Efficient Mining of XML Query Patterns for Caching

Liang Huai Yang  Mong Li Lee  Wynne Hsu
School of Computing
National University of Singapore

Presenter: Liang Huai Yang
Outline

• Motivations
• QPT Mining Problem
• Our Approach- FastXMiner
  – Candidate Generation
  – Frequency Counting
• Performance Study
• Related Work
• Conclusion
Motivations

• XML query caching
  – Caching hot queries
• View selection
  – Finding user preference for
    • Data warehousing
    • Data integration
• Automate DB tuning or administration
  – Select index for XML data
  – Guide storage
QPT Mining Problem

• Query Pattern Tree (QPT)
  – Model XPath, XQuery
  – Special tags “*” and “//”

• Rooted Subtree (RST)
  – Distinguished root
  – Preserve parent-child relationship
  – k-edge RST^k

(a) a QPT

(b) A Rooted Subtree

"XML DB"
QPT Mining Problem

• Partial order for label matching:
  – Given two labels $x$ and $x'$, if $x = x'$, then $x \leq x'$
  – For any label $x \in \text{tagSet}$, define $x \leq * \leq //$

• Extended tree inclusion
  – A tree $T = <V, E>$ is *included* in another tree $T' = <V', E'>$ if there exists a mapping $\varphi$ such that
    • $\text{Root}(T') = \varphi (\text{Root}(T))$ and $\forall v \in V, \exists v' \in V's.t. v' = \varphi (v)$ where $v.label = v'.label$
    • $\varphi$ preserves the “parent-child” relation,
      – *↔1edge
      – //↔subpath
QPT Mining Problem

- Example of tree inclusion
QPT Mining Problem

• Tree inclusion
  – Difficult
    • “a/b/b” is included in “a/b//c”?
    • Recursion handling
  – Expensive

• Some simplifications
  – QPTs expansion
  – No sibling repetitions
QPT Mining Problem

- **Frequent Query Pattern Trees**
  - Given a query pattern tree database $D=\{QPT_1, \ldots, QPT_N\}$, and $0<\sigma \leq 1$ called the minimum support, find all $\sigma$-frequent rooted subtrees.
Candidate Generation

- Right most branch expansion
  - Non-Schema-guided enumeration
Candidate Generation

- Schema-guided enumeration
  - Global query pattern tree (GQPT)

- Example
Candidate Generation

- Equivalence class
  - $R_{\text{prefix}}$: two rst share the same prefix
  - $EC = \{rst \mid rst \text{ with same prefix}\}$

Prefix = “1,2,-1,3”

- Arrange rst of EC in ascending order
Candidate Generation

- Join of two RSTs
  \[ RST_1^k \text{ and } RST_2^k \]

- Example
Candidate Generation

- EC partitioning

- Advantage
  - Efficient RST generation:
    - For join part, simple suffix comparison
    - For rmlne: no tree traverse, only expand last node
Candidate Generation

• Example
Frequency Counting

- Naïve approach—XQPMiner
  - Match each RST against QPTs
- FastXMiner
  - Optimizations
    - Tree encoding
    - Associate TIDs with each QPT
    - Each RST keeps TID list for some time
Frequency Counting

• Avoid tree inclusion tests
  – For join part: if $RST_{ij}^{k+1} = RST_i^k \join RST_j^k$ then compute $RST^{k+1}.TIDList = RST_i^k.TIDList \cap RST_j^k.TIDList$
  – For rmlne part: if $RST_{ij}^{k+1}$ is a multi-branch RST, then it is a join of two $k$-edge RSTs.
  – Only single-branch RST need tree inclusion test!

• Pruning strategy

  If $k+1$-edge RST is frequent, then all its $k$-edge RSTs must be frequent
Performance Study

• Two sets of experiments:
  – Performance of FastXMiner
    • In C++
  – Preliminary results of caching
    • In Java

• Pentium IV 2.4 GHz, Windows XP, 1 GB RAM.
# Performance Study - FastXMiner

## Properties of datasets

<table>
<thead>
<tr>
<th></th>
<th>Datasets</th>
<th>DBLP</th>
<th>DBLP - Uniform</th>
<th>SSPlay</th>
<th>Sigmod Record</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>G-QPT</strong></td>
<td>Num. of nodes</td>
<td>98</td>
<td>98</td>
<td>67</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Max depth</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Num. of //</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Max fanout</td>
<td>12</td>
<td>12</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td><strong>QPT in DB</strong></td>
<td>Ave # of nodes</td>
<td>7.4</td>
<td>9.2</td>
<td>7.5</td>
<td>5.5</td>
</tr>
<tr>
<td></td>
<td>Max depth</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Max fanout</td>
<td>12</td>
<td>12</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>
Performance Study - FastXMiner

- Effect of Varying Minimum Support

(a) SigmodRecord, 200K, Zipf

(b) SSPlay, 200K, Zipf
Performance Study - FastXMiner

• Effect of Varying G-QPT Size
Performance Study - FastXMiner

- Effect of Varying Number of QPTs

![Graph showing the performance of FastXMiner and XQPMiner with varying number of QPTs in DB. The DBLP minsupp is set to 0.5%, and the data follows a Zipf distribution.]
Performance Study - Caching

- XML caching system
  - DBLP data 81.2MB
  - Use SQL Server 2000 simulate XML index
  - Querying execution method: structural join
- Caching policy: replace the infrequent first
  - LRU vs FQPT_LRU
  - MRU vs FQPT_MRU
Performance Study - Caching

- Generate XML queries

<table>
<thead>
<tr>
<th>Probabilities of Queries.</th>
<th>Query types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probability</td>
<td></td>
</tr>
<tr>
<td>0.04</td>
<td>Infrequent query patterns</td>
</tr>
<tr>
<td>0.16</td>
<td>6 group of frequent query patterns</td>
</tr>
</tbody>
</table>

- For each query
  - Assign a different probability to different elements for different groups:
    E.g. Query template1: Dblp(1), inproceedings(1), key(0.1),
         author(0.7), title(0.9), year(1), pages(0.5), crossref(0.4),
         booktitle(0.7), ee(0.5), url(0.2), review(0.9), comments(0.3)
Performance Study - Caching

• Performance metrics
  – Average response time
  – Cost saving ratio

\[
\frac{\sum H_i C_i}{\sum R_i C_i}
\]

• \( H_i \) is the hit rate of \( Q_i \) in the cache
• \( R_i \) is the total occurrences of \( Q_i \)
• \( C_i \) is the cost of execution \( Q_i \)
Performance Study - Caching

- Effect of Varying Number of Queries

(a) Average Response Time
   (Cache Size 40MB)

(b) Cost Saving Ratio
   (Cache Size 40MB)
Related Work

  – A candidate is a tree-expression represented by a k-sequence (p₁,…,pₖ)
  – No search for all frequent subtrees

• M. Zaki. Efficiently Mining Frequent Trees in a Forest. ACM SIGKDD, 2002.
  – Subtree inclusion preserves the ancestor-descendent relation

  – Subtree inclusion
    • Exact match
    • Preserves the parent-child relation
Conclusion

• Contributions
  – FastXMiner
  – Advantage of frequent QPTs in caching

• Future work
  – Relax some assumptions
  – Adapt FastXMiner to stream scenario
  – Considering attribute values, e.g. predicates