Part XV

Updating XML Documents

Outline of this part

1. Updating XML Trees
   - Update Specification
   - XQuery Update
2. Impact on XPath Accelerator Encoding
3. Impacts on Other Encoding Schemes

Updating XML trees

Throughout the course, up to now, we have **not** been looking into **updates** to XML documents at all.

- If we want to discuss *efficiency/performance* issues w.r.t. mappings of XML documents to databases, though, we need to take **modifications** into account as well as pure retrieval operations.
- As always during *physical database design*, there is a trade-off between accelerated retrieval and update performance.
- The following examples are formulated in **XQuery Update**[44], an extension to XQuery that currently has W3C draft status.

[44] `http://www.w3.org/TR/xquery-update-10/`

Updates and tree structures

During our discussion of XQuery, we have seen that *tree construction* has been a major concern. Updates, however, *cannot* be expressed with XQuery.

- Yet, we need to be able to specify *modifications* of existing XML documents/fragments as well.
- We certainly need to be able to express:
  - modification of all aspects (name, attributes, attribute values, text contents) of XML nodes, and
  - modifications of the tree structure (add/delete/move nodes or subtrees).
- As in the case of SQL, target node(s) of such modifications should be identified by means of queries.

XQuery Update: Identify, then modify

XQuery Update statements

```
insert node n
  after [before] p (: insert n as following sibling :)

insert node n
  as last [first] into p (: insert n as rightmost child of p :)

delete nodes p

replace node p
  with n

replace value of node p
  with v (: creates child text node if p is an element node :)
```

1. Given a context node, evaluate XPath expression p to **identify** target XML element node(s).
2. Node n has new identity. Last clause **preserves** p's identity.

XQuery Update: Text node updates

Obviously, the kind of c determines the overall impact on the updated tree and its encoding.

XUpdate: replacing text by text

```
<a>
  <b id="0">foo</b>
  <b id="1">bar</b>
</a>
```

```
replace value of node //b[@id = 1]
  with "foo"
```

```
<a>
  <b id="0">foo</b>
  <b id="1">foo</b>
</a>
```

- New content v: a **text node**.

XQuery Update: Text node updates

Translated into, *e.g.*, the XPath Accelerator representation, we see that

- Replacing text nodes by text nodes has **local impact** only on the *pre*/*post* encoding of the updated tree.

XUpdate statement leads to local relational update

pre	*post*	···	*text*
0	4		NULL
1	1		NULL
2	0		foo
3	3		NULL
4	2		bar

$\Rightarrow$

pre	*post*	···	*text*
0	4		NULL
1	1		NULL
2	0		foo
3	3		NULL
4	2		foo

- Similar observations can be made for updates on comment and processing instruction nodes.

XQuery Update: Structural updates

XUpdate: inserting a new subtree

```
<a>
  <b><c><d/><e/></c></b>
  <f><g/>
     <h><i/><j/></h>
  </f>
</a>
```

⇓

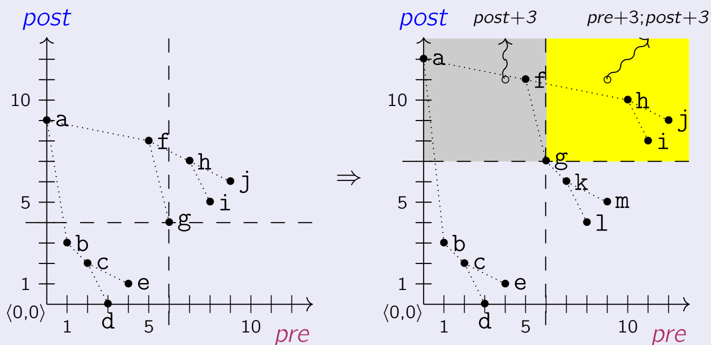
```
insert node <k><l/><m/></k>
  as first into /a/f/g
```

```
<a>
  <b><c><d/><e/></c></b>
  <f><g><k><l/><m/></k></g>
     <h><i/><j/></h>
  </f>
</a>
```

Question: What are the effects w.r.t. our structure encoding. . . ?

XQuery Update: Global impact on encoding

Global shifts in the *pre*/*post* Plane



XQuery Update: Global impact on *pre*/*post* plane

Insert a subtree of n nodes below parent element v

1. $post(v) \leftarrow post(v) + n$
2. $\forall v' \in v/\texttt{following::node()}$:
   $pre(v') \leftarrow pre(v') + n; post(v') \leftarrow post(v') + n$
3. $\forall v' \in v/\texttt{ancestor::node()}$:
   $post(v') \leftarrow post(v') + n$

Cost (tree of N nodes)

$$\underbrace{O(N)}_{②} + \underbrace{O(\log N)}_{③}$$

Update cost

③ is not so much a problem of cost but of **locking.** Why?

Updates and fixed-width encodings

Theoretical result [Milo *et.al.*, PODS 2002]

There is a sequence of updates (subtree insertions) for any persistent[45] tree encoding scheme $\mathcal{E}$, such that $\mathcal{E}$ **needs labels of length** $\Omega(N)$ to encode the resulting tree of N nodes.

- **Fixed-width** tree encodings (like XPath Accelerator) are inherently **static.**

  $\Rightarrow$ **Non**-solutions:

  - **Gaps** in the encoding,
  - encodings based on **decimal fractions**.

[45] A node keeps its initial encoding label even if its tree is updated.

A variable-width tree encoding: ORDPATH

Here we look at a particular variant of a hierarchical numbering scheme, optimized for updates.

- The **ORDPATH** encoding (used in MS SQL Server™) assigns node labels of **variable length.**

ORDPATH labels for an XML fragment

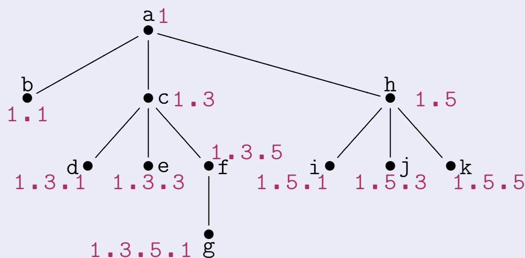
1. The fragment root receives label 1.
2. The nth ($n = 1, 2, \dots$) child of a parent node labelled p receives label $p\,\textbf{.}\,(2 \cdot n - 1)$.

- Internally, ORDPATH labels are not stored as **.**-separated ordinals but using a prefix-encoding (similarities with Unicode).

ORDPATH encoding: Example

ORDPATH encoding of a sample XML fragment

```
<a>
  <b/>
  <c>
    <d/><e/>
    <f><g/></f>
  </c>
  <h>
    <i/><j/><k/>
  </h>
</a>
```



Note:

- **Lexicographic** order of ORDPATH labels ≡ document order

⇒ **Clustered index on ORDPATH labels** will be helpful.

ORDPATH: Insertion between siblings

In ORDPATH, the **insertion of new nodes** between two existing sibling nodes is referred to as "*careting in*" (caret $\widehat{=}$ insertion mark, $\curlywedge$).

ORDPATH: node insertion

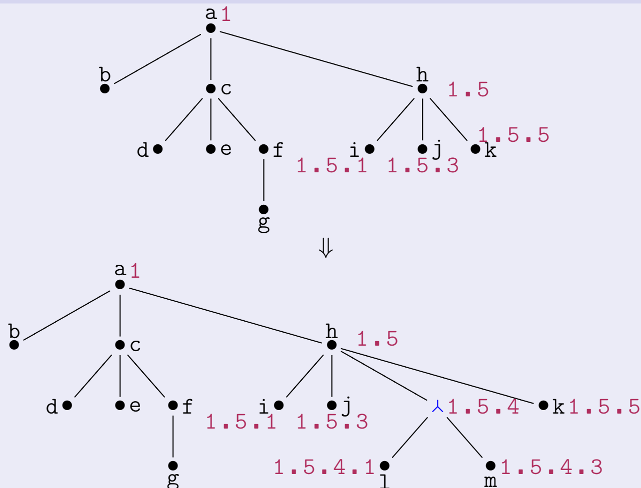
Let $(v_1, \ldots, v_n)$ denote a sequence of nodes to be inserted between two existing sibling nodes with labels $p \centerdot s$ and $p \centerdot (s+2)$, s odd. After insertion, the new label of v_i is

$$p \centerdot (s+1) \centerdot (2 \cdot i - 1) \ .$$

Label $p \centerdot (s+1)$ is referred to as a **caret.**

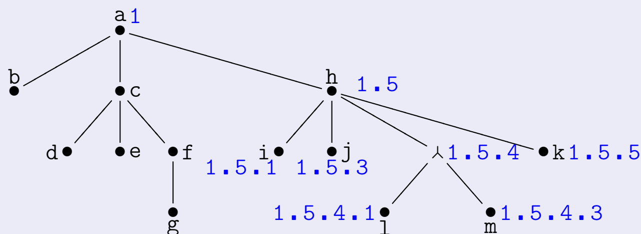
ORDPATH: Insertion between siblings (Example)

Insertion of (<l/>, <m/>) between <j/> and <k/>



ORDPATH: Insertion between siblings

ORDPATH: Insertions at arbitrary locations?



Determine ORDPATH label of new node v inserted

1. to the right of <k/>,
2. to the left of <i/>,
3. between <j/> and <l/>,
4. between <l/> and <m/>,

Processing XQuery and ORDPATH

Is ORDPATH a suitable encoding $\mathcal{E}$?

Mapping core operations of the XQuery processing model to operations on ORDPATH labels:

v`/parent::node()`

1. Let $p\,.\,m\,.\,n$ denote v's label (n is odd).
2. If the rightmost ordinal (m) is even, remove it. Goto ②.

In other words: the carets (⅄) do not count for ancestry.

v`/descendant::node()`

1. Let $p\,.\,n$ denote v's label (n is odd).
2. Perform a lexicographic index range scan from $p\,.\,n$ to $p\,.\,(n+1)$—the *virtual following sibling* of v.

ORDPATH: Variable-length node encoding

- Using (4 byte) integers for all numbers in the hierarchical numbering scheme is an obvious waste of space!
- Fewer (and variable number of) bits are typically sufficient;
- they may bear the risk of running out of new numbers, though. In that case, even ORDPATH cannot avoid *renumbering*.
  - In principle, though, *no bounded* representation can absolutely avoid the need for renumbering.
- Several approaches have been proposed so as to alleviate the problem, for instance:
  - use a variable number of bits/bytes, akin to Unicode,
  - apply some (order-preserving) hashing schemes to shorten the numbers,
  - ...

ORDPATH: Variable-length node encoding

- For a 10 MB XML sample document, the authors of ORDPATH observed label lenghts between 6 and 12 bytes (using Unicode-like compact representations).
- Since ORDPATH labels encode **root-to-node** paths, node labels share **common prefixes.**

ORDPATH labels of <l/> and <m/>

```
1.5.4.1
1.5.4.3
```

$\Rightarrow$ Label comparisons often need to inspect encoding bits at the far right.

- MS SQL Server™ employs further path encodings organized in **reverse** (node-to-root) order.
- **Note:** Fixed-length node IDs (such as, *e.g.*, preorder ranks) typically fit into CPU registers.