Efficient Structural Joins on Indexed XML Documents

Shu-Yao Chien
Zografoula Vagena
Donghui Zhang
Vassilis J. Tsotras
Carlo Zaniolo

Presented By: Samuel R. Collins
scollins@cc.gatech.edu
What’s it all about?

- This paper proposes efficient structural join algorithms in the presence of tag indices
  - B+-tree based structural join algorithms
  - Introduce the utilization of sibling pointers to improve performance
  - Comparison with R-tree algorithm presented
Agenda

- Querying XML Documents
- Previous Work
  - Node Numbering Schemes
  - Structural Joins
  - XML Indexing
- Structural Joins Using B+-Trees
- Performance Analysis
- Summary
Querying XML Documents

- Combine selections on element contents and structural relationships (path expressions) between tagged elements
- Example Query:
  `section[title="overview"]//figure[caption="R-tree"]`
  finds all figures with caption="R-tree" under sections whose title is “overview”
Querying XML Documents

- Traditional indexing schemes, such as B+-trees, can be extended to support value based queries
- Path expression queries pose a much harder problem
  - Require computation of structural joins
Querying XML Documents

- **Structural joins** are used to find all pairs of elements satisfying the primitive structural relationships specified in a query.

  - *parent-child* relationship
    - Example: `section/figure`

  - *ancestor-descendant* relationship
    - Example: `section//figure`
Querying XML Documents

- Efficient support for structural joins is therefore needed for efficient implementation of XML queries
  - Quickly determine structural relationship among any pair of tree nodes
  - Efficiently find all occurrences of a structural relationship
Previous Work

- Numbering Schemes
  - Allows faster determination of structural relationships if embedded on the document’s tree
  - One approach assigns *preorder* and *postorder* ranks as well as *level* in the XML tree
    - Affected by document updates: Node ranks change when inserting and deleting nodes
  - Another approach assigns *(start, end)* interval
    - Durable
Durable Numbering Scheme

Figure 1: A sample XML document.
Previous Work

- Structural Joins
  - Can take advantage of numbering schemes to determine all pairs of ancestor-descendants
  - Considered core operations in optimizing XML queries
  - Various techniques have been proposed
    - Relational DBMS
    - Native XML Query Engines
    - Stack-Tree-Desc algorithm represents state-of-the-art in structural joins
Previous Work

- Indexing XML Data
  - Techniques have been proposed that do not facilitate a numbering scheme. These works create labeled directed graphs.
  - Unlike a schema they are not static, and thus may change with an update.
  - Recent proposed node numbering schemes use B+-trees and R-trees to capture XML document structures.
Structural Joins (B+-Trees)

- Authors propose structural join algorithm using B+-tree and Node Numbering
  - Consider a single large document
  - Concentrate on the *ancestor-dependant* join
  - Assume that a separate index is used to cluster elements from the same tag
  - In practice, multiple indices can be combined by adding the tag name in the search key 
    *(tag,start)*, See figure 2.
Structural Joins (B+-Trees)

Figure 2: The B+-tree corresponding to the XML document of figure 1.
Why Index for Structural Joins?

Figure 3: Motivation for using the B+-tree index.
Algorithm Anc_Desc_B+

1. Let $a$, $d$ be the first elements of $A$ and $D$;
2. while ( not at the end of $A$ or $D$ ) do
3.     if ( $a$ is an ancestor of $d$ ) then
4.         Locate all elements in $A$ that are ancestors of $d$ and push them into stack;
5.     Let $a$ be the last element pushed;
6.     Output $d$ as a descendant of all elements in stack;
7.     Let $d$ be the next element in $D$;
8.     else if ( $a.end < d.start$ ) then
9.         Pop all stack elements which are before $d$;
10.    Let $l$ be the last element popped;
11.    Let $a$ be the element in $A$ (locate using $B+$-tree) having the smallest $start$ that is larger than $l.end$;
12.    else /* $a$ is after $d$, or $a$ is a descendant of $d$ */
13.         Output $d$ as a descendant of all elements in stack;
14.    if ( ancestor stack is empty ) then
15.         Let $d$ be the element in $D$ (locate using $B+$-tree) having the smallest $start$ that is larger than $a.start$;
16.    else
17.         Let $d$ be the next element in $D$;
18.         endif
19.     endif
20. endwhile
Embedding Containment Forest

- Enhancement to B+-Tree to improve performance
- Each element corresponds to a node in the structure and is linked to other elements from the same tag
  - Parent, first-child, sibling pointers
Embedding Containment Forest

Figure 5: C-forests on tags A and D for the XML document of Figure 1.
Embedding Containment Forest

- Properties
  - The (start,end) interval of each node contains all intervals in its subtree
  - Start numbers in the forest follow a preorder traversal
  - The start (end) numbers of sibling nodes are in increasing order

- Embedding C-forest for a given tag can be accomplished by adding the C-forest parent and next-sibling pointers amongst leaf records of B+-tree
  - Improves algorithm. B+-tree traversal is avoided
Performance Analysis

- CPU time and number of I/O's are used to measure performance

<table>
<thead>
<tr>
<th>Notation:</th>
<th>Meaning</th>
<th>Section:</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_index</td>
<td>structural join using sequential scan (Stack-Tree-Desc [1])</td>
<td>2</td>
</tr>
<tr>
<td>$B+$</td>
<td>structural join using B+ tree indices (Anc_Des.B+)</td>
<td>3</td>
</tr>
<tr>
<td>$B+sp$</td>
<td>structural join using B+ trees with sibling pointers (Anc_Des.B+sp)</td>
<td>3.1</td>
</tr>
<tr>
<td>$B+p_{sp}$</td>
<td>structural join using B+ trees and partial list of sibling pointers</td>
<td>6.2</td>
</tr>
<tr>
<td>$R^*$</td>
<td>structural join using $R^*$-trees with 1-dimensional intervals</td>
<td>4</td>
</tr>
<tr>
<td>$R^*2$</td>
<td>structural join using $R^*$-trees with 2-dimensional points</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1: Implemented Algorithms.
Performance Analysis

<table>
<thead>
<tr>
<th>Join Ancestors</th>
<th>no_index</th>
<th>B+</th>
<th>B+ psp</th>
<th>B+ sp</th>
<th>R*</th>
<th>R*2</th>
</tr>
</thead>
<tbody>
<tr>
<td>90%</td>
<td>182</td>
<td>180</td>
<td>180</td>
<td>190</td>
<td>230</td>
<td>228</td>
</tr>
<tr>
<td>70%</td>
<td>150</td>
<td>149</td>
<td>150</td>
<td>155</td>
<td>198</td>
<td>196</td>
</tr>
<tr>
<td>55%</td>
<td>132</td>
<td>130</td>
<td>130</td>
<td>140</td>
<td>176</td>
<td>178</td>
</tr>
<tr>
<td>40%</td>
<td>109</td>
<td>108</td>
<td>108</td>
<td>114</td>
<td>160</td>
<td>156</td>
</tr>
<tr>
<td>25%</td>
<td>86</td>
<td>84</td>
<td>84</td>
<td>90</td>
<td>132</td>
<td>130</td>
</tr>
<tr>
<td>15%</td>
<td>74</td>
<td>67</td>
<td>67</td>
<td>70</td>
<td>122</td>
<td>119</td>
</tr>
</tbody>
</table>

Table 2: Effect of skipping only ancestors in join performance.

- All the algorithms except, R-tree based ones, perform similarly
Performance Analysis

Figure 10: Effect of skipping only descendants.

Figure 11: Effect of skipping both ancestors and descendants.
Summary

- Indexing schemes can be enhanced to support their structural join algorithm
- Indexing schemes can be made durable, thus support updates on XML documents
- Their indexed algorithms are more robust than the state-of-the-art algorithms