Structural Joins, Twig Joins and Path Stack

Seminar:
XML & Datenbanken

Student:
Irina ANDREI

Konstanz, 11.07.2006
Outline

1. Structural Joins
   • Tree-Merge
   • Stack-Tree
2. Path-Join Algorithms
   • PathStack
   • PathMPMJ
3. Twig Joins:
   • TwigStack
   • TJFast
4. Comparisons
5. Conclusion
6. References
Outline

1. Structural Joins
   - Tree-Merge
   - Stack-Tree
2. Path-Join Algorithms
   - PathStack
   - PathMPMJ
3. Twig Joins:
   - TwigStack
   - TJFast
4. Comparisons
5. Conclusion
6. References
Data Model and Query Patterns

- XQuery path expressions can be represented as a node-labeled tree pattern with elements and string values as node labels.
- Node labels = set of (attribute, value) pairs.
- Example:
  \[ \text{book[title = \textquote{XML}]/author[. = \textquote{jane}]} \]
Matching Basic Structural Relationships

1. Decompose the complex query tree pattern in basic binary structural relationships
2. Match each of the binary relationships against the XML database
3. “Stitch” together the basic matches

- Matching ancestor-descendant structural relationships:
  a) (parent, child) repeated joins (ancestor, descendant) => too much query processing time
  b) (ancestor, descendant) => too much space
Representing Positions of Elements and String Values in an XML Database

- Position of element occurrence:
  (DocId, StartPos : EndPos, LevelNum)

- Position of string occurrence:
  (DocId, StartPos, LevelNum)

- Checking an ancestor-descendant structural relationship is as easy as checking a parent-child structural relationship
  - \( n_2 (D_2, S_2:E_2, L_2) \) descendant of \( n_1 (D_1, S_1:E_1, L_1) \) if \( D_1=D_2, \ S_1<S_2, \ E_1>E_2 \)
  - \( n_2 (D_2, S_2:E_2, L_2) \) child of \( n_1 (D_1, S_1:E_1, L_1) \) if \( D_1=D_2, \ S_1<S_2, \ E_1>E_2 \) and \( L_1+1=L_2 \)
Sample XML Document Fragment

```xml
<book>
  <title>XML</title>
  <allauthors>
    <author>jane</author>
    <author>john</author>
  </allauthors>
  <year>2000</year>
  <chapter>
    <head>Origins</head>
    <section>
      <head>...</head>
      <section>...</section>
    </section>
    <section>...</section>
  </chapter>
  <chapter>...</chapter>
</book>
```
Outline

1. Structural Joins
   • Tree-Merge
   • Stack-Tree
2. Path-Join Algorithms
   • PathStack
   • PathMPMJ
3. Twig Joins:
   • TwigStack
   • TJFast
4. Comparisons
5. Conclusion
6. References
TreeMerge Join Algorithms

- Natural extension of traditional relational merge joins
- Basic idea is to perform a modified merge-join, possibly performing multiple scans through the “inner” join operand to the extent necessary
- Operands: ancestor / descendant sets
  - AList is the list of potential ancestors, in sorted order of StartPos
  - DList is the list of potential descendants in sorted order of StartPos
- the results are sorted (primarily) by the outer operand =>
  - Tree-Merge-Anc
  - Tree-Merge-Desc
Algorithm Tree-Merge-Anc

Algorithm Tree-Merge-Anc (AList, DList)
/* Assume that all nodes in AList and DList have the same DocId */
/* AList is the list of potential ancestors, in sorted order of StartPos */
/* DList is the list of potential descendants in sorted order of StartPos */

begin-desc = DList->firstNode; OutputList = NULL;
for (a = AList->firstNode; a != NULL; a = a->nextNode) {
    for d = begin-desc; (d != NULL && d.StartPos < a.StartPos); d = d->nextNode) {
        /* skipping over unmatchable d’s */
        begin-desc = d;
        for (d = begin-desc; (d != NULL && d.EndPos < a.EndPos); d = d->nextNode) {
            if ((a.StartPos < d.StartPos) && (d.EndPos < a.EndPos) &&
                (d.LevelNum = a.LevelNum + 1))
                /* the optional condition is for parent-child relationships */
                append (a,d) to OutputList;
        }
    }
}
Analysis of the Tree-Merge Algorithms

- The space and time complexities of Algorithm Tree-Merge-Anc are $O(|\text{AList}|+|\text{DList}|+|\text{OutputList}|)$ for the ancestor-descendant structural relationship.

- For parent-child: $O((|\text{AList}|+|\text{DList}|)^2)$ (a,b)

- The time complexity of the algorithm Tree-Merge-Desc can be $O((|\text{Alist}|+|\text{DList}|+|\text{OutputList}|)^2)$ in the worst case (c,d).
Outline

1. Structural Joins
   ✓ Tree-Merge
   • Stack-Tree
2. Path-Join Algorithms
   • PathStack
   • PathMPMJ
3. Twig Joins:
   • TwigStack
   • TJFast
4. Comparisons
5. Conclusion
6. References
Stack-Tree Join Algorithms

- Basic idea is: take the two input operand lists, AList and DList, both sorted on their (DocId, StartPos) values and conceptually merge (interleave) them.

- As the merge proceeds, we determine the ancestor-descendant relationship, if any, between the current top of stack and the next node in the merge, i.e., the node with the smallest value of StartPos.

- Based on this comparison, we manipulate the stack, and produce output.
  - Stack-Tree-Desc
  - Stack-Tree-Asc
Algorithm Stack-Tree-Desc

Algorithm Stack-Tree-Desc (AList, DList)
/* Assume that all nodes in AList and DList have the same DocId */
/* AList is the list of potential ancestors, in sorted order of StartPos */
/* DList is the list of potential descendants in sorted order of StartPos */

a = AList->firstNode; d = DList->firstNode; OutputList = NULL;
while (the input lists are not empty or the stack is not empty) {
    if ((a.StartPos > stack->top.EndPos) && (d.StartPos > stack->top.EndPos)) {
        /* time to pop the top element in the stack */
        tuple = stack->pop();
    } else if (a.StartPos < d.StartPos) {
        stack->push(a)
        a = a->nextNode
    } else {
        for (a₁ = stack->bottom; a₁ != NULL; a₁ = a₁->up) {
            append (a₁, d) to OutputList
        }
        d = d->nextNode
    }
}
Execution steps
Analysis of the Stack-Tree Algorithms

- The space and time complexities of Algorithms Stack-Tree-Desc and –Anc are
  \[ O(|AList| + |DList| + |OutputList|) \]
  for both ancestor-descendant and parent-child structural relationships.

- Algorithm Stack-Tree-Desc is a non-blocking algorithm.

- The I/O complexity of Algorithm Stack-Tree-Desc and –Anc is
  \[ O(|AList|/B + |DList|/B + |OutputList|/B) \]
  where B is the blocking factor
Outline

1. Structural Joins
   ✓ Tree-Merge
   ✓ Stack-Tree
2. Path-Join Algorithms
   • PathStack
   • PathMPMJ
3. Twig Joins:
   • TwigStack
   • TJFast
4. Comparisons
5. Conclusion
6. References
Data Model

- The problem of stitching together the basic matches obtained using binary structural joins requires identifying a good join ordering in a cost-based manner, taking selectivities and intermediate result size estimates into account.

- Query tree pattern = twig pattern

- In general, at each node in the query twig pattern, there is a node predicate on the attributes (e.g., tag, content) of the node in question.

```
book[title='XML']/author[fn='jane' AND ln='doe']
```
Twig Pattern Matching

- Finding all occurrences of twig pattern in a database – core operation in XML query processing

- Given a query twig pattern Q and an XML database D, a match of Q in D is identified by mapping from nodes in Q to nodes in D, such that:
  - Query node predicates are satisfied by the corresponding database nodes
  - The structural (p-c, a-d) relationships between query nodes are satisfied by the database nodes.

- Positions: (DocId, LeftPos : RightPos, LevelNum)
Notation

- $q = \text{twig patterns / the root node of the twig pattern}$
- Node operations:
  - `isLeaf: Node \rightarrow Bool`
  - `isRoot: Node \rightarrow Bool`
  - `parent: Node \rightarrow Node`
  - `children: Node \rightarrow \{Node\}`
  - `subtreeNodes: Node \rightarrow \{Node\}`

- Stream $T_q$ associated with each node $q$ in the query twig pattern: positional representations sorted by $(\text{DocId, LeftPos})$

- Operations over streams:
  - `eof, advance, next, nextL, nextR`
Notation cont.

- In the stack-based algorithms PathStack and TwigStack there is a stack $S_q$ for each query node $q$.
- Each data node in the stack consists of a pair:
  (positional representation of a node from $T_q$, pointer to a node in $S_{\text{parent}(q)}$)
- `empty, pop, push, topL, topR`
Outline

1. Structural Joins
   ✓ Tree-Merge
   ✓ Stack-Tree
2. Path-Join Algorithms
   • PathStack
   • PathMPMJ
3. Twig Joins:
   • TwigStack
   • TJFast
4. Comparisons
5. Conclusion
6. References
PathStack

- Generalization of *Stack-Tree-Desc*
- Computes answers to a query path pattern
- Key idea: repeatedly construct (compact) stack encodings of partial and total answers to the query path pattern, by iterating through the stream nodes in sorted order of their `LeftPos` values from the query root to the query leaf
Algorithm PathStack

Algorithm PathStack(q)
01 while ¬end(q)
02 \( q_{min} = \text{getMinSource}(q) \) // identifies the stream containing next node
03 for \( q_i \) in subtreeNodes(q) // clean stacks
04 while (¬empty(S\(_{qi}\)) \&\& topR(S\(_{qi}\))<nextL(T\(_{q_{min}}\))
05 pop(S\(_{qi}\)) // remove partial answers from the stacks that cannot be extended to total answers
06 moveStreamToStack(T\(_{q_{min}}\), S\(_{q_{min}}\), pointer to top(S\(_{parent(q_{min})}\)))
  /* augments the partial answers encoded in the stacks with the new stream node */
07 if (isLeaf(q\(_{min}\)))
08 showSolutions(S\(_{q_{min}}\), 1)
09 pop(S\(_{q_{min}}\))

Function end(q)
    return any \( q_i \in \text{subtreeNodes}(q) : \) isLeaf(q\(_i\)) \Rightarrow \text{eof}(T\(_{q_i}\))
Algorithm PathStack cont.

Function getMinSource(q)
    return \( q_i \in \text{subtreeNodes}(q) \) such that \( \text{nextL}(T_{q_i}) \) is minimal

Procedure moveStreamToStack(T_q, S_q, p)
    push(S_q, (next(T_q), p))
    advance(T_q)

Procedure showSolutions(SN, SP)
01  index[SN]=SP
02  if (SN==1) //we are in the root
03      //output solutions from the stacks
04      Output (S[n].index[n],...,S[1].index[1])
05  else //recursive call
06      for i=1 to S[SN].index[SN].pointer_to_parent
07          showSolutions(SN-1,i)
Analysis of PathStack

- For p-c relationships is needed the LevelNum
- worst-case I/O and CPU time complexities linear in the sum of sizes of the n input and the output lists
- Worst-case space complexity is the minimum of
  - The sum of sizes of the n input lists
  - The maximum length of a root-to-leaf path in D
- Worst-case time complexity is independent of the sizes of any intermediate results
Outline

1. Structural Joins
   ✓ Tree-Merge
   ✓ Stack-Tree
2. Path-Join Algorithms
   ✓ PathStack
   • PathMPMJ
3. Twig Joins:
   • TwigStack
   • TJFast
4. Comparisons
5. Conclusion
6. References
PathMPMJ

- Generalization of the MPMGJN binary algorithm
- Consider the path query q1//q2//q3
- Basic idea: get the first (next) element from the stream $T_{q_1}$ and generate all solutions that use that particular element from $T_{q_1}$
- Then, advance $T_{q_1}$ and backtrack $T_{q_2}$ and $T_{q_3}$ accordingly
- Each query node has a stack of $k$ marks, where $k$ is the number of its ancestors in the query
Outline

1. Structural Joins
   ✓ Tree-Merge
   ✓ Stack-Tree
2. Path-Join Algorithms
   ✓ PathStack
   ✓ PathMPMJ
3. Twig Joins
   • TwigStack
   • TJFast
4. Comparisons
5. Conclusion
6. References
Limitations of using PathStack

- Many intermediate results may not be part of any final answer
- In this case, using PathStack as a subroutine is suboptimal
Outline

1. Structural Joins
   ✓ Tree-Merge
   ✓ Stack-Tree
2. Path-Join Algorithms
   ✓ PathStack
   ✓ PathMPMJ
3. Twig Joins
   • TwigStack
   • TJFast
4. Comparisons
5. Conclusion
6. References
Twig Stack

- Refines PathStack and computes answers to a query twig pattern
- Uses a chain of linked stacks to compactly represent partial results to individuals query root-to-leaf paths
- No large intermediate results are created
- 2 phases:
  - Some solutions to individual query root-to-leaf paths are computed
  - These solutions are merge-joined to compute the final answers
- Key difference between PathStack and I TwigStack: before a node $h_q$ from the stream $T_q$ is pushed on its stack $S_q$, TwigStack ensures that:
  - Node $h_q$ has a descendant $h_{qi}$ in each of the streams $T_{qi}$
  - Each of the nodes $h_{qi}$ recursively satisfies the first property.
Analysis of TwigStack

- worst-case I/O and CPU time complexities: linear in the sum of sizes of the n input lists and the output list
- Worst-case space complexity minimum of:
  - The sum of sizes of the n input lists
  - N times the maximum length of a root-to-leaf path in D
- Independent of the sizes of intermediate results
Outline

1. Structural Joins
   ✓ Tree-Merge
   ✓ Stack-Tree
2. Path-Join Algorithms
   ✓ PathStack
   ✓ PathMPMJ
3. Twig Joins
   ✓ TwigStack
   • TJFast
4. Comparisons
5. Conclusion
6. References
DataModel

- Previous holistic twig matching algorithms are inefficient for queries with wildcards in branching nodes

  //a/*[b]/c

- The twig pattern node may be:
  - Element tag
  - Text value
  - *Wildcard “*”*
Dewey Labeling Scheme

- From the label of an element alone, we can derive the names of all elements in the path from the root to this element.
- Each element is presented by a vector:
  - The root is labeled by an empty string $\epsilon$.
  - For a non-root element $u$, $\text{label}(u) = \text{label}(s).x$, where $u$ is the $x$-th child of $s$.
- Element $u$ is an ancestor of element $s$ if and only if $\text{label}(u)$ is a prefix of $\text{label}(s)$.

Example: $\text{label}(u) = \text{“1.2.3.4”} \Rightarrow$

- XML path pattern matching can be directly reduced to string matching.
Extended Dewey

- Uses the *modulo* function to create a mapping from an integer to an element name, such that given a sequence of integers, we can convert it into the sequence of element names.

- *child names clue* = all (distinct) names of children (DTD)

- $\text{CT}(t) = \{t_0, t_1, \ldots, t_{n-1}\}$

- For any element $e_i$ with name $t_i$ assign an integer $x_i$ such that $x_i \mod n = i$.

- Example: $\text{CT}(\text{book}) = \{\text{author, title, chapter}\}$. $e_i$ is child of book and $x_i = 8 \iff t_i = \text{chapter}$, because $x_i \mod 3 = 2$. 
Extended Dewey cont.

- for any element $u$ with parent $s$ in an XML tree,
  - if $u$ is a text value, then $x = i_1$;
  - otherwise, assume that the element name of $u$ is the $k$-th tag in $CT(ts)$ ($k=0,1,...,n-1$), where $ts$ denotes the tag of element $s$.
    - (2.1) if $u$ is the first child of $s$, then $x = k$;
    - (2.2) otherwise assume that the last component of the label of the left sibling of $u$ is $y$, then
      $$x = \begin{cases} 
      \left\lfloor \frac{y}{n} \right\rfloor \cdot n + k & \text{if } (y \mod n) < k; \\
      \left\lfloor \frac{y}{n} \right\rfloor \cdot n + k & \text{otherwise.}
    \end{cases}$$

- The extended Dewey does not alter the asymptotic space complexity of the original one.
- *finite state transducer* (FST) - convert labels into the sequence of element names which reveals the *whole path from the root to this element*.
An XML tree with extended Dewey labels
Properties of Extended Dewey

1. [Ancestor Name Vision] Given any extended Dewey label of an element, we can know all its ancestors' names (through FST).

2. [Ancestor Label Vision] Given any extended Dewey label of an element, we can know all its ancestors' label.

3. [Prefix relationship] Two elements have ancestor-descendant relationships if and only if their extended Dewey labels have a prefix relationship.

4. [Tight Prefix relationship] Two elements $a$ and $b$ have parent-child relationships if and only if their extended Dewey labels $\text{label}(a)$, $\text{label}(b)$ have a tight prefix relationship. That is: (i) $\text{label}(a)$ is a prefix of $\text{label}(b)$; and (ii) $\text{label}(b).\text{length} - \text{label}(a).\text{length} = 1$.

5. [Order relationship] Element $a$ follows (or precedes) element $b$ if and only if $\text{label}(a)$ is greater (or smaller) than $\text{label}(b)$ with lexicographical order.
TJFast

- Based on extended Dewey labeling scheme
- To answer a twig query, TJFast only needs to access the labels of the leaf query nodes =>
  - Reduce disk access
  - Support the efficient evaluation of queries with wildcards in branching nodes
Notation

- $q$ - twig pattern
- $p_n$ - path pattern from the root to the node $n \in q$
- node operations:
  - isLeaf: Node → Bool;
  - isBranching: Node → Bool;
  - leafNodes: Node → {Node};
  - directBranchingOrLeafNodes: Node → {Node}
- Associated with each leaf node $f$ in a query twig pattern there is a stream $T_f$ contains extended Dewey labels of elements that match the node type $f$.
- The elements in the stream are sorted by the ascending lexicographical order
- operations over a stream $T_f$: current($T_f$), advance($T_f$) and eof($T_f$)
Notation cont.

- Algorithm TJFast keeps a data structure during execution: a set $S_b$ for each branching node $b$ (initially empty)
- Each two elements in set $S_b$ have an ancestor-descendant or parent-child relationship
- Maximal size of $S_b = \text{the length of the longest path}$ in the document
- Each element cached in sets likely participates in query answers
Algorithm TJFast

- 2 phases
  - some solutions to individual root-leaf path patterns are computed
  - These solutions are merge-joined to compute the answers to the query twig pattern
- if two path solutions can be merged, the necessary condition is that they have the common element to match the *branching* query node
- Example: a[./b]/c has path solutions: \((a_1; b_1)\) and \((a_2; c_1)\). If \(a_1 = a_2\) => the two solutions can be merged
Algorithm TJFAst

Algorithm 1 TJFast
1: for each \( f \) 2 leafNodes(root)
2: \( \text{locateMatchedLabel}(f) \) // locate the first element whose path matches
   //the individual root-leaf path pattern
3: endfor
4: while (¬end(root)) do
5:   \( f_{act} = \text{getNext}(\text{topBranchingNode}) \) //next Stream to be processed
6:   outputSolutions(\( f_{act} \)) //some path matching solutions
7:   advance(\( T_{fact} \))
8:   \( \text{locateMatchedLabel}(f_{act}) \) //next matching element
9: end while
10: mergeAllPathSolutions()

Procedure locateMatchedLabel(\( f \))
/* Assume that the path from the root to element get(\( T_f \)) is \( n_1/n_2/.../n_k \)
   and \( pf \) denotes the path pattern from the root to leaf node \( f \) */
1: while ¬((\( n_1/n_2/.../n_k \) matches pattern \( pf \)) \( ^\wedge \) (\( n_k \) matches \( f \))) do
2:   advance(\( T_x \))
3: end while
Algorithm TJFast

Algorithm getNext:
- Identifies next stream to process
- Update the sets $S_b$ associated with branching nodes $b$
- **worst case** I/O complexity of TJFast is linear in the sum of the sizes of input and output lists.
- **worst-case space complexity** is $O(d^2 \times |b| + d \times |f|)$,
  - $|f|$ is the number of leaf nodes in $q$
  - $|b|$ is the number of branching nodes in $q$
  - $d$ is the length of the longest label in the input lists
- if the query contains *parent-child* relationships between branching nodes and their children, Algorithm TJFast is no longer guaranteed to be I/O optimal
Outline

1. Structural Joins
   ✓ Tree-Merge
   ✓ Stack-Tree
2. Path-Join Algorithms
   ✓ PathStack
   ✓ PathMPMJ
3. Twig Joins
   ✓ TwigStack
   ✓ TJFast
4. Comparisons
5. Conclusion
6. References
Structural Joins: Tree-Merge vs. Stack-Tree

![Bar chart comparing response times for different methods.](image)
PathStack vs. TwigStack

(a) Execution time

(b) Number of partial solutions
PathStack versus TJFast using XMark data

(a) Number of elements read  (b) Size of disk files scanned  (c) Execution time
PathStack versus TJFast using random data

(a) Number of elements read  
(b) Size of disk files scanned  
(c) Execution time
Comparisons

- Substantial performance benefits of using holistic twig joins over binary structural joins (for arbitrary join orders)
- Among holistic path join algorithms, PathStack is significantly faster than PathMPMJ
- For the case of twig patterns, the use of TwigStack is better (both in time and space) than the independent use of PathStack on each root-to-leaf path, even when the twig pattern contains p-c relationships
Outline

1. Structural Joins
   ✓ Tree-Merge
   ✓ Stack-Tree
2. Path-Join Algorithms
   ✓ PathStack
   ✓ PathMPMJ
3. Twig Joins
   ✓ TwigStack
   ✓ TJFast
4. Comparisons
5. Conclusion
6. References
Conclusion

- Algorithms equally relevant for RDBMS and native XML query engines
- the Stack-Tree family of structural join algorithms, PathStack and TwigStack are both I/O and CPU optimal, and practically efficient
- TJFast only needs to access labels of leaf nodes to answer queries and significantly reduce I/O cost
- TJFast can efficiently evaluate queries with wildcards steps, which cannot be handled by algorithms with region encoding
Outline

1. Structural Joins
   ✓ Tree-Merge
   ✓ Stack-Tree
2. Path-Join Algorithms
   ✓ PathStack
   ✓ PathMPMJ
3. Twig Joins
   ✓ TwigStack
   ✓ TJFast
4. Comparisons
5. Conclusion
6. References
References

- Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, Yuqing Wu - Structural Joins: A Primitive for Efficient XML Query Pattern Matching

- Nicolas Bruno, Nick Koudas, Divesh Srivastava - Holistic Twig Joins: Optimal XML Pattern Matching

- Jiaheng Lu, Tok Wang Ling, Chee-Yong Chan, Ting Chen - From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching
Thank you for your attention!