Similarity Search
Trees and Relational Databases

Nikolaus Augsten

Free University of Bozen-Bolzano
Faculty of Computer Science
DIS

Unit 4 – March, 2012
Outline

1 What is a Tree?

2 Encoding XML in a Relational Database
   - Adjacency List Encoding
   - Dewey Encoding
   - Interval Encoding
   - Experimental Comparison of the Encodings
   - XML and Trees
Outline

1. What is a Tree?

2. Encoding XML in a Relational Database
   - Adjacency List Encoding
   - Dewey Encoding
   - Interval Encoding
   - Experimental Comparison of the Encodings
   - XML and Trees
Graph: a pair \((N, E)\) of nodes \(N\) and edges \(E\) between nodes of \(N\)

Tree: a directed, acyclic graph \(T\)
- that is connected and
- no node has more than one incoming edge

Edges: \(E(T)\) are the edges of \(T\)
- an edge \((p, c)\) ∈ \(E(T)\) is an ordered pair
- with \(p, c \in N(T)\)

“Special” Nodes: \(N(T)\) are the nodes of \(T\)
- parent/child: \((p, c) \in E(T)\) ⇔ \(p\) is the parent of \(c\), \(c\) is the child of \(p\)
- siblings: \(c_1\) and \(c_2\) are siblings if they have the same parent node
- root node: node without parent (no incoming edge)
- leaf node: node without children (no outgoing edge)
- fanout: fanout \(f_v\) of node \(v\) is the number of children of \(v\)
Unlabeled Trees

- **Unlabeled Tree:**
  - the focus is on the structure, not on distinguishing nodes
  - however, we need to distinguish nodes in order to define edges
  - \( \Rightarrow \) each node \( v \) has a unique identifier \( id(v) \) within the tree

- **Example:** \( T = (\{1, 3, 5, 4, 7\}, \{(1, 3), (1, 5), (5, 4), (5, 7)\}) \)
Edge Labeled Trees

**Edge Labeled Tree:**
- an edge \( e \in E(T) \) between nodes \( a \) and \( b \) is a triple \( e = (id(a), id(b), \lambda(e)) \)
- \( id(a) \) and \( id(b) \) are node IDs
- \( \lambda(e) \) is the edge label (not necessarily unique within the tree)

**Example:**
\[
T = \{(1, 3, a), (1, 5, b), (5, 4, c), (5, 7, a)\}
\]
**Node Labeled Trees**

- **Node Labeled Tree:**
  - A node $v \in N(T)$ is a pair $(id(v), \lambda(v))$
  - $id(v)$ is unique within the tree
  - Label $\lambda(v)$ needs not to be unique

- **Intuition:**
  - The identifier is the key of the node.
  - The label is the data carried by the node.

- **Example:**
  $$T = \{(1, a), (3, c), (5, b), (4, c), (7, d)\},
  \{(1, 3), (1, 5), (5, 4), (5, 7)\}$$

```
(1, a)
  /   \
(3, c) (5, b)
     /   \
(4, c) (7, d)
```
What is a Tree?

Notation and Graphical Representation

Notation:
- node identifiers: \( \text{id}(v_i) = i \)
- tree identifiers: \( T_1, T_2, \ldots \)

Graphical representation:
- we omit brackets for (identifier,label)-pairs
- we (sometimes) omit node identifiers at all
- we do not show the direction of edges
  (edges are always directed from root to leave)

unlabeled tree | edge labeled tree | node labeled tree
--- | --- | ---
[Diagram of unlabeled tree] | [Diagram of edge labeled tree] | [Diagram of node labeled tree]
Ordered Trees

- Ordered Trees: siblings are ordered
- contiguous siblings $s_1 < s_2$ have no sibling $x$ such that $s_1 < x < s_2$
- $c_i$ is the $i$-th child of $p$ if
  - $p$ is the parent of $c_i$, and
  - $i = |\{x \in N(T) : (p, x) \in E(T), x \leq c_i\}|$

Example:

Unordered Trees

```
      a
     / \  \
    c   b  d
   / \    /
  e   f  e
```

Ordered Trees

```
      a
     / \  \
    c   b  d
   / \  / \ \
  e   f f e
```

Note: “ordered” does not necessarily mean “sorted alphabetically”
What is a Tree?

Edit Operations

- **We assume** ordered, labeled trees
- **Rename node**: \( \text{ren}(v, l') \)
  - change label \( l \) of \( v \) to \( l' \neq l \)
- **Delete node**: \( \text{del}(v) \) (\( v \) is not the root node)
  - remove \( v \)
  - connect \( v \)’s children directly to \( v \)’s parent node (preserving order)
- **Insert node**: \( \text{ins}(v, p, k, m) \)
  - remove \( m \) consecutive children of \( p \), starting with the child at position \( k \), i.e., the children \( c_k, c_{k+1}, \ldots, c_{k+m-1} \)
  - insert \( c_k, c_{k+1}, \ldots, c_{k+m-1} \) as children of the new node \( v \) (preserving order)
  - insert new node \( v \) as \( k \)-th child of \( p \)
- **Insert and delete are** inverse edit operations (i.e., insert undoes delete and vice versa)
What is a Tree?

Example: Edit Operations

$T_0 \xrightarrow{ins((v_5, b), v_1, 2, 2)} T_1 \xleftarrow{del(v_5)} T_2$

$v_1, a$
$v_3, c$
$v_4, c$
$v_7, d$

$\xrightarrow{ren(v_4, x)}$

$v_1, a$
$v_3, c$
$v_4, c$
$v_5, b$
$v_7, d$

$\leftarrow ren(v_4, c)$

$v_1, a$
$v_3, c$
$v_5, b$
$v_4, x$
$v_7, d$
Outline

1. What is a Tree?

2. Encoding XML in a Relational Database
   - Adjacency List Encoding
   - Dewey Encoding
   - Interval Encoding
   - Experimental Comparison of the Encodings
   - XML and Trees
Outline

1 What is a Tree?

2 Encoding XML in a Relational Database
   - Adjacency List Encoding
   - Dewey Encoding
   - Interval Encoding
   - Experimental Comparison of the Encodings
   - XML and Trees
Motivation: Trees and Relational Databases

- **Relational Databases:**
  - highly developed systems
  - mature storage and querying capabilities

- **But:** there is a gap between ordered trees and relations
  - relations are *sets* (no order)
  - relations store *tuples* (no hierarchy)

- How can we store an (ordered) tree in a relation?
Adjacency List

- **Adjacency List:**
  - list of nodes
  - each node stores pointer to parent

- **Relational Implementation:**
  - node is tuple \((nid, pid)\)
  - \(nid\) the node ID
  - \(pid\) the node ID of the parent node

- **Example:**

  Tree | Adjacency List | Relational Implementation
  --- | --- | ---
  ![Tree Diagram](image) | ![Node List](image) | \[
  \begin{array}{|c|c|}
  \hline
  \text{nid} & \text{pid} \\
  \hline
  1 & @ \\
  3 & 1 \\
  5 & 1 \\
  4 & 5 \\
  7 & 5 \\
  \hline
  \end{array}
  \]
Extending the Adjacency List Model

- Node labeled trees: \((v, p, \lambda(v))\)
  - \(v, p \in N(T)\) are nodes
  - \(v\) is a child of \(p\)
  - \(\lambda(v)\) is the label of \(v\)

- Edge labeled trees: \((v, p, \lambda((p, v)))\)
  - \(v, p \in N(T)\) are nodes
  - \((p, v) \in E(T)\) is an edge
  - \(\lambda((p, v))\) is the label of the edge \((p, v)\)

- Ordered trees: \((v, p, i)\)
  - \(v, p \in N(T)\) are nodes
  - \(v\) is the \(i\)-th child of \(p\)

- All combinations possible...
Edit Operations with the Adjacency List Encoding

- **Tree relation** $T(nid, pid, lbl, pos)$
- **Rename**: $ren(v, l')$
  - update single tuple $(v, p, l, i) \rightarrow (v, p, l', i)$
- **Delete node**: $del(v)$
  - delete single tuple
  - update right siblings and all children of $v$
- **Insert node**: $ins(v, p, k, m)$
  - insert single tuple
  - update right siblings ($pos \geq k$) and all children of new node $v$
Example: Delete Node in Adjacency Encoding

- Delete node \( v \) with \( \text{id}(v) = 4 \)
  - delete single tuple
  - update children of \( v \)
  - update right siblings of \( v \)

<table>
<thead>
<tr>
<th>( \text{nid} )</th>
<th>( \text{pid} )</th>
<th>( \text{pos} )</th>
<th>( \text{lbl} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>d</td>
</tr>
<tr>
<td>4 (red)</td>
<td>2</td>
<td>2</td>
<td>e (red)</td>
</tr>
<tr>
<td>5 (red)</td>
<td>2</td>
<td>2</td>
<td>f (red)</td>
</tr>
<tr>
<td>6 (red)</td>
<td>2</td>
<td>3</td>
<td>g (red)</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>1</td>
<td>h</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>2</td>
<td>i</td>
</tr>
<tr>
<td>9 (red)</td>
<td>2</td>
<td>4</td>
<td>i (red)</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>1</td>
<td>l</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td>2</td>
<td>m</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>5</td>
<td>n</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>3</td>
<td>o</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>1</td>
<td>p</td>
</tr>
</tbody>
</table>
Update Efficiency

- **Worst case**: all children of v and of p must be updated
  - $O(f_{max})$ node updates, where $f_{max}$ is the maximum fanout in the tree
  - $f_{max}$ typically small compared to tree size
  - update *very efficient*

- **Implementation hints**:
  - unique index on $nid$ and on $(pid, pos)$ will speed up queries
  - use ...ORDER BY pos ASC/DESC in update statement to avoid duplicates
Preorder Traversal

- **Preorder:** in XML also “document order”
  - visit root
  - traverse subtrees rooted in children (from left to right) in preorder
- **Example:** preorder = (a, d, f, e, c, b)

- **Implementation:**
  - start with root
  - recursively select children of root

- **Efficiency:**
  - children of all ancestors on recursion-stack
  - \(O(n)\) queries for children — very inefficient
Outline

1. What is a Tree?

2. Encoding XML in a Relational Database
   - Adjacency List Encoding
   - Dewey Encoding
   - Interval Encoding
   - Experimental Comparison of the Encodings
   - XML and Trees
Dewey Encoding

- **Dewey Decimal Classification:**
  - used in libraries to classify books by topics
  - developed by Melvil Dewey in 1876
- **Dewey Encoding**\(^1\) [TVB\(^+\)02]:
  - list of nodes
  - each node stores path from the root
- **Example:**

<table>
<thead>
<tr>
<th>nid</th>
<th>dp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1.1</td>
</tr>
<tr>
<td>5</td>
<td>1.2</td>
</tr>
<tr>
<td>4</td>
<td>1.2.1</td>
</tr>
<tr>
<td>7</td>
<td>1.2.2</td>
</tr>
</tbody>
</table>

\(^1\)also “Edge Enumeration” [Cel04]
About the Dewey Paths

“◦” concatenates a Dewey path $dp$ with an integer $i$ (sibling position) e.g., $1.2◦2 = 1.2.2$

Sort order: $1.2 < 1.3, 1.1 < 1.1.2, 1.9 < 1.10$
Dewey encoding implicitly orders trees!

- **Node labeled trees:** \((v, dp, \lambda(v))\)
  - \(v \in N(T)\) is a node ID
  - \(dp\) is the Dewey path to \(v\)
  - \(\lambda(v)\) is the label of \(v\)

- **Edge labeled trees:** \((v, dp, \lambda)\)
  - \(v \in N(T)\) is a node ID
  - \(dp\) is the Dewey path to \(v\)
  - \(\lambda\) is the label of the edge from the parent of \(v\) to \(v\)
Edit Operations with the Dewey Encoding

- **Tree relation** $T(nid, dp, lbl)$
- **Rename node**: $\text{ren}(v, l')$
  - update single tuple $(v, dp, l) \rightarrow (v, dp, l')$
  - no structure updates
- **Delete node**: $\text{del}(v)$
  - remove single tuple $(v, dp_v, l)$
  - update nodes with $dp > dp_v$
    - (descendants of $v$ and descendants $v$’s right-hand siblings)
- **Insert node**: $\text{ins}(v, p, k, m)$
  - update nodes with $dp \geq dp(p) \circ k$
    - (children of $p$ at position $k$ or larger, and all their descendants)
  - insert single tuple $(v, dp(p) \circ k, \lambda(v))$
- **Efficiency**:
  - $O(n)$ in the worst case (insert/delete leftmost child of root node)
  - better for nodes with (i) few descendants and (ii) few right siblings
  - $O(1)$ for lonely leaf child of a node
Example: Delete Node in Dewey Encoding

- Delete node $v$ with $id(v) = 4$
  - delete single tuple
  - update descendants of $v$
  - update right siblings of $v$ and their descendants

<table>
<thead>
<tr>
<th>nid</th>
<th>dp</th>
<th>lbl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>1.2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>1.2.1</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>1.2.2</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>1.2.2.1</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>1.2.2.1</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>1.2.2.1</td>
<td>h</td>
</tr>
<tr>
<td>8</td>
<td>1.2.2.1</td>
<td>i</td>
</tr>
<tr>
<td>9</td>
<td>1.2.3</td>
<td>k</td>
</tr>
<tr>
<td>10</td>
<td>1.2.3.1</td>
<td>l</td>
</tr>
<tr>
<td>11</td>
<td>1.2.3.2</td>
<td>m</td>
</tr>
<tr>
<td>12</td>
<td>1.2.4</td>
<td>n</td>
</tr>
<tr>
<td>13</td>
<td>1.3</td>
<td>o</td>
</tr>
<tr>
<td>14</td>
<td>1.3.1</td>
<td>p</td>
</tr>
</tbody>
</table>
Preorder

Tree relation \( T(nid, dp, lbl) \)

Implementation:
- sort by attribute \( dp \)
- result is preorder traversal

Efficiency:
- single query with sort on string attribute
- efficient (especially with index on \( dp \))
Goals:
- minimize space overhead for Dewey path $dp$
- sorting Dewey path should result in preorder traversal

Separator character: e.g., 1.2.5, 1.17
- overhead: small (separator char)
- sorting: natural sort order not consistent with preorder ($1.2.5 > 1.17$)

Fixed length: e.g., 0001 0002 0005, 0001 0017
- overhead: large (small and large numbers require same space)
- sorting: sort order ok

Variable length encoding (UTF-8):
- UTF-8: 1 byte: $0 \ldots (2^7 - 1)$, 2 bytes: $2^7 \ldots (2^{11} - 1)$, etc.
- overhead: small space overhead
- sorting: sort order ok (supported by many databases, e.g. PostgreSQL)
Outline

1. What is a Tree?

2. Encoding XML in a Relational Database
   - Adjacency List Encoding
   - Dewey Encoding
   - Interval Encoding
   - Experimental Comparison of the Encodings
   - XML and Trees
Interval Encoding [DTCÖ03, ABG05]

- **Idea:** Parent “contains” children, like interval contains other intervals
- **Example:**

```
  a
 / \
c   b
|   |
c   c  d
```

- **Interval Encoding:**
  - assign numbers to interval start and end points
  - store interval start and end point with each node
Definition (Interval Encoding)

An interval encoding of a tree is a relation $T$ that for each node $v$ of the tree contains a tuple $(\lambda(v), \text{lft}, \text{rgt})$; $\lambda(v)$ is the label of $v$, $\text{lft}$ and $\text{rgt}$ are the endpoints of the interval representing the node. $\text{lft}$ and $\text{rgt}$ are constrained as follows:

- $\text{lft} < \text{rgt}$ for all $(\text{lbl}, \text{lft}, \text{rgt}) \in T$,
- $\text{lft}_a < \text{lft}_d$ and $\text{rgt}_a > \text{rgt}_d$ if node $a$ is an ancestor of $d$, and $(\lambda(a), \text{lft}_a, \text{rgt}_a) \in T$, and $(\lambda(d), \text{lft}_d, \text{rgt}_d) \in T$,
- $\text{rgt}_v < \text{lft}_w$ if node $v$ is a left sibling of node $w$, and $(\lambda(v), \text{lft}_v, \text{rgt}_v) \in T$, and $(\lambda(w), \text{lft}_w, \text{rgt}_w) \in T$, 
Example

Example algorithm for a valid interval encoding:
- traverse tree in preorder
- use an incremental counter
- assign left interval value \( lft \) when node is first visited
- assign right interval value \( rgt \) when node is last visited

```
1 a
  \( lft = 10 \)
  \( rgt = 10 \)
2 c
  \( lft = 3 \)
  \( rgt = 3 \)
3 b
  \( lft = 9 \)
  \( rgt = 9 \)
4 d
  \( lft = 7 \)
  \( rgt = 7 \)
5 c
  \( lft = 6 \)
  \( rgt = 6 \)
6
7 d
  \( lft = 8 \)
  \( rgt = 8 \)
```
Edit Operations with the Interval Encoding

- **Tree relation**  $T(id, lbl, lft, rgt)$
- **Rename node**: $\text{ren}(v, l')$
  - update single tuple $(id(v), l, L, R) \rightarrow (id(v), l', L, R)$
  - no structure updates
- **Delete node**: $\text{del}(v)$
  - remove single tuple $(id(v), l, L, R)$
  - remaining tree is valid and correct
- **Insert node**: $\text{ins}(v, p, k, m)$
  - find left and right interval values $L$ and $R$
  - if values not free, update ancestors and nodes following in preorder
  - insert single tuple $(id(v), \lambda(v), L, R)$

**Efficiency**:
- rename and delete are very efficient (constant time)!
- insert may be $O(n)$ in worst case (inefficient)
- sparse numbering reduces number of updates for insert
Delete node $v$ with $id(v) = 4$

- delete single tuple
- remaining tree is valid and correct
Example: Insert Node in Interval Encoding

Insert new node with label e:
\[ \text{ins}((4, e), 2, 2, 3) \]

- update the ancestors of the new node
- update the nodes following the new node in preorder
- insert single tuple
Preorder

- Tree relation $T(\text{id, lbl, lft, rgt})$
- Implementation:
  - sort by attribute $\text{lft}$
  - result is preorder traversal
- Efficiency:
  - single query with sort on integer attribute
  - very efficient (especially with index on $\text{lft}$)
Interval Encoding with sparse numbering:
- leave numbers free for future insert
- avoids global reordering until gaps are filled
- node deletions re-open gaps

Example:

```
  0  a  90
  /    \
10 c  20  30 b  80
  /  \
40 c  50  60 d  70
```

Note: Floating-point values do not solve the problem!

Sparse numbering using \((order, size)\)-pairs [LM01]:
- store node position as \((order, size)\)-pair
- \(order\) corresponds to left interval value
- \(order + size\) corresponds to right interval value

Example:
Inserting a node:
   a) find the correct gap(s) in the tree
   b) if the/each gap is large enough: insert new node
   c) otherwise: ...?

Solution 1: shift left/right values until new node fits
   cheapest way for inserting a single node
   but: only a small number of gaps are opened

Solution 2: reset all gaps
   more expensive than shifting
   but: happens less frequently because all gaps in the tree are opened

Shifting or resetting gaps are called “hard updates”
Shifting Gaps is Cheaper than Resetting All Gaps

Runtime of shifting and resetting:
- "Sparse+": resets all gaps
- "Sparse": shifts gaps

(Graph from [Dag08])
Average number of hard updates when a new node is inserted (gap size 100):
- “Sparse+”: resets all gaps
- “Sparse”: shifts gaps
Impact of the gap size on the number of hard updates:

- “Sparse+”: resets all gaps
- “Sparse”: shifts gaps

(Graph from [Dag08])
Outline

1. What is a Tree?

2. Encoding XML in a Relational Database
   - Adjacency List Encoding
   - Dewey Encoding
   - Interval Encoding
   - Experimental Comparison of the Encodings
   - XML and Trees
Delete performance of Adjacency List, Dewey, and Interval Encoding (Sparse+ [Dag08], gap size 100)

- Each data point in graph shows avg. runtime over 800 deletions
- Descendants: avg. number of descendants of deleted nodes
- Fanout: avg. fanout of deleted nodes
Insert Performance

- Insert performance of Adjacency List, Dewey, and Interval Encoding (Sparse+ [Dag08], gap size 100)
- Each data point in graph shows avg. runtime over 800 insertions
- Descendants: avg. number of descendants of inserted nodes
- Fanout: avg. fanout of inserted nodes

(Graph from [Dag08])
Efficiency of the Preorder Traversal

- Preorder traversal performance of Adjacency List, Dewey, and Interval Encoding (Sparse+ [Dag08], gap size 100)

(Graph from [Dag08])
## Comparing the Encodings

<table>
<thead>
<tr>
<th></th>
<th>Adjacency</th>
<th>Dewey</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>+</strong></td>
<td>update very efficient</td>
<td>preorder efficient</td>
<td>preorder very efficient</td>
</tr>
<tr>
<td></td>
<td>simple implementation</td>
<td>update efficiency: between</td>
<td>simple implementation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>others</td>
<td></td>
</tr>
<tr>
<td><strong>−</strong></td>
<td>preorder very inefficient</td>
<td>update worst case is $O(n)$</td>
<td>insert is $O(n)$ on average</td>
</tr>
<tr>
<td></td>
<td></td>
<td>space overhead for storing</td>
<td>(patch: sparse numbering)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>paths</td>
<td></td>
</tr>
</tbody>
</table>
Outline

1. What is a Tree?

2. Encoding XML in a Relational Database
   - Adjacency List Encoding
   - Dewey Encoding
   - Interval Encoding
   - Experimental Comparison of the Encodings
   - XML and Trees
Many possibilities – we will consider
- single-label tree
- double-label tree

Pros/cons depend on application!
XML as a Single-Label Tree

- The XML document is stored as a tree with:
  - XML element: node labeled with element tag name
  - XML attribute: node labeled with attribute name
  - Text contained in elements/attributes: node labeled with the text-value

- Element nodes contain:
  - nodes of their sub-elements
  - nodes of their attributes
  - nodes with their text values

- Attribute nodes contain:
  - single node with their text value

- Text nodes are always leaves

- Order:
  - sub-element and text nodes are ordered
  - attributes are not ordered (approach: store them before all sub-elements, sort according to attribute name)
Example: XML as a Single-Label Tree

<article title='pq-Grams'>
  <author>Augsten</author>
  <author>Boehlen</author>
  <author>Gamper</author>
</article>
XML as a Double-Label Tree

- Node labels are pairs
- The XML document is stored as a tree with:
  - XML element: node labeled with (tag-name, text-value)
  - XML attribute: node labeled with (attribute-name, text-value)
- Element nodes contain:
  - nodes of their sub-elements and attributes
- Attribute nodes are always leaves
- Element nodes without attributes or sub-elements are leaves
- Order:
  - sub-element nodes are ordered
  - attributes are not ordered (approach: see previous slide)
- Limitation: Can represent
  - *either* elements with sub-elements and/or attributes
  - *or* elements with a text value
Example: XML as a Double-Label Tree

<article title='pq-Grams'>
  <author>Augsten</author>
  <author>Boehlen</author>
  <author>Gamper</author>
</article>

(article, ε)

(title, pq-Grams) (author, Augsten) (author, Boehlen) (author, Gamper)
Example: Single- vs. Double-Label Tree

<xhtml>
  <p>This is <b>bold</b> font.</p>
</xhtml>

Single-Label Tree

```
  <xhtml>
    <p>This is <b>bold</b> font.</p>
  </xhtml>
```

Double-Label Tree

```
(xhtml, ε)
(p, ?)
(b, bold)
```
We discuss two popular parsers for XML:

- DOM – Document Object Model
- SAX – Simple API for XML
DOM – Document Object Model

- W3C\(^2\) standard for accessing and manipulating XML documents
- Tree-based: represents an XML document as a tree
  (single-label tree with additional node info, e.g. node type)
- Elements, attributes, and text values are nodes
- DOM parsers load XML into main memory
  - random access by traversing tree :-)
  - large XML documents do not fit into main memory :-(

\(^2\)http://www.w3schools.com/dom
“de facto” standard for parsing XML

- Event-based: reports parsing events (e.g., start and end of elements)
  - no random access :-(
  - you see only one element/attribute at a time
  - you can parse (arbitrarily) large XML documents :-(

- Java API available for both, DOM and SAX
- For importing XML into a database: use SAX!

---

3http://www.saxproject.org

Nikolaus Augsten (DIS)


David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A comprehensive XQuery to SQL translation using dynamic interval encoding.