Similarity Search

Trees and Relational Databases

Nikolaus Augsten

nikolaus.augsten@sbg.ac.at

Dept. of Computer Sciences
University of Salzburg
http://dbresearch.uni-salzburg.at

Version November 16, 2016

Wintersemester 2016/2017

Outline

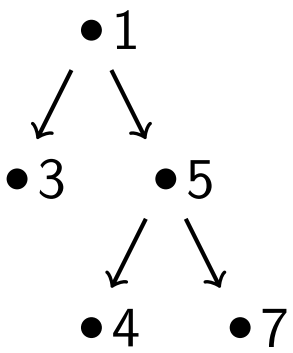
Outline

What is a Tree?

- 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: $E(\mathrm{T})$ are the edges of T
  - an edge $(\mathrm{p}, \mathrm{c}) \in E(\mathrm{T})$ is an ordered pair
  - with $\mathrm{p}, \mathrm{c} \in N(\mathrm{T})$
- "Special" Nodes: $N(\mathrm{T})$ are the nodes of T
  - parent/child: $(\mathrm{p}, \mathrm{c}) \in E(T) \Leftrightarrow$ 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

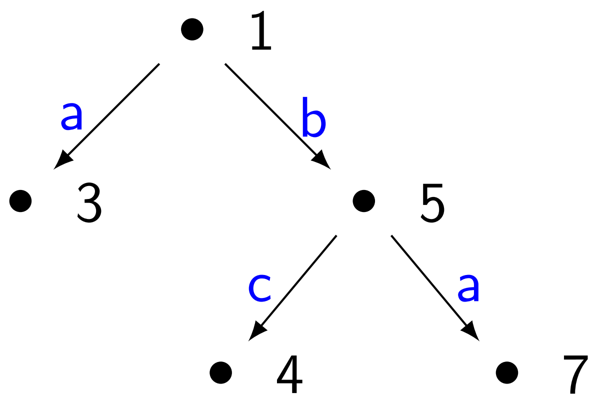
- Unlabeled Tree:
  - the focus is on the structure, not on distinguishing nodes
  - however, we need to distinguish nodes in order to define edges
    $\Rightarrow$ each node v has a unique identifier id(v) within the tree
- Example: $T = (\{1, 3, 5, 4, 7\}, \{(1, 3), (1, 5), (5, 4), (5, 7)\})$



Edge Labeled Trees

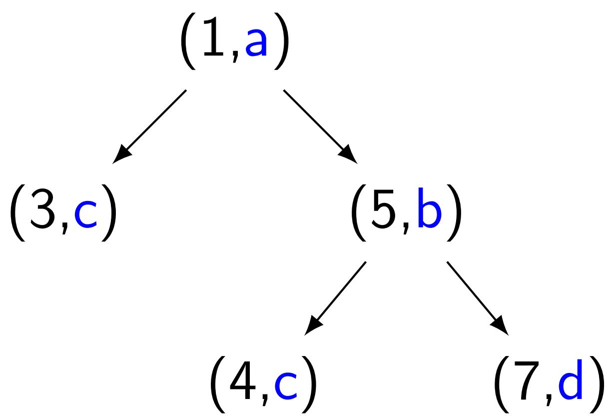
- Edge Labeled Tree:
  - an edge $e \in E(\mathrm{T})$ between nodes a and b is a triple $e = (\mathrm{id}(\mathrm{a}), \mathrm{id}(\mathrm{b}), \lambda(e))$
  - id(a) and id(b) are node IDs
  - $\lambda(e)$ is the edge label (not necessarily unique within the tree)
- Example:

$$\mathrm{T} = (\{1, 3, 5, 4, 7\}, \{(1, 3, \mathtt{a}), (1, 5, \mathtt{b}), (5, 4, \mathtt{c}), (5, 7, \mathtt{a})\})$$



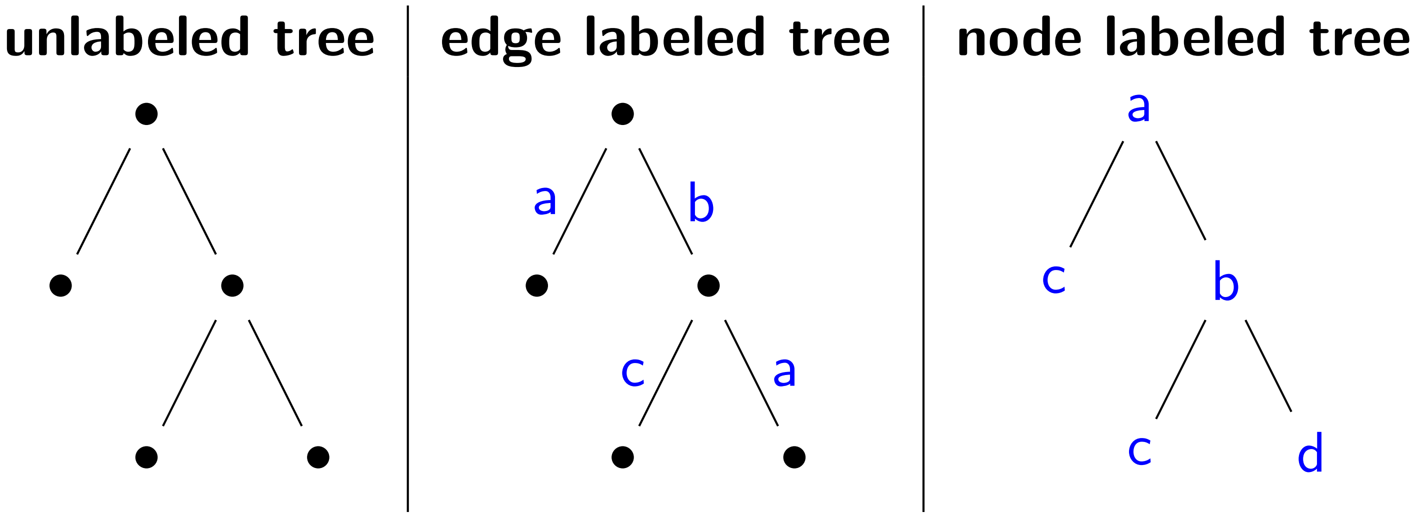
Node Labeled Trees

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



Notation and Graphical Representation

- Notation:
  - node identifiers: $\mathrm{id}(\mathrm{v}_i) = i$
  - tree identifiers: $\mathrm{T}_1, \mathrm{T}_2, \ldots$
- 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)



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
  - $i = |\{x \in N(T) : (p, x) \in E(T), x \le c_i\}|$
- Example:

Unordered Trees | **Ordered Trees**

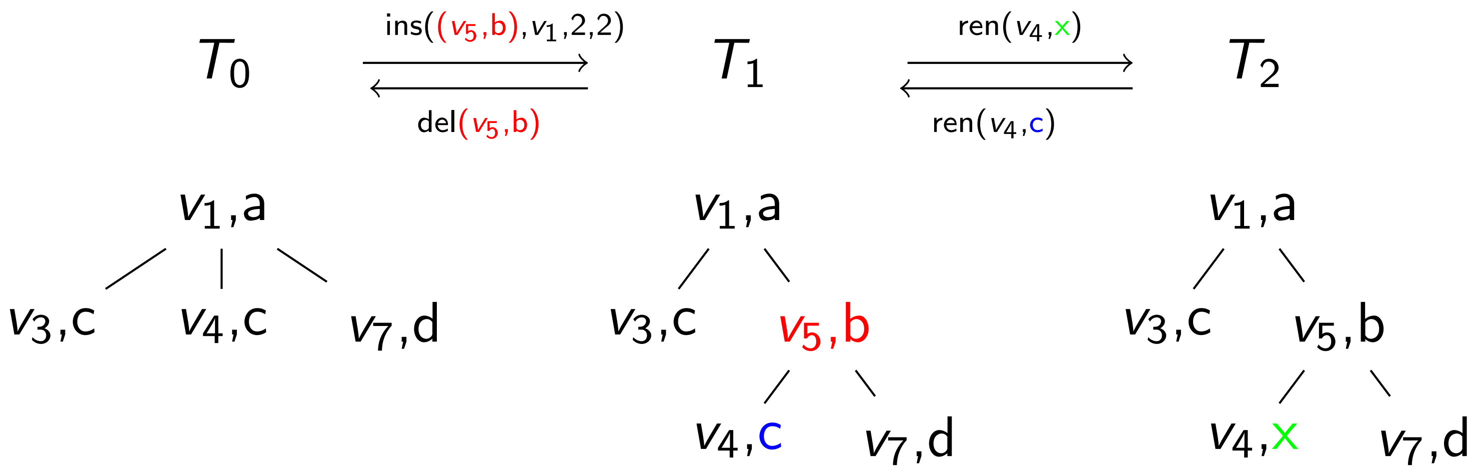
```
     a               a       |      a               a
    /|\             /|\      |     /|\             /|\
   c b d     =     d b c     |    c b d     ≠     d b c
        /\        /\         |         /\        /\
       e  f      f  e        |        e  f      f  e
```

- Note: “ordered” does not necessarily mean “sorted alphabetically”

Edit Operations

- We assume ordered, labeled trees
- Rename node: $ren(\mathrm{v}, l')$
  - change label l of v to $l' \neq l$
- Delete node: $del(\mathrm{v})$ (v is not the root node)
  - remove v
  - connect v's children directly to v's parent node (preserving order)
- Insert node: $ins(\mathrm{v}, \mathrm{p}, k, m)$
  - remove m consecutive children of p, starting with the child at position k, i.e., the children $\mathrm{c}_k, \mathrm{c}_{k+1}, \ldots, \mathrm{c}_{k+m-1}$
  - insert $\mathrm{c}_k, \mathrm{c}_{k+1}, \ldots, \mathrm{c}_{k+m-1}$ 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)

Example: Edit Operations



Outline

Outline

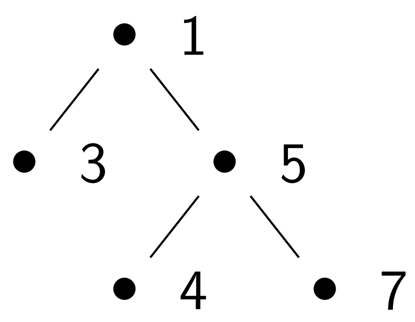
Motivation: Trees and Relational Databases

- 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?

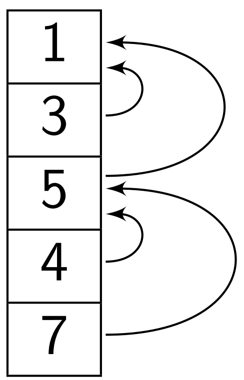
Adjacency List

- 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 implementatic

nid	pid
1	@
3	1
5	1
4	5
7	5

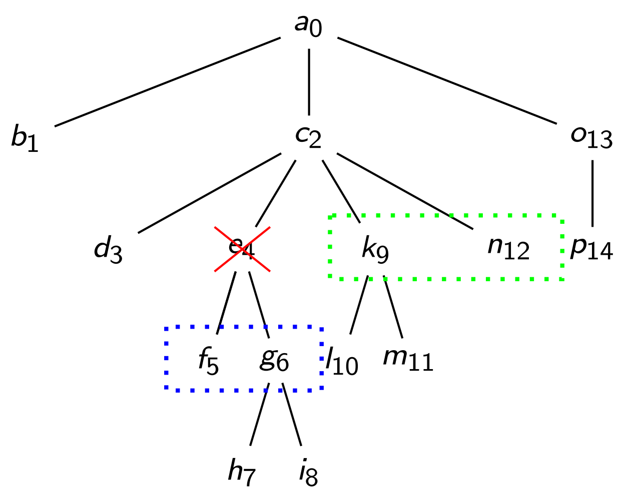
Extending the Adjacency List Model

- Node labeled trees: $(\mathrm{v}, \mathrm{p}, \lambda(\mathrm{v}))$
  - $\mathrm{v}, \mathrm{p} \in N(\mathrm{T})$ are nodes
  - v is a child of p
  - $\lambda(\mathrm{v})$ is the label of v
- Edge labeled trees: $(\mathrm{v}, \mathrm{p}, \lambda((\mathrm{p}, \mathrm{v})))$
  - $\mathrm{v}, \mathrm{p} \in N(\mathrm{T})$ are nodes
  - $(\mathrm{p}, \mathrm{v}) \in E(\mathrm{T})$ is an edge
  - $\lambda((\mathrm{p}, \mathrm{v}))$ is the label of the edge (p, v)
- Ordered trees: $(\mathrm{v}, \mathrm{p}, i)$
  - $\mathrm{v}, \mathrm{p} \in N(\mathrm{T})$ 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: $ren(\text{v}, l')$
  - update single tuple $(\text{v}, \text{p}, l, i) \rightarrow (\text{v}, \text{p}, l', i)$
- Delete node: $del(\text{v})$
  - delete single tuple
  - update right siblings and all children of v
- Insert node: $ins(\text{v}, \text{p}, k, m)$
  - insert single tuple
  - update right siblings ($pos \geq k$) and all children of new node v

Example: Delete Node in Adjacency Encoding



nid	pid	pos	lbl
0	-	-	a
1	0	1	b
2	0	2	c
3	2	1	d
~~4~~	~~2~~	~~2~~	~~e~~
5	~~4~~ 2	~~1~~ 2	f
6	~~4~~ 2	~~2~~ 3	g
7	6	1	h
8	6	2	i
9	2	~~3~~ 4	k
10	9	1	l
11	9	2	m
12	2	~~4~~ 5	n
13	0	3	o
14	13	1	p

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 $= (\texttt{a}, \texttt{d}, \texttt{f}, \texttt{e}, \texttt{c}, \texttt{b})$

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

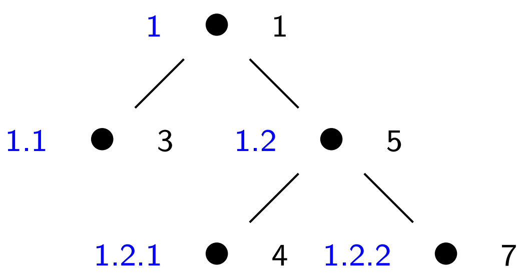
- Implementation:
  - start with root
  - recursively select children of root
- Efficiency:
  - children of all ancestors on recursion-stack
  - $O(n)$ queries for children — very inefficient

Outline

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:

tree

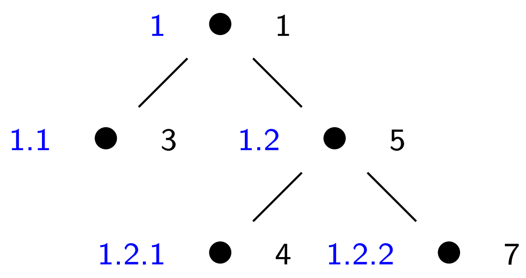


relational implementation

nid	pid
1	1
3	1.1
5	1.2
4	1.2.1
7	1.2.2

[1] also "Edge Enumeration" [Cel04]

About the Dewey Paths



- "$\circ$" concatenates a Dewey path *dp* with an integer *i* (sibling position) e.g., $1.2 \circ 2 = 1.2.2$
- Sort order: $1.2 < 1.3$, $1.1 < 1.1.2$, $1.9 < 1.10$

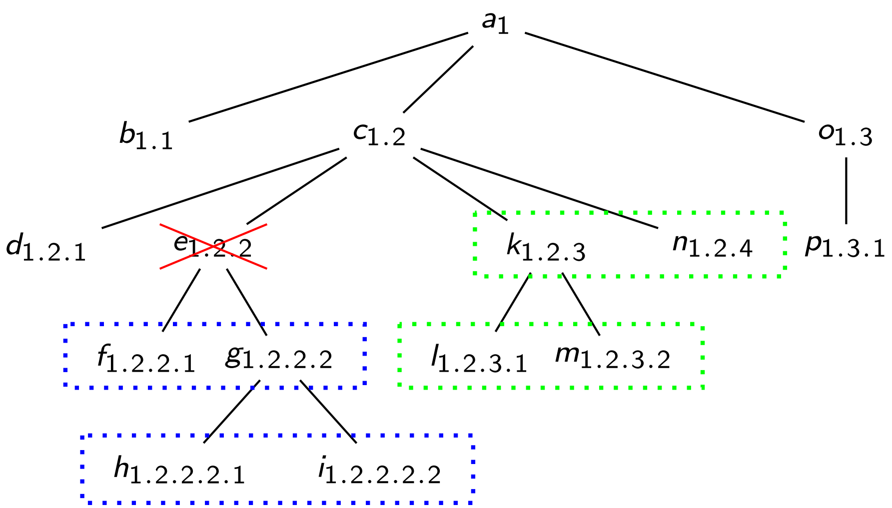
Extending the Dewey Encoding

- Dewey encoding implicitly orders trees!
- Node labeled trees: $(\mathrm{v}, dp, \lambda(\mathrm{v}))$
  - $\mathrm{v} \in N(\mathrm{T})$ is a node ID
  - dp is the Dewey path to v
  - $\lambda(\mathrm{v})$ is the label of v
- Edge labeled trees: $(\mathrm{v}, dp, \lambda)$
  - $\mathrm{v} \in N(\mathrm{T})$ is a node ID
  - dp is the Dewey path to v
  - λ 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: $ren(\text{v}, l')$
  - update single tuple $(\text{v}, dp, l) \rightarrow (\text{v}, dp, l')$
  - no structure updates
- Delete node: $del(\text{v})$
  - remove single tuple $(\text{v}, dp_{\text{v}}, l)$
  - update nodes with $dp > dp_{\text{v}}$ (descendants of v and descendants v's right-hand siblings)
- Insert node: $ins(\text{v}, \text{p}, k, m)$
  - update nodes with $dp \geq dp(\text{p}) \circ k$ (children of p at position k or larger, and all their descendants)
  - insert single tuple $(\text{v}, dp(\text{p}) \circ k, \lambda(\text{v}))$
- 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



nid	dp	lbl
0	1	a
1	1.1	b
2	1.2	c
3	1.2.1	d
~~4~~	~~1.2.2~~	~~e~~
5	~~1.2.2.1~~ 1.2.2	f
6	~~1.2.2.2~~ 1.2.3	g
7	~~1.2.2.2.1~~ 1.2.3.1	h
8	~~1.2.2.2.2~~ 1.2.3.2	i
9	~~1.2.3~~ 1.2.4	k
10	~~1.2.3.1~~ 1.2.4.1	l
11	~~1.2.3.2~~ 1.2.4.2	m
12	~~1.2.4~~ 1.2.5	n
13	1.3	o
14	1.3.1	p

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*)

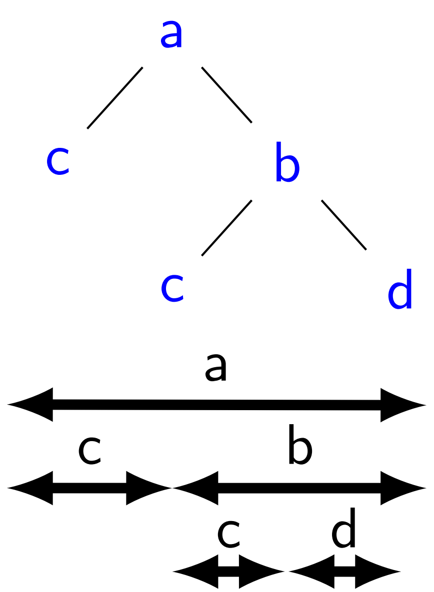
Implementation: Storing the Dewey Path

- 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

Interval Encoding [DTCÖ03, ABG05]

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



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

Interval Encoding

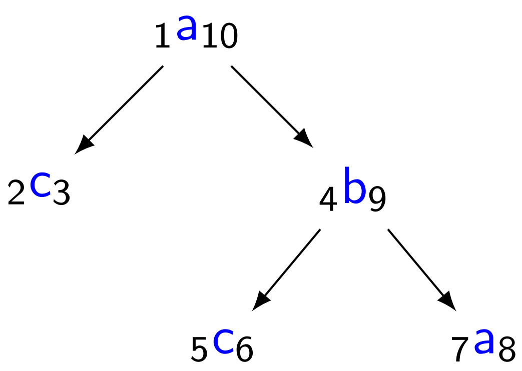
Definition (Interval Encoding)

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

- $\mathit{lft} < \mathit{rgt}$ for all $(\mathit{lbl}, \mathit{lft}, \mathit{rgt}) \in T$,
- $\mathit{lft}_{\mathrm{a}} < \mathit{lft}_{\mathrm{d}}$ and $\mathit{rgt}_{\mathrm{a}} > \mathit{rgt}_{\mathrm{d}}$ if node a is an ancestor of d, and $(\lambda(\mathrm{a}), \mathit{lft}_{\mathrm{a}}, \mathit{rgt}_{\mathrm{a}}) \in T$, and $(\lambda(\mathrm{d}), \mathit{lft}_{\mathrm{d}}, \mathit{rgt}_{\mathrm{d}}) \in T$,
- $\mathit{rgt}_{\mathrm{v}} < \mathit{lft}_{\mathrm{w}}$ if node v is a left sibling of node w, and $(\lambda(\mathrm{v}), \mathit{lft}_{\mathrm{v}}, \mathit{rgt}_{\mathrm{v}}) \in T$, and $(\lambda(\mathrm{w}), \mathit{lft}_{\mathrm{w}}, \mathit{rgt}_{\mathrm{w}}) \in T$,

Example

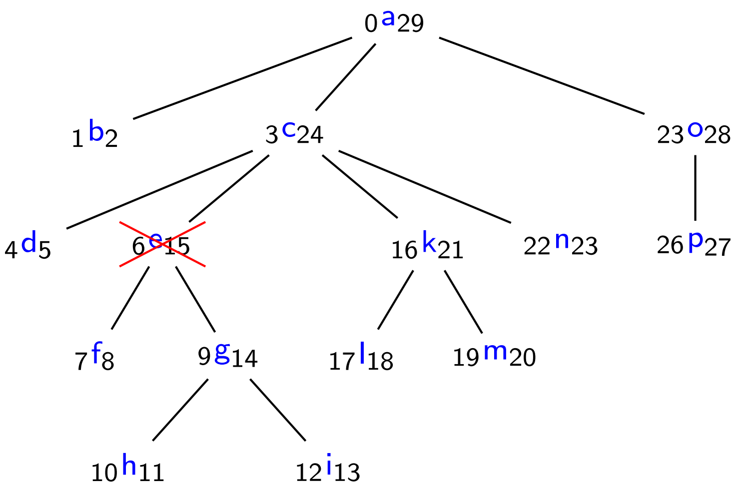
- Example algorithm for a valid interval encoding:
  - traverse tree in preorder
  - use an incremental counter
  - assign left interval value *lft* when node is first visited
  - assign right interval value *rgt* when node is last visited



Edit Operations with the Interval Encoding

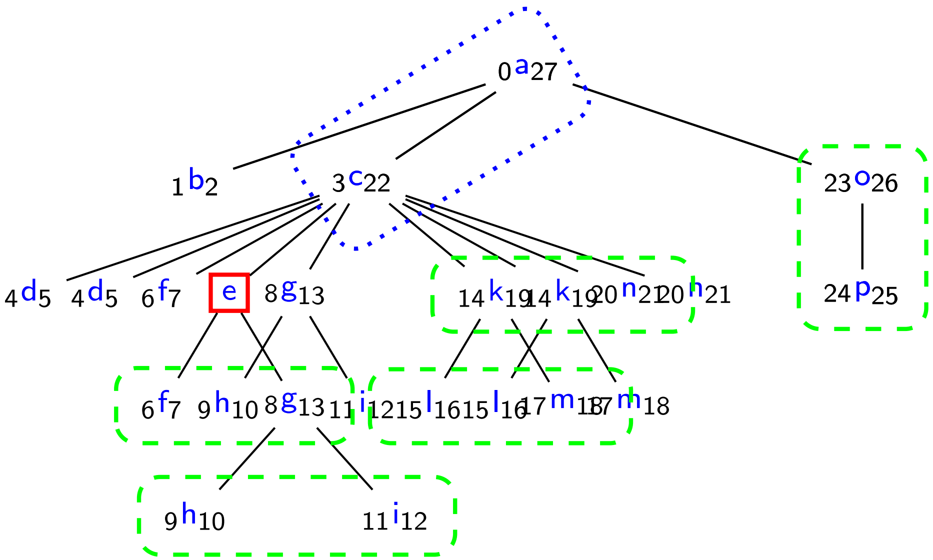
- Tree relation $T(id, lbl, lft, rgt)$
- Rename node: $ren(\text{v}, l')$
  - update single tuple $(\text{id}(\text{v}), l, L, R) \rightarrow (\text{id}(\text{v}), l', L, R)$
  - no structure updates
- Delete node: $del(\text{v})$
  - remove single tuple $(\text{id}(\text{v}), l, L, R)$
  - remaining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

Example: Delete Node in Interval Encoding



nid	lbl	lft	rgt
0	a	0	29
1	b	1	2
2	c	3	24
3	d	4	5
~~4~~	~~e~~	~~6~~	~~15~~
5	f	7	8
6	g	9	14
7	h	10	11
8	i	12	13
9	k	16	21
10	l	17	18
11	m	19	20
12	n	22	23
13	o	25	28
14	p	26	27

Example: Insert Node in Interval Encoding



nid	lbl	lft	rgt
0	a	0	~~27~~ 29
1	b	1	2
2	c	3	~~22~~ 24
3	d	4	5
5	f	~~6~~ 7	~~7~~ 8
6	g	~~8~~ 9	~~13~~ 14
7	h	~~9~~ 10	~~10~~ 11
8	i	~~11~~ 12	~~12~~ 13
9	k	~~14~~ 16	~~19~~ 21
10	l	~~15~~ 17	~~16~~ 18
11	m	~~17~~ 19	~~18~~ 20
12	n	~~20~~ 22	~~21~~ 23
13	o	~~23~~ 25	~~26~~ 28
14	p	~~24~~ 26	~~25~~ 27
4	e	6	15

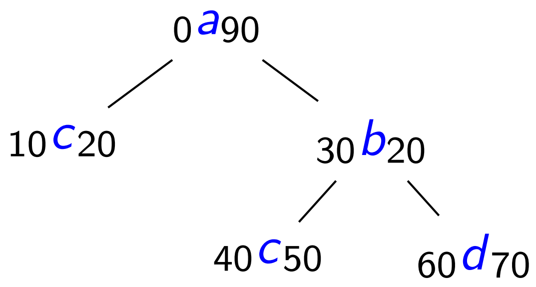
- Insert new node with label e: ins((4, e), 2, 2, 3)
  - update the ancestors of the new node
  - update the nodes following the new node in preorder
  - insert single tuple

Preorder

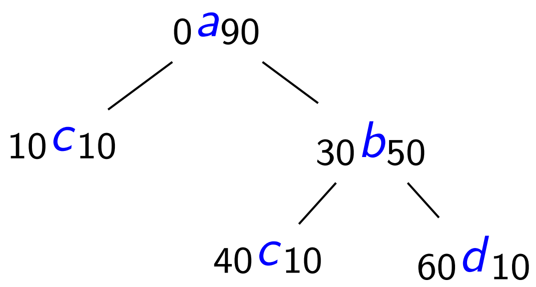
- Tree relation $T(id, lbl, lft, rgt)$
- Implementation:
  - sort by attribute *lft*
  - result is preorder traversal
- Efficiency:
  - single query with sort on integer attribute
  - very efficient (especially with index on *lft*)

Improving Insert/Delete Performance: Sparse Numbering

- Interval Encoding with sparse numbering:
  - leave numbers free for future insert
  - avoids global reordering until gaps are filled
  - node deletions re-open gaps
- Example:



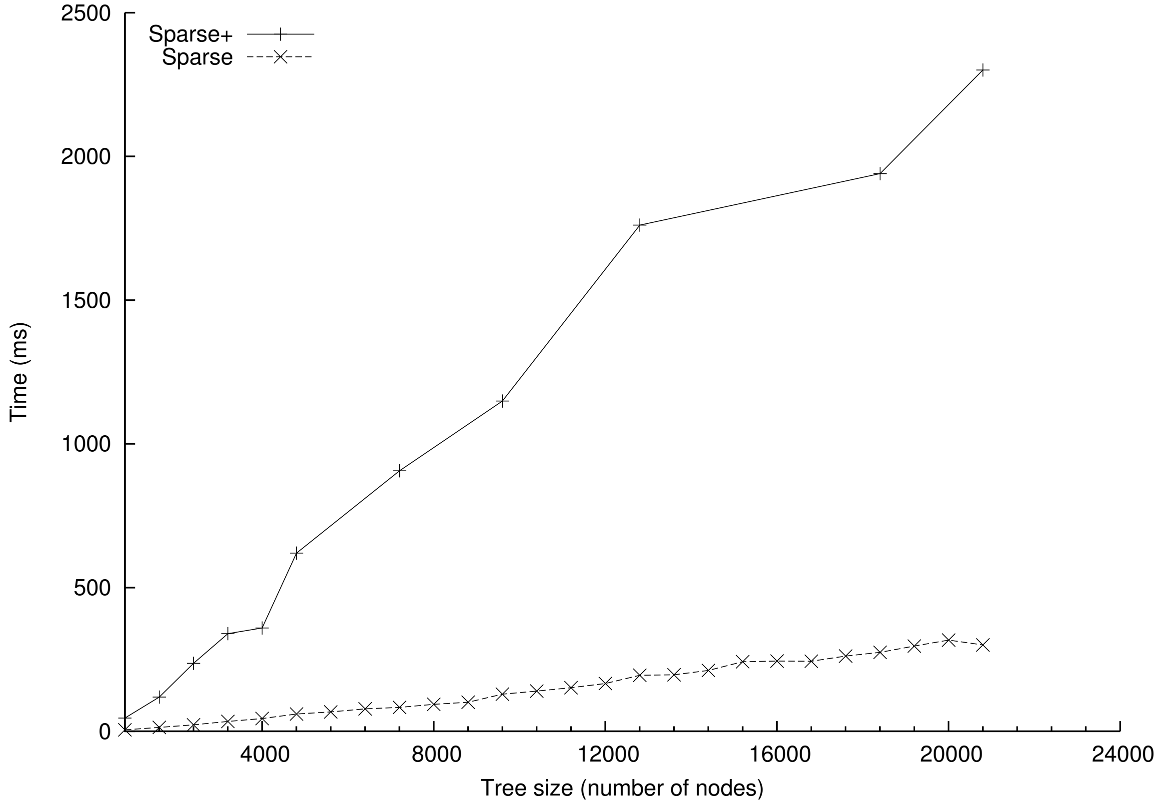
- Note: Floating-point values do not solve the problem!
- Sparse numbering using $(order, size)$-pairs [LM01]:
  - store node position as $(order, size)$-pair
  - $order$ corresponds to left interval value
  - $order + size$ corresponds to right interval value
- Example:



Node Insertion: How To Deal with Full Gaps?

- 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"

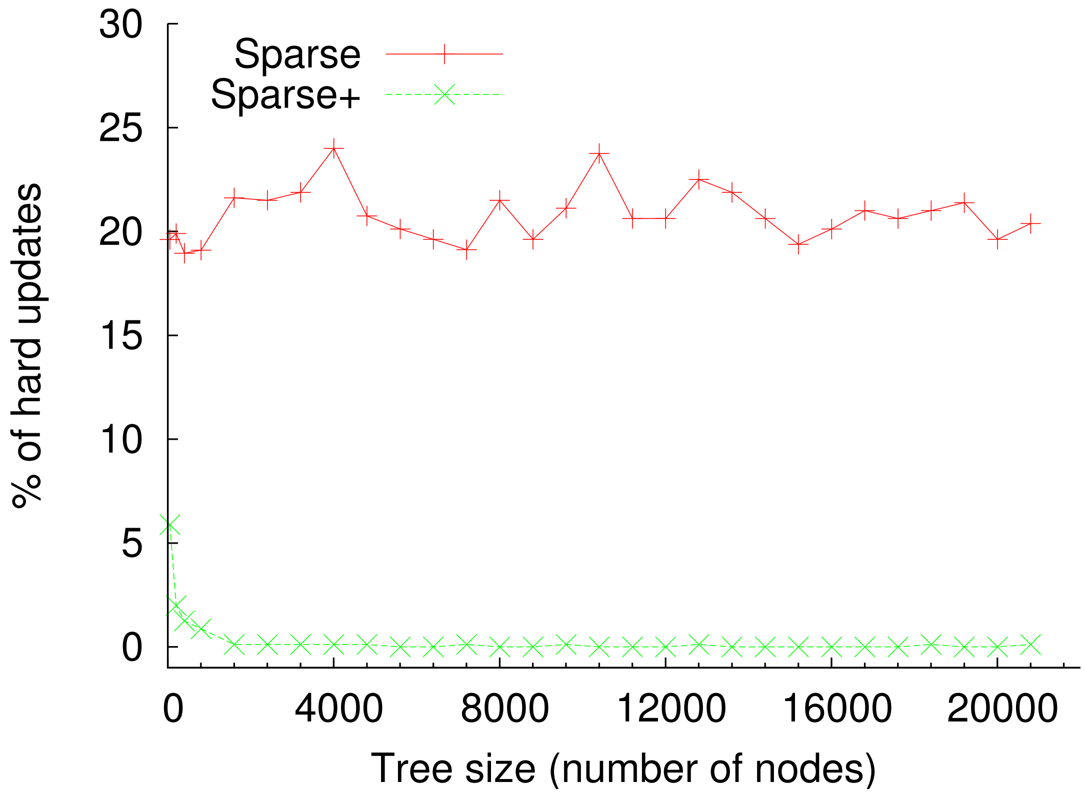
Shifting Gaps is Cheaper than Resetting All Gaps



(Graph from [Dag08])

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

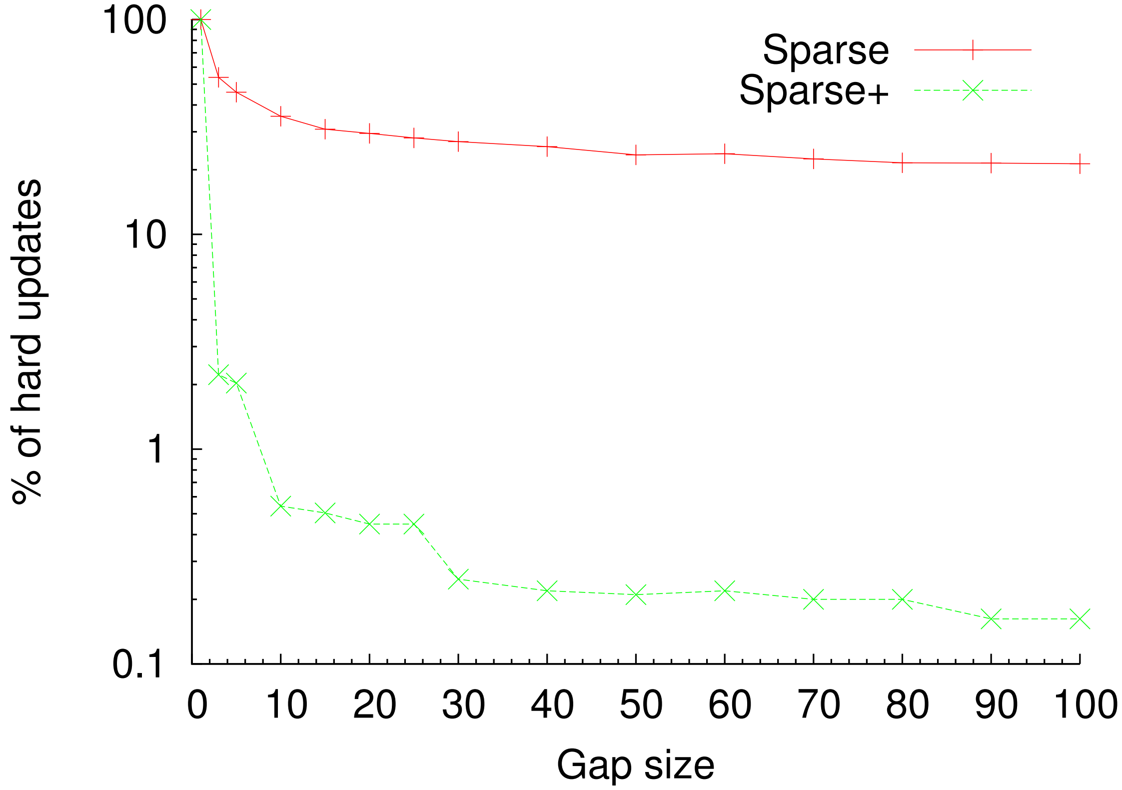
How Often Do We Need a Hard Update



(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

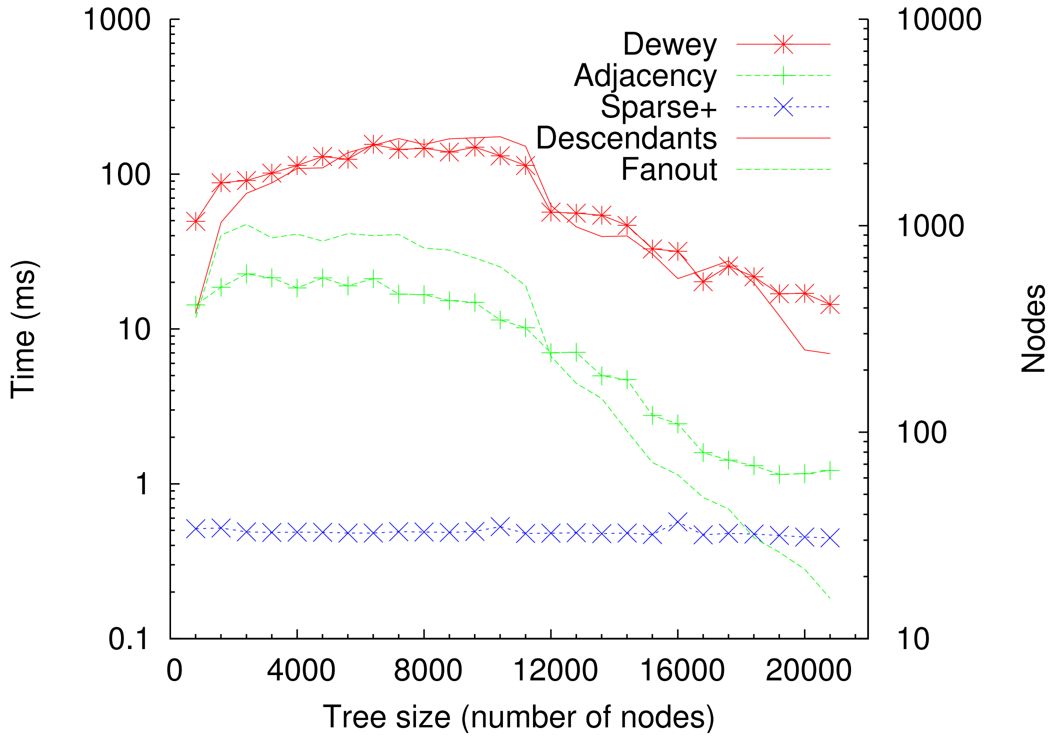


(Graph from [Dag08])

- Impact of the gap size on the number of hard updates:
  - "Sparse+": resets all gaps
  - "Sparse": shifts gaps

Outline

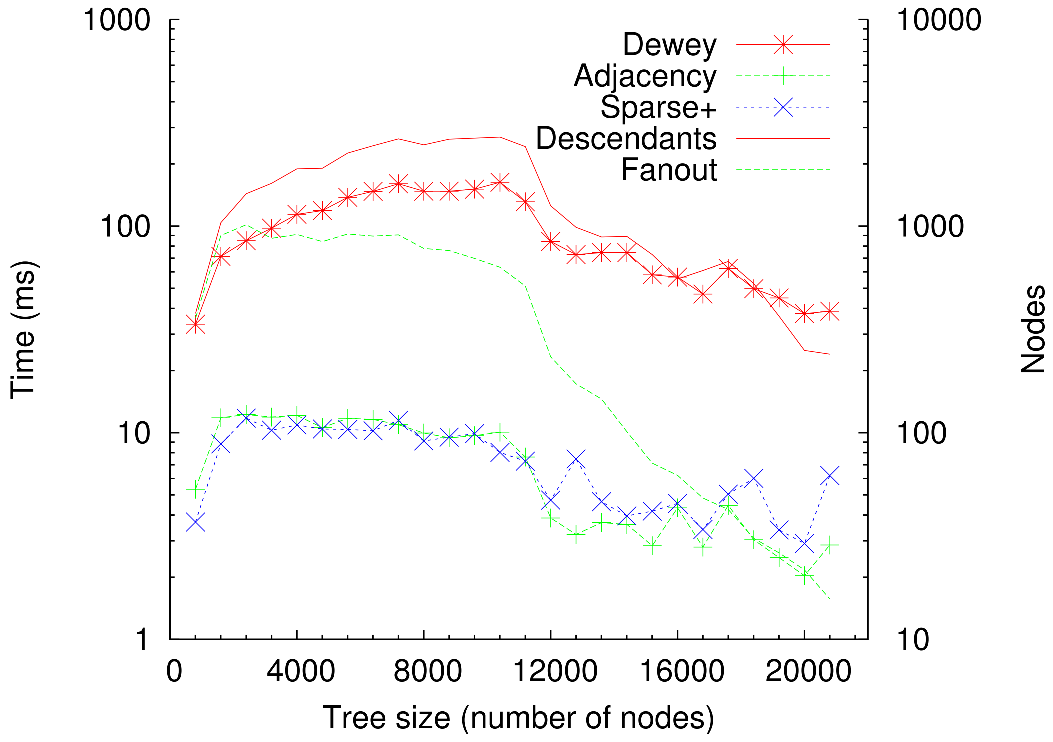
Delete Performance



(Graph from [Dag08])

- 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

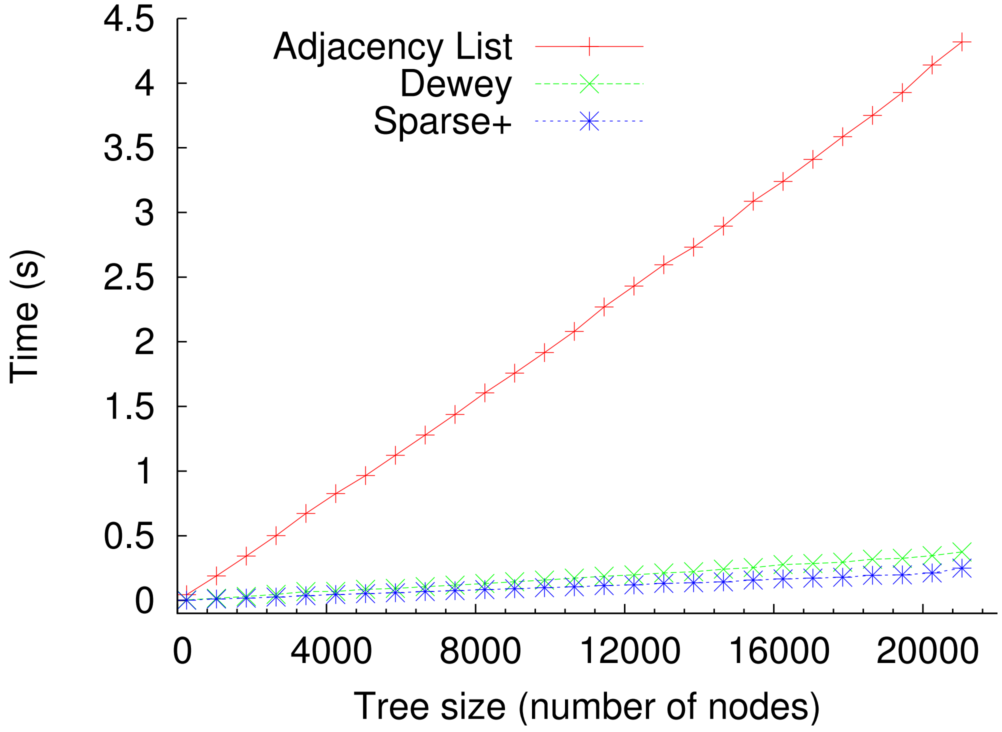
Insert Performance



(Graph from [Dag08])

- 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

Efficiency of the Preorder Traversal



(Graph from [Dag08])

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

Comparing the Encodings

	Adjacency	Dewey	Interval
+	• update very efficient • simple implementation	• preorder efficient • update efficiency: between others	• preorder very efficient • simple implementation
−	• preorder very inefficient	• update worst case is $O(n)$ • space overhead for storing paths	• insert is $O(n)$ on average (patch: sparse numbering)

Outline

Representing XML as a Tree

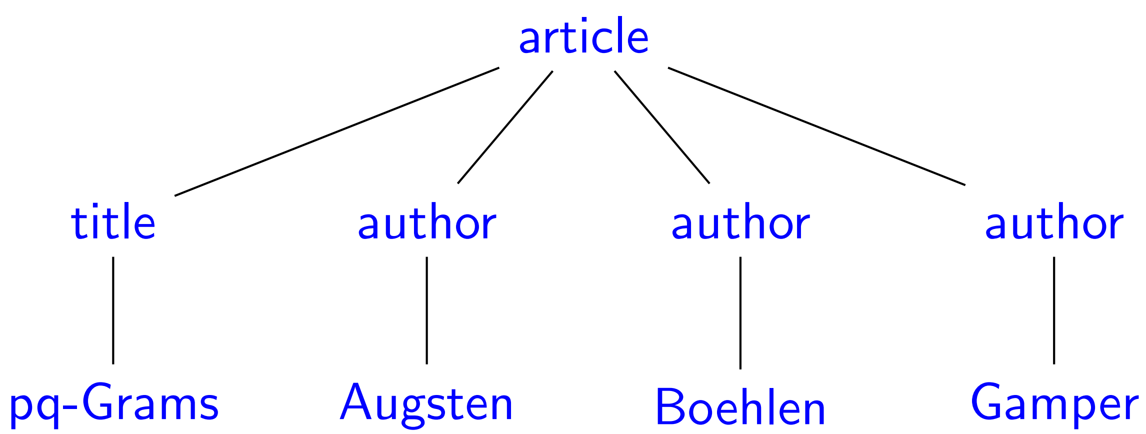
- Many possibilities – we will consider
  - single-label tree
  - double-label tree
- Pros/cons depend on application!

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>
```

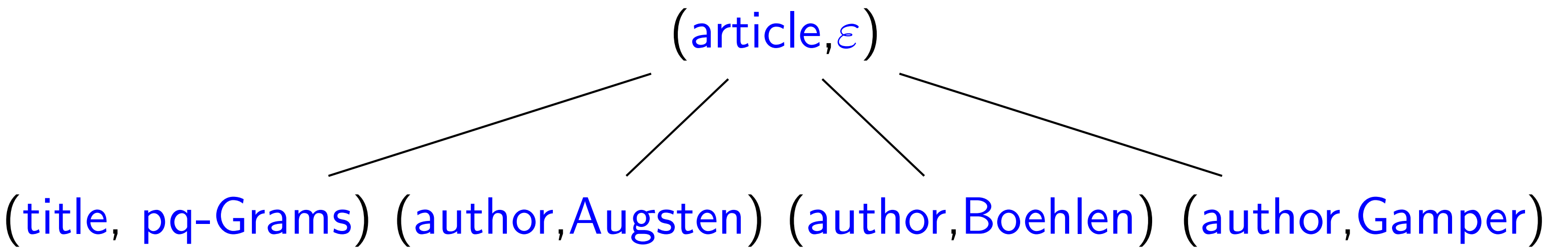


XML as a Double-Label Tree

- 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

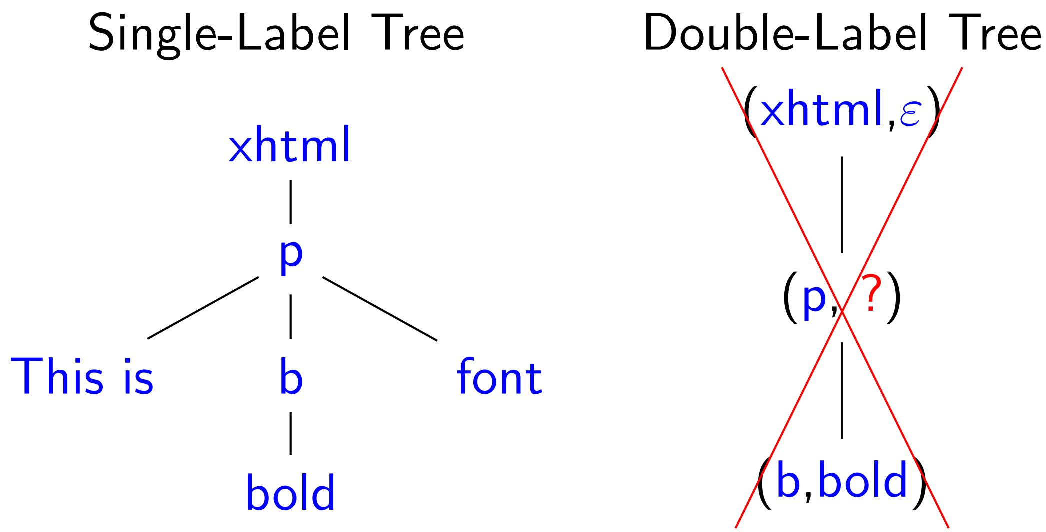
Example: XML as a Double-Label Tree

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



Example: Single- vs. Double-Label Tree

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



Parsing XML

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`

SAX – Simple API for XML

- "de facto" standard for parsing XML[3]
- 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!

[3] `http://www.saxproject.org`

Nikolaus Augsten, Michael Böhlen, and Johann Gamper.
Approximate matching of hierarchical data using *pq*-grams.
In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 301–312, Trondheim, Norway, September 2005. ACM Press.

Joe Celko.
Trees and Hierarchies in SQL for Smarties.
Morgan Kaufmann Publishers Inc., 2004.

Eigminas Dagys.
Storing XML using interval encoding with sparse numbering.
Master thesis, Free University of Bozen-Bolzano, March 2008.

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

In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 623–634, San Diego, California, June 2003. ACM Press.

Quanzhong Li and Bongki Moon.
Indexing and querying XML data for regular path expressions.
In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 361–370, Roma, Italy, September 2001. Morgan Kaufmann Publishers Inc.

Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang.
Storing and querying ordered XML using a relational database system.
In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204–215, Madison, Wisconsin, June 2002. ACM Press.