Holistic Twig Joins on Indexed XML Documents

Haifeng Jiang, Wei Wang, Hongjun Lu, Jerey Xu Yu

Presented by Wanxing (Sarah) Xu

2008.10
XML Query process Method

- Structure Join: PAIR at one time
- Path Stack: PATH at one time
- Twig Stack: TWIG at one time

- Indexed-Twig: Using index for nodes to accelerate
How to Accelerate?

- Algorithm:
  - TSGeneric⁺: giving more opportunity to jump
- Index:
  - TR-tree: jumping faster and further each time
Algorithms

- The Generic Twig Join Algorithm (TSGeneric)
- The TSGeneric+ Algorithm
The **TSGeneric** Algorithm

- Algorithm \( \text{getNext}(q) \)
  - Returns a query node \( q_x \) in the subtree \( q \) satisfying all the following:
  - \( q_x \) has a solution extension.
  - if \( q_x \) has siblings, \( C_{q_x} \rightarrow \text{start} < C_{q_s} \rightarrow \text{start} \) for all sibling \( q_s \) of \( q_x \).
  - If \( q_x \neq q \), \( C_{\text{parent}(q_x)} \rightarrow \text{start} > C_{q_x} \rightarrow \text{start} \).
**getNext**($q$)

```plaintext
Algorithm 2 getNext($q$)

1: if isLeaf($q$) then
2:     return $q$;
3: end if
4: 
5: for $q_i$ in children($q$) do
6:     $n_i$ = getNext($q_i$);
7: end for
8: $n_{min}$ = minarg$_{n_i}$ {$C_{n_i} \rightarrow$ start};
9: $n_{max}$ = maxarg$_{n_i}$ {$C_{n_i} \rightarrow$ start};
10: while $C_q$ \rightarrow end < $C_{n_{max}}$ \rightarrow start do
11:     $C_q$ \rightarrow advance();
12: end while
13: if $C_q$ \rightarrow start < $C_{n_{min}}$ \rightarrow start then
14:     return $q$;
15: else
16:     return $n_{min}$;
```

If $q$ is a leaf, return $q$.
If $q$ is not a leaf, determine if the children of $q$ has solution extension rooted at them, else return descendant of $q$ with solution extension.

Determine descendant of $q$ with minimum and maximum start attribute.
Advance cursor of $q$ so so that $C_q$ \rightarrow end >= $C_{n_{max}}$ \rightarrow start.
If $C_q$ \rightarrow start < $C_{n_{min}}$ \rightarrow start return node $q$, else return descendant $n_{min}$ of $q$ with minimum start attribute.

**Corollary 1**
Cursor Interface

- $C_q \rightarrow \text{fwdBeyond}(C_p)$ forwards $C_q$ to the first element $e$, such that $e.\text{start} > C_p.\rightarrow\text{start}$
- $C_q \rightarrow \text{fwdToAncestorOf}(C_p)$ forwards the cursor to the first ancestor of $C_p$ and returns TRUE. If no such ancestor exists, it stops at the first element $e$, such that $e.\text{start} > C_p.\rightarrow\text{start}$, and returns FALSE.
### Algorithm 2 get\(\text{Next}(q)\)

1: if isLeaf\((q)\) then  
2: \hspace{1em} return \(q\);  
3: for \(q_i\) in children\((q)\) do  
4: \hspace{1em} \(n_i = \text{getNext}(q_i)\);  
5: \hspace{1em} if \(n_i \neq q_i\) then  
6: \hspace{2em} return \(n_i\);  
7: end for  
8: \(n_{\text{min}} = \min \arg_{n_i} \{ C_{n_i} \rightarrow \text{start} \};\)  
9: \(n_{\text{max}} = \max \arg_{n_i} \{ C_{n_i} \rightarrow \text{start} \};\)  
10: while \(C_q \rightarrow \text{end} < C_{n_{\text{max}}} \rightarrow \text{start}\) do  
11: \hspace{1em} \(C_q \rightarrow \text{advance}();\)  
12: end while  
13: if \(C_q \rightarrow \text{start} < C_{n_{\text{min}}} \rightarrow \text{start}\) then  
14: \hspace{1em} return \(q\);  
15: else  
16: \hspace{1em} return \(n_{\text{min}}\);  

### Algorithm 3 get\(\text{NextCursor}(q)\)

1: if isLeaf\((q)\) then  
2: \hspace{1em} return \(q\);  
3: for \(q_i\) in children\((q)\) do  
4: \hspace{1em} \(n_i = \text{getNextCursor}(q_i)\);  
5: \hspace{1em} if \(n_i \neq q_i\) then  
6: \hspace{2em} return \(n_i\);  
7: end for  
8: \(n_{\text{min}} = \min \arg_{n_i} \{ C_{n_i} \rightarrow \text{start} \};\)  
9: \(n_{\text{max}} = \max \arg_{n_i} \{ C_{n_i} \rightarrow \text{start} \};\)  
10: if \(C_q \rightarrow \text{fwdToAncestorOf}(C_{n_{\text{max}}}) == \text{TRUE}\) then  
11: \hspace{1em} if \(C_q\) is an ancestor of \(C_{n_{\text{min}}}\) then  
12: \hspace{2em} return \(q\);  
13: \hspace{1em} return \(n_{\text{min}}\);
XML Data Sample
getNext(q) (Examples)

getNext(root) = ?

Data Streams and Cursors:

- $C_a$: $a_1, a_2, a_3, a_4, a_5, a_6, a_7$
- $C_b$: $b_1, b_2, b_3, b_4, b_5$
- $C_c$: $c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}$
When can We Jump?

Lemma 1

Suppose a call of `getNextCursor(root)` returns a query node $q$. If the stack $S_{qa}$ of any ancestor $q_a$ of node $q$ is empty, then the current extension of node $q$ does not contribute to any further results and element $C_q$ can be discarded.
The *TSGenrice*\(^+\) Algorithm

- A cursor-based structural join algorithm (*SJCursor*)
- *Broken Edge* \((p, c)\): if elements \(C_p\) and \(C_c\) do not have an ancestor-descendant relationship
- *SJCursor*: finds the first ancestor-descendant pair starting from the current cursors of the two nodes connected by the edge.
getNextExt(q)

**Algorithm 3** getNextCursor(q)

1: if isLeaf(q) then
2: return q;
3: for qi in children(q) do
4: n_i = getNextCursor(q_i);
5: if n_i ≠ q_i then
6: return n_i;
7: end for
8: n_min = min arg_{n_i} \{C_{n_i} = start\};
9: n_max = max arg_{n_i} \{C_{n_i} = start\};
10: if C_q \rightarrow \text{fwdToAncestorOf}(C_{n_{max}}) = \text{TRUE} then
11: if C_q is an ancestor of C_{n_{min}} then
12: return q;
13: return n_{min};

**Algorithm 5** getNextExt(q)

1: if isLeaf(q) then
2: return q;
3: if empty(S_q) then
4: LocateExtension(q);
5: return q;
6: for qi in children(q) do
7: n_i = getNextExt(q_i);
8: if n_i ≠ q_i then
9: return n_i;
10: end for
11: n_min = min arg_{n_i} \{C_{n_i} = \text{start}\};
12: n_max = max arg_{n_i} \{C_{n_i} = \text{start}\};
13: if C_q \rightarrow \text{fwdToAncestorOf}(C_{n_{max}}) = \text{TRUE} then
14: if C_q is an ancestor of C_{n_{min}} then
15: return q;
16: return n_{min};
**Algorithm 6 LocateExtension (q)**

1: while (not end(q)) and (not hasExtension(q)) do
2:    \((p, c) = \text{PickBrokenEdge}(q)\); \{see section 4.1\}
3:    SJCursor \((p, c)\);
4: end while

Function hasExtension(q)
1: for each edge \((p, c)\) in the sub query tree q do
2:    if isBroken(p, c) then
3:        return FALSE;
4:    end for
5: return TRUE;

**Algorithm 7 PickBrokenEdge (q)**

1: Let Edges[1...K] be the vector containing all \(K\) broken edges in \(q\) in breadth first order;
2: if heuristic == MD then
3:    \((p_s, c_s) = \text{maxarg}_{(p_i, c_i)} \text{AvgDist}_{p_i} c_i\)
4: else if heuristic == TD then
5:    \((p_s, c_s) = \text{Edges}[1]);
6: else
7:    \((p_s, c_s) = \text{Edges}[K]);
8: return \((p_s, c_s)\);
Heuristics for picking an Edge

- **Maximum Distance (MD):** choose the edge whose next match is the *farthest* from the current cursors of its two nodes, so that we can skip the most number of edges.
- **Top Down (TD):** choose the first edge according to the breadth first traversal order
- **Bottom Up (BU):** choose the last edge according to the breadth first traversal order
An example of MD, TD, BU

maximum average distance (MD)

locate extension (a)

top-down (TD)

bottom-up (BU)

(1) X: broken edge
(2) numbers are average distances estimated
**SJCursor**\((p, c)\) Algorithm

<table>
<thead>
<tr>
<th>Algorithm 4 SJCursor ((p, c))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: while ((\text{not end}(C_p)) \text{ and } (\text{not end}(C_c))) and (\text{isBroken}(p, c)) do</td>
</tr>
</tbody>
</table>
| 2: \[i\text{f } C_p\rightarrow\text{start} < C_c\rightarrow\text{start} \\text{ then}
| 3: \quad C_p\rightarrow\text{fwdToAncestorOf}(C_c); |
| 4: \] else |
| 5: \quad C_c\rightarrow\text{fwdBeyond}(C_p); |
| 6: \] end while |
| Function \text{isBroken}(p, c) |
| 1: return \(\text{not } (C_p\rightarrow\text{start} < C_c\rightarrow\text{start} \text{ and } C_c\rightarrow\text{start} < C_p\rightarrow\text{end})\); |

If the edge is not broken, or either \(C_p\) or \(C_c\) reaches the end, return. Otherwise, proceed below.

If \(C_p\rightarrow\text{start} < C_c\rightarrow\text{start}\), move \(C_p\) to the first ancestor element of \(C_c\) (or beyond \(C_c\) if no such ancestor exists).

Otherwise, forward \(C_c\) to the first element whose start value is larger than \(C_p\rightarrow\text{start}\).
Calling $SJCursor(a, b)$
Calling \texttt{SJCursor}(a, b) (2)
Calling SJCursor\((a, b)\) (3)
Calling \textit{SJCursor}(a, b) (4)
Calling \textit{SJCursor}(b, c)
How to Accelerate?

- Algorithm:
  - TSGeneric\(^+\): giving more opportunity to jump
- Index:
  - TR-tree: jumping faster and further each time
XR-tree

- **XML Region Tree** (Jiang et al., ICDE 2002)
- Based on B⁺-Tree (based on the start position of each element $E_i(s_i, e_i)$, i.e. $s_i$).
- Extended internal nodes with **stab lists** and bookkeeping information.
- Nice property: given an element, all its ancestors and descendents can be identified very efficiently.
Stab

- Element with region $E_i(s_i, e_i)$; Key $k$
- $E_i$ is said to be **stabbed** by $k$, or $k$ **stabs** $E_i$ $\iff s_i \leq k \leq e_i$
- A set of ordered keys $k_j (0 \leq j < n)$ where $k_x < k_y$ if $x < y$.
- $E_i$ is said to be **primarily stabbed** by $k_j$, or $k_j$ **primarily tabs** $E_i$: $k_j$ is the smallest key that stabs $E_i$ among a set of ordered keys.
- The **(primary) stab list** of a key $k_j$ is the list of elements that are (primarily) stabbed by $k_j$, denoted as $(P)SL_i$ or $(P)SL_{(k_j)}$. 
Example of Stab, Stab Lists

- $SL_0 = \{E_0, E_1, E_2\}$; $PSL_0 = \{E_0, E_1, E_2\}$
- $SL_2 = \{E_0, E_4, E_5\}$; $PSL_2 = \{E_4, E_5\}$
- $SL_3 = \{E_0, E_4\}$; $PSL_3 = \emptyset$
**Internal Nodes**

- The start and end position $ps_j, pe_j$, of a key $k_j$, are defined as the start and end position of the first element in the primary stab list of $k_j$, if not empty; or $(nil, nil)$ if empty.

- $(k_0, s_0, e_0)$, $(k_1, s_3, e_3)$, $(k_2, s_4, e_4)$,
- $(k_3, nil, nil)$, $(k_4, s_6, e_6)$. 
• An element $e$ is included in the **stab list of an index page** $I$ if:
  
  (1) there exists some key $k$ in $I$ such that $e.start <= k <= e.end$ (or $k$ stabs the region of element $e$); and
  
  (2) no ancestor page of $I$ has a key that stabs $e$, i.e. $I$ is the highest index page that stabs $e$. 

**Figure 4:** The XR-tree for $c$ elements in Figure 1
Search for all descendants

- \( B^+ \)-tree is based on the start position of each element.
- Equivalent to \( B^+ \)-tree range search for \( e.start < R.start < e.end \) (elements do not have overlaps).

Figure 4: The XR-tree for \( c \) elements in Figure 1
Search for all ancestors

- All the ancestors of $e$ can be collected from the stab lists of index pages and the leaf page when we navigate down the XR-tree using $e\.start$

Figure 4: The XR-tree for $c$ elements in Figure 1
Cursor interfaces

- $C_q \rightarrow \text{advance}()$
- $C_q \rightarrow \text{fwdBeyond}(C_p)$
- $C_q \rightarrow \text{fwdToAncestorOf}(C_p)$

Figure 4: The XR-tree for $c$ elements in Figure 1
Performance of XR-tree

- Space: linear in the size of the XML document
- Time
  - $h$: $B^+$-tree heights; $R$: result size; $B$: block size
  - Search for all descendants: $O(h+R/B)$ in the worst case
  - Search for all ancestors: $O(h+R)$ in the worst case
  - Insert/delete: $O(h+c)$, amortized
Performance Study

- TwigStack, using TSGeneric
  - TwigStack (with no Index)
  - TwigStackXB (TwigStack with XB-tree index)
- XRTwig, using TSGeneric\(^+\) and XR-tree index
  - XRTwig(TD)
  - XRTwig(BU)
  - XRTwig(MD)

<table>
<thead>
<tr>
<th></th>
<th>TSGeneric</th>
<th>TSGeneric(^+)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Index</td>
<td>TwigStack</td>
<td></td>
</tr>
<tr>
<td>XB-tree</td>
<td>TwigStackXB</td>
<td></td>
</tr>
<tr>
<td>XR-tree</td>
<td></td>
<td>XRTwig (TD)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XRTwig (BU)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XRTwig (MD)</td>
</tr>
</tbody>
</table>
Figure 6: Experimental results for query Q1
Figure 7: Experimental results for query Q2

Figure 8: Experimental results for query Q3
Comparison of heuristics

Figure 9: #Page accesses under different edge-picking heuristics (thousand)
Thank you!

Questions?