) for different queries](image)

- XBTwigStack
- SingleDFA
- IdxDFA
- INLJ
- StrIdx

Queries:
- X1
- X2
- X4
- X6
Custom Data

- Goal: isolate important features
- Query //a//b[.//c]//d
  - Simple enough for detailed investigation
  - Complex enough to provide large number of different data access possibilities
- Vary selectivity of each element separately
- Add recursion to key elements (root, leaf)
Custom Data

![Graph showing time (sec) vs. selectivity for different datasets and algorithms]

- **XBTwigStack**
- **IdxDFA**
- **INLJ**

**Dataset: Selectivity**
- D1: 100
- D2: 80
- D3: 50
- D4: 10
- D5: 01
Custom Data

Dataset: Selectivity

Time (sec)

XBTwigStack
IdxDFA
INLJ
On large dataset, 40mi nodes, 1Gb, 1% selectivity
- Difference of 40s between XBTwig and INLJ best plan
XBTwigStack x INLJ

R1 R3 R4 R7 R9

ABCD ABDC BACD BADC BCAD BDCA BDCA CBAD CBDA DBAC DBCA XBTWIG
Conclusions

- Categorization of TPQ processing algorithms
- Adaptations for processing TPQ
  - DFA + accessing nodes from B^+tree
  - INLJ + ancestor skipping
- DFA-based improved, IdxDFA, not enough
- Structural summary available and smaller than document: StrIdx
- XBTwigStack: most robust and predictable
  - INLJ when high selectivity: no guarantee about chosen plan without optimizer module
Questions?