

# 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**<sup>44</sup>, an extension to XQuery that currently has W3C draft status.

---

<sup>44</sup><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

<i>pre</i>	<i>post</i>	...	<i>text</i>
0	4		NULL
1	1		NULL
2	0		foo
3	3		NULL
4	2		bar

⇒

<i>pre</i>	<i>post</i>	...	<i>text</i>
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>

```

↓

```

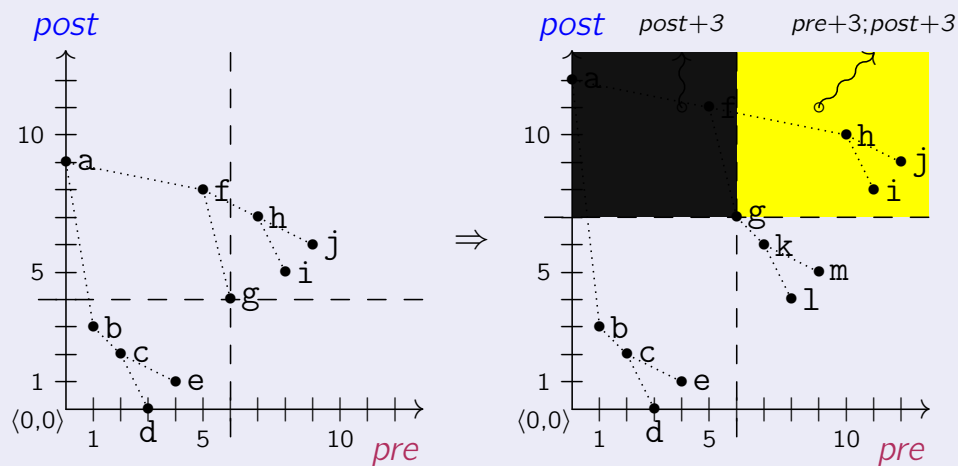
<a>
  <b><c><d/><e/></c></b>
  <f><g><k><l/><m/></k></g>
    <h><i/><j/></h>
</f>
</a>

```

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

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

## XQuery Update: Global impact on encoding

Global shifts in the *pre/post* PlaneXQuery Update: Global impact on *pre/post* planeInsert a subtree of  $n$  nodes below parent element  $v$ 

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

Cost (tree of  $N$  nodes)

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

## 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<sup>45</sup> 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**.
  - ⇒ **Non-solutions**:
    - **Gaps** in the encoding,
    - encodings based on **decimal fractions**.

<sup>45</sup>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<sup>TM</sup>) assigns node labels of **variable length**.

### ORDPATH labels for an XML fragment

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

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

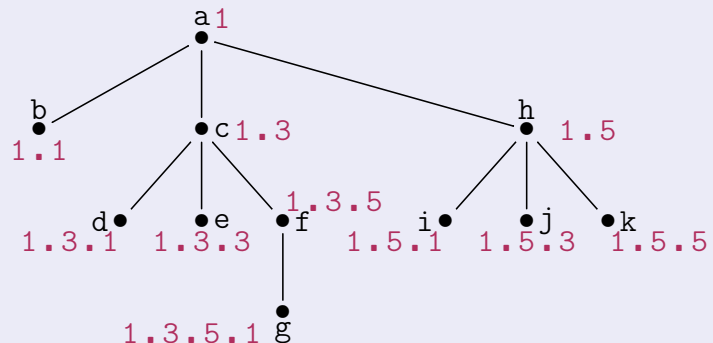
## ORDPATH encoding: Example

### 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  $\equiv$  document order
- $\Rightarrow$  **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 “*caretting in*” (caret  $\hat{=}$  insertion mark,  $\wedge$ ).

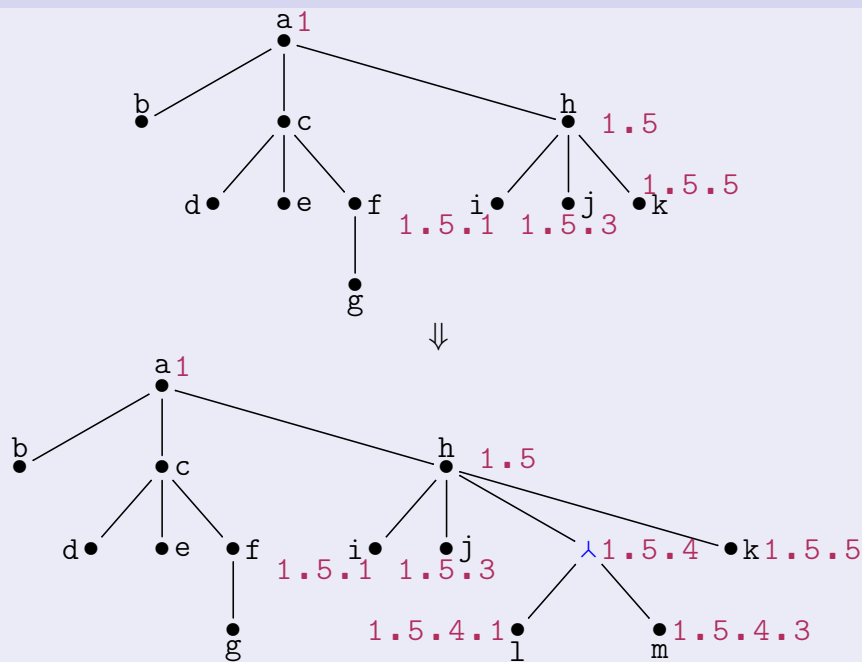
### ORDPATH: node insertion

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

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

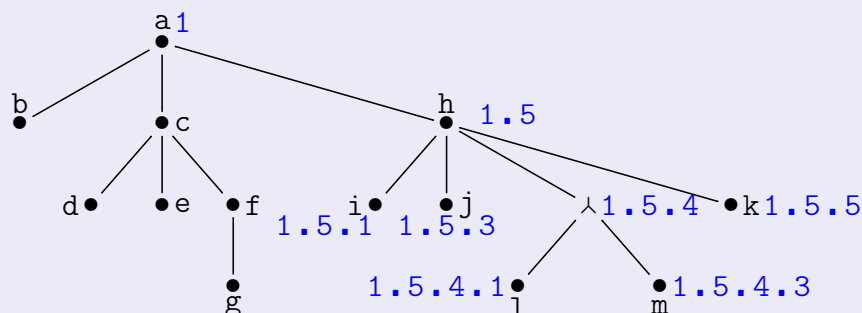
Label  $p.(s+1)$  is referred to as a **caret**.

## ORDPATH: Insertion between siblings (Example)

Insertion of  $\langle l/\rangle$ ,  $\langle m/\rangle$  between  $\langle j/\rangle$  and  $\langle k/\rangle$ 

## ORDPATH: Insertion between siblings

ORDPATH: Insertions at arbitrary locations?

Determine ORDPATH label of new node  $v$  inserted

- ① to the right of  $\langle k/\rangle$ ,
- ② to the left of  $\langle i/\rangle$ ,
- ③ between  $\langle j/\rangle$  and  $\langle l/\rangle$ ,
- ④ between  $\langle l/\rangle$  and  $\langle m/\rangle$ ,



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

In other words: the carets ( $\wedge$ ) do not count for ancestry.

`v/descendant::node()`

- ① Let  $p \cdot n$  denote  $v$ 's label ( $n$  is odd).
- ② Perform a lexicographic index range scan from  $p \cdot n$  to  $p \cdot (n + 1)$ —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 lengths 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

⇒ 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.