Recursive XML Schemas, Recursive XML Queries, and Relational Storage: XML-to-SQL Query Translation

By: Rajasekar Krishnamurthy, etc
Presented by: Wenguo Liu
11/18/2003

Outline
- Introduction
- Formal Model
- Query Translation Over Recursive XML schemas
- Extensions to more complex path expressions
- Conclusions

Introduction – Problem
- Translate path expression queries to SQL
  - Schema-based XML Storage shredding of XML into relations
  - Recursion in the schema
  - Recursive XML queries
    - Path expression queries having the descendant axis (//)

Introduction – Related Work
- Shredding XML data into relations
  - Schema-based techniques
  - Schema-oblivious shredding
- Translating XML queries into SQL
- Claim of this paper
  - None of previous work solves the query translation problem for schema-based shredding in the presence of recursion in the XML schema

Formal Model – XML Schema Graph
- A directed graph (V, E)
  - V: vertices corresponding to elements and attributes
    - Labeled with name of the element or attribute
  - E: edges representing parent-child relationships
    - Have a label from {?, *, +, ε}

Formal Model – Relational Schema
- Mapping XML elements and relational columns
  - Method: annotate XML schema graph
    - Non-leaf (internal) node <= Relation name
    - Leaf node <= column name
    - Node n has multiple incoming edges
      - parentcode = val

Formal Model – Annotated XML Schema Graph
- DTD
  - <?xml encoding="US-ASCII">>
  - <!ELEMENT book(title, author*, section*)>
  - <!ELEMENT title (#PCDATA)>
  - <!ELEMENT author (#PCDATA)>
  - <!ELEMENT section (title, p*, section?, note*)>
  - <!ELEMENT note (#PCDATA)>
  - Recursive
  - XML Schema Graph
  - Annotated XML Schema Graph
Formal Model – XML to Relational Mappings

- Relation has an id field (primary key)
- Preserve Document structure
  - Parentid
  - parentcode

Formal Model – Relations

<table>
<thead>
<tr>
<th>Book</th>
<th>Author</th>
<th>Para</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>title</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>parentid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>parentid</td>
</tr>
<tr>
<td>parentid</td>
<td>parentcode</td>
</tr>
<tr>
<td>title</td>
<td></td>
</tr>
</tbody>
</table>

Path Expression Queries

- Simple path expression (SPE)
  - $s_1[i_1]s_2[i_2]...s_k[i_k]$
  - $s_i$ is / or //
  - $i_j$ is a tag name
  - $s_j$ represents a navigation step of the path expression

- Generalized simple path expression (GSPE)
  - $p_1p_2...p_k$
  - Each $p_i$ denotes a disjunction of simple path expressions

Query Translation – PathId

- Query $Q = p_1p_2...p_k$
- The path in schema graph satisfy $Q$
- Problems
  - Could be infinite for recursive schema!
  - Could be exponential for DAG schema graphs!
- Solution
  - Representing the matching paths as a graph!

Query Translation Over Recursive XML Schemas

- PathId stage
  - Identify the paths in the XML schema graph that satisfy the query
- SQLGen stage
  - Construct an equivalent relational query from the XML-to-Relational mapping

Query Translation - PathId

- Result from PathId
  - Cross product schema $S_{SQ}$
  - Matching paths encoded in $S_{SQ}$
  - Union of all root-to-leaf paths in $S_{SQ}$ corresponds to the query result
Query Translation – SQLGen

SQLGen steps
1. Identify strongly connected components (SCCs) in S
2. Merge adjacent SCCs based on dominance
3. For each SCC
   i. Generate the query for this SCC
   ii. A relational query \( T(n) \) is associated with each left node \( n \)
4. finalQ = \( \bigcup \) if \( n \) is a leaf node
   "select * from \( T(n) \)"
5. Output results
   i. If duplicate elimination needed, output "select distinct(*) from finalQ"
   ii. Else, "select * from finalQ"

SQLGen – Step 3: SQLForDAG \( (c) \)

- Associate a temporary relation \( T(n) \) with a node \( n \) which
  - is leaf node, or
  - has a parent or child in a different component, or
  - has multiple incoming/outgoing edges
- \( T(n) \) is the union of all SQL generated along possible paths

A recursive Schema

SQLGen – Step 3:

SQLForDAG \( (c) \)

SQLGen – Step 3:

SQLForRecursive \( (c) \)

- Associate a temporary relation \( T_R \) to \( c \)
  - Initialization part
    - Captures all incoming edges into \( c \) from a different component, (2,8) and (3,7)
  - Recursive part
    - The union across all edges within the component

SQLGen – Step 3:

SQLForRecursive

with TR as (  
  select R7.*, id(7)  
  from TR, R7  
  where R7.parentid=TR.id  
  and TR.schemanode=id(8)  
  union all  
  select R7.*  
  from TR, R7  
  where R7.parentid=TR.id  
  and TR.schemanode=id(8)  
)
Extensions to more complex path expressions

- Branching path expression queries
  - \( p_1[p_2] \) or \( p_1[p_2 \text{ op value}] \), \( p_1 \) and \( p_2 \) are GSPE queries

- PathId stage
  - Deal with \( p_1 \) first, let \( F \) be the final states
  - Compute an automaton for \( p_2 \) with a \( f \) in \( F \) as start state

- SQLGen stage
  - Deal with \( p_1 \) and \( p_2 \) consecutively

Handling Order

- Handling order in XPath semantics
  - Queries need to return results in document order
  - XML into relations with positions
    - Maintain the relative position among sibling XML elements
    - Not the focus of this paper

Conclusions

- An algorithm to translate path expression queries to SQL
  - Recursion in the XML schema
  - Recursion in the queries
  - This algorithm is not quite general
  - Apply to SPE and GSPE
  - Linear recursion in SQL99 is sufficient for this translation

Discussions

- XML-to-Relational mapping
  - There must be no data in the relations other than that which is present in the XML document? True?
  - SPE and GSPE are quite similar to regular expressions
  - Why not just use regular expression approach?
  - More complex applications
    - FLWR