Indexing and Querying XML Data for Regular Path Expressions

Quanzhong Li
Bongki Moon

Brought to you by George Thomas {george@cc.gatech.edu}
February 27, 2003
What’s the big deal?

- Numbering scheme for elements and attributes
- **XISS**
  - Indexing and storing system for XML
- Query processing paradigm
  - Decompose regular path expressions
  - Path-join algorithms to process these expressions
Storage for Retrieval

- Tree model for XML data
- Several XML query languages
  - Regular path expressions
- Approaches based on tree traversals are potentially expensive
- Optimal query plan
  - Value
    - Element names/values
    - Attribute names/values
  - Structure
    - Ancestor-descendant relationships
Numbering scheme

- *Extended preorder*
- Ancestor-descendant relationship between elements and attributes
- Support proposed join algorithms
Dietz’s Numbering Scheme

- First to use tree traversal order to determine the ancestor-descendant relationship between any pair of tree nodes
- Based on preorder and postorder

For two given nodes $x$ and $y$ of a tree $T$, $x$ is an ancestor of $y$ if and only if $x$ occurs before $y$ in the preorder traversal of $T$ and after $y$ in the post-order traversal.
The Good and Bad

The Good
- Determine the ancestor-descendant relationship in constant time

The Bad
- Inflexible
- Inserts/Deletes
The Extended Preorder Numbering Scheme

- Uses *extended preorder* and a *range of descendants*
- Each node has a pair of numbers
  - \(<\text{order}, \text{size}>\)
- For a tree node \(y\) and its parent \(x\), \(\text{order}(x) < \text{order}(y)\) and \(\text{order}(y) + \text{size}(y) \leq \text{order}(x) + \text{size}(x)\)
- The interval \([\text{order}(y), \text{order}(y) + \text{size}(y)]\) is contained in the interval \([\text{order}(x), \text{order}(x) + \text{size}(x)]\)
- For two sibling nodes \(x\) and \(y\), if \(x\) is the predecessor of \(y\) in preorder traversal, \(\text{order}(x) + \text{size}(x) < \text{order}(y)\)
- For a tree node \(x\), \(\text{size}(x) \geq \sum \text{size}(y)\) for all \(y\)'s that are a direct child of \(x\)

For two given nodes \(x\) and \(y\) of a tree \(T\), \(x\) is an ancestor of \(y\) if and only if \(\text{order}(x) < \text{order}(y) \leq \text{order}(x) + \text{size}(x)\).
The Good and the Bad

- **The Good**
  - More flexible
  - Tolerant of dynamic data updates
  - Order values of deleted nodes can be recycled

- **The Bad**
  - They don’t tell us how they plan to recycle these order values
XML Indexing and Storage System
Three major index structures
  - Element index
  - Attribute index
  - Structure index
Two additional structures
  - Name index
  - Value table
Data loader (not discussed)
Query processor (not discussed)
XISS Components

- Document id (did) for each XML document
- Name index
  - B+ tree
  - Name identifier (nid)
- Element, attribute, structure indices
  - B+ tree
  - Name identifiers (nid) as keys
Element and Attribute Indices

- **Element index**
  - Allows quick search for elements with the same name string

- **Attribute index**
  - Each record has a value identifier (vid)
    - Key into the value table
Structure Index

- Collection of linear arrays
  - Elements and attributes are sorted by the *order* value
  - Attributes are placed before their sibling elements
Supported operations

- For a given element name string, say figure, find a list of elements having the same name string (*i.e.* figure), grouped by documents which they belong to (element index)
- For a given attribute name string, say caption, find a list of attributes having the same name string (*i.e.*, caption), grouped by documents which they belong to (attribute index)
- For a given element, find its parent element and child elements (or attributes). For a given attribute, find its parent element (structure index)
Conventions and recommendations

- Conventional approaches
  - Top-down
    - Tree traversal cost
  - Bottom-up
    - Expensive if there are more ancestors and fewer descendants
  - Hybrid
    - Effectiveness not always guaranteed
Join Algorithms and Decomposing Path expressions

Different basic subexpressions
- a subexpression with a single element or a single attribute
- a subexpression with an element and an attribute (e.g., figure[@caption = "Tree Frogs"] \{EA Join\})
- a subexpression with two elements (e.g., chapter/figure or chapter/_*/figure) \{EE Join\}
- a subexpression that is a Kleene closure (+,*) of another subexpression \{KC Join\}
- a subexpression that is a union of two other subexpressions

Cons: No detail on decomposition strategies
Performance results

- EE Join algorithm outperforms the bottom-up method
  - Access patterns
  - Nature of data
- Bottom-up method outperformed the EA Join algorithm
  - Nature of data
- KC Join performance was not evaluated
- Query processing time increased almost linearly, as the size of the XML data increased
Open Issues

- A more formal analysis of the algorithms
- Explore optimal ways to decompose expressions
- Explore trade-off between disk access efficiency and storage utilization