

Part XIV

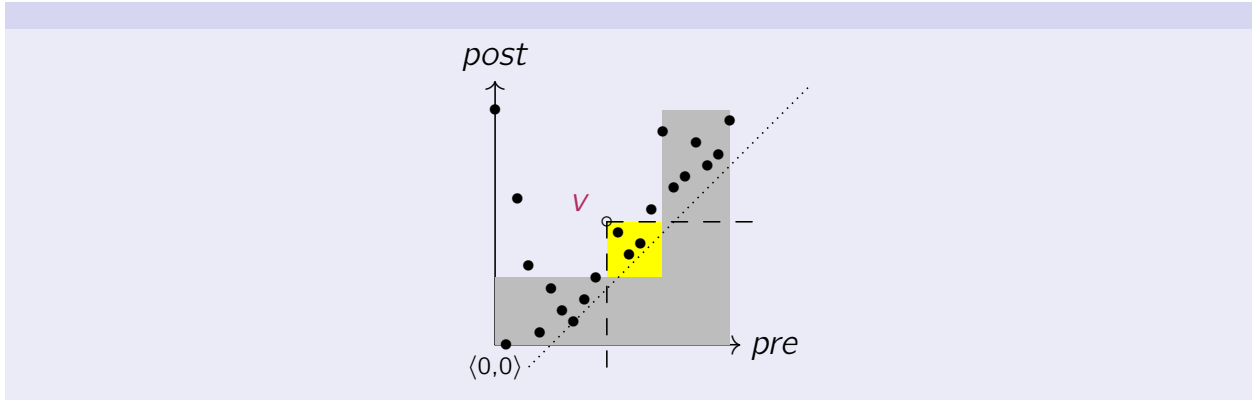
XPath Accelerator Optimization

Outline of this part

- 1 Scan Ranges
 - descendant Axis
- 2 Stretched *Pre/Post* Plane
- 3 XPath Symmetries

Scan ranges: descendant axis

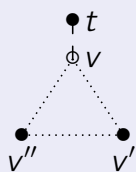
Consider a descendant step originating in context node v :



A significant fraction of the i_{pre} and i_{post} B-tree index scan is guaranteed to deliver **false hits** only.

Shrink-wrapping the descendant window

Subtree below v



- v'' has min. postorder rank below v
- v' has max. preorder rank below v
- $pre(v') = pre(v) + size(v)$
- $post(v'') = post(v) - size(v)$
- Sufficient to scan B-tree in the $(pre(v), pre(v'))$ range

- $size(v) = |v/descendant :: node()|$



If we can derive (a reasonable estimate for) $size(v)$ from $pre(v)$ and $post(v)$, we can **shrink** the descendant window.

Shrink-wrapping the descendant window

An alternative characterization of preorder/postorder ranks

$$\begin{aligned}
 pre(v) &= |v/\text{preceding}::\text{node}()| + \overbrace{|v/\text{ancestor}::\text{node}()|}^{= \text{level}(v)} + 1 \\
 post(v) &= |v/\text{preceding}::\text{node}()| + \overbrace{|v/\text{descendant}::\text{node}()|}^{= \text{size}(v)} + 1
 \end{aligned}$$

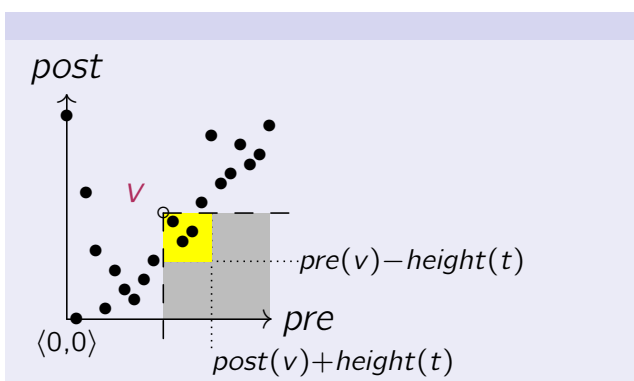
↓

$$\begin{aligned}
 post(v) - pre(v) &= \text{size}(v) - \underbrace{\text{level}(v)} \\
 &\leq \text{height}(t)
 \end{aligned}$$

Estimate the location of v' and v'' in the $pre/post$ plane

$$pre(v') \leq post(v) + \text{height}(t) \quad | \quad post(v'') \geq pre(v) - \text{height}(t)$$

Shrink-wrapping the descendant window



- Size of B-tree scan region now **dependent on actual subtree size** below v (and independent of fragment t 's size!).
- Scan region size estimate maximally off by $\text{height}(t)$.

Overestimation of descendant window size

How significant would you judge this estimation error? How to avoid the error at all?

Stretched *pre/post* plane

- While **index intersection** (IXAND) and **window shrinking** go a long way in making location step evaluation efficient in the *pre/post* plane, windows are still evaluated in a two-step process, leading to false hits.
 - A different way to approach this problem is to employ **concatenated** $\langle pre, post \rangle$ **B-trees**.
- Here, instead we will exploit the observation that predicate $window(\cdot)$ solely depends on comparisons ($<$, $>$) on *pre* and *post*. The **absolute *pre/post* values are immaterial**.

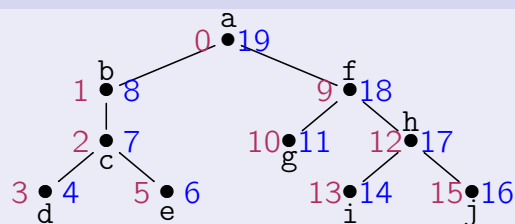
Stretched *pre/post* plane

“Stretched” (or coupled) *preorder/postorder* ranks

Perform a depth-first, left-to-right traversal of the skeleton tree. Maintain counter *rank* (initially 0).

- 1 Whenever a node v is visited first, assign $pre(v) \leftarrow rank$; increment $rank$.
- 2 When v is visited last, assign $post(v) \leftarrow rank$; increment $rank$.

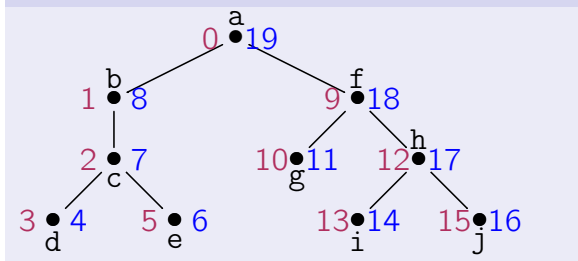
Example



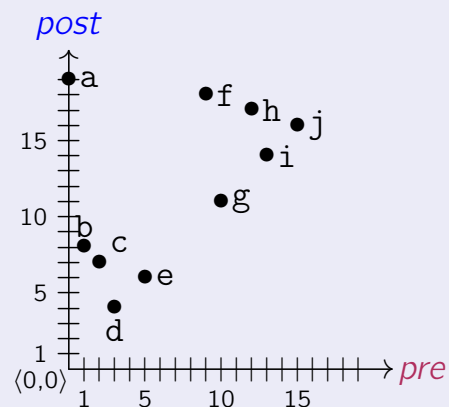
This encoding is also known as “start–end” numbering.

Stretched *pre/post* plane

start-end numbering



Stretched *pre/post* plane



Node identifiers of bit width n encode 2^{n-1} nodes.

XPath axes in the stretched *pre/post* plane

Node distribution in the stretched *pre/post* plane has interesting properties:

- The axes *window*(\cdot) predicates continue to work as before.

Further:

Characterization of descendant axis

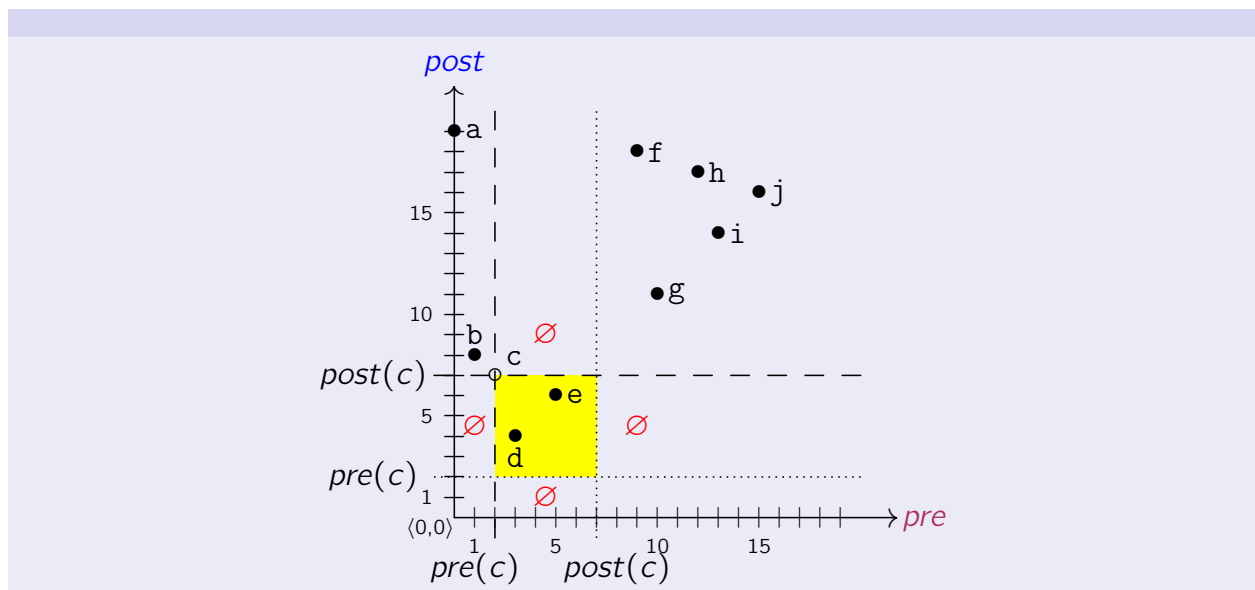
Node v is selected by $c/\text{descendant}::\text{node}()$, iff

$$\text{pre}(v) \in (\text{pre}(c), \text{post}(c)) \quad \text{or} \quad \text{post}(v) \in (\text{pre}(c), \text{post}(c))$$

Subtree size (exact, no estimation)

For any node v :

$$\text{size}(v) = 1/2 \cdot (\text{post}(v) - \text{pre}(v) - 1)$$

c/descendant::node()XPath axes in the stretched *pre/post* plane

In terms of query windows, on the stretched *pre/post* plane we may modify *window*(\cdot) as follows:

Axis descendant in the stretched plane

$$window(\text{descendant}::t, v) = \begin{cases} \langle (pre(v), post(v)), *, *, elem, t \rangle \\ \text{or} \\ \langle *, (pre(v), post(v)), *, elem, t \rangle \end{cases}$$

A **single index scan** suffices (no IXAND, no false hits).

- Axes descendant-or-self and child benefit, too.

Leaf node access

For a certain class of XPath steps, we can statically⁴³ infer that **all result nodes will be leaves** (let c denote an arbitrary XPath expression):

- $c/text()$, $c/comment()$, $c/processing-instruction()$
- $c[not(child::node())]$

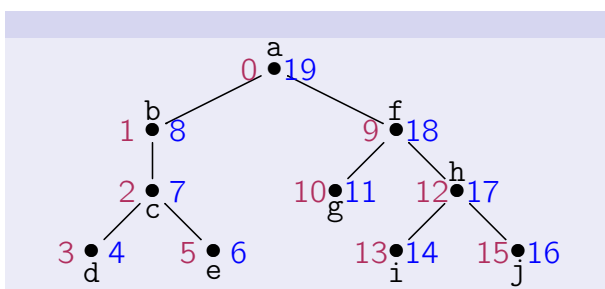
Characterization of any leaf node ℓ

A diagonal in the stretched *pre/post* plane:

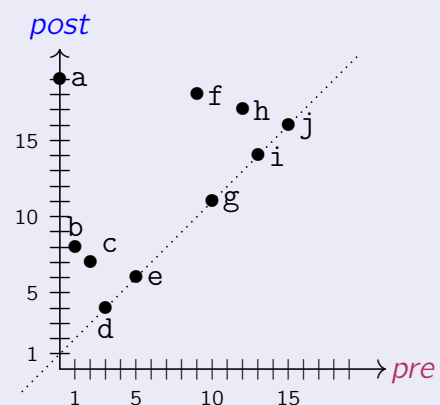
$$post(\ell) = pre(\ell) + 1$$

⁴³At query compile time.

Leaves diagonal



$$post(\ell) = pre(\ell) + 1$$



“Backwards” step processing

Presence of the leaves diagonal enables the RDBMS to evaluate certain XPath expressions in a “backwards” fashion.

Exploit symmetries in XPath

Consider the query

$$\text{descendant::}t/\text{child::text}()$$

We can instead process the **equivalent symmetric** query

$$\underbrace{\text{descendant::text}()}_{\text{found on leaves diagonal}}[\text{parent::}t]$$

NB. The latter query does *not* require window evaluation at all.

Exploiting schema/DTD information

The presence of a **DTD** (or XML Schema description) for a pre-/postorder encoded document may be used to generalize the leaves diagonal discussion.

- From a DTD we can derive **maximal/minimal subtree sizes** for any XML element node v with tag t .
- Together with

$$\begin{aligned} \text{size}(v) &= 1/2 \cdot (\text{post}(v) - \text{pre}(v) - 1) \\ &\quad \updownarrow \\ \text{post}(v) &= 2 \cdot \text{size}(v) + \text{pre}(v) + 1 \end{aligned}$$

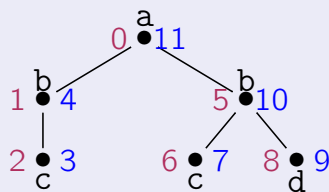
we can establish a **stripe** in the stretched *pre/post* plane which is guaranteed to contain all elements with tag t .

Exploiting schema/DTD information

Sample DTD and encoding of a **valid** fragment

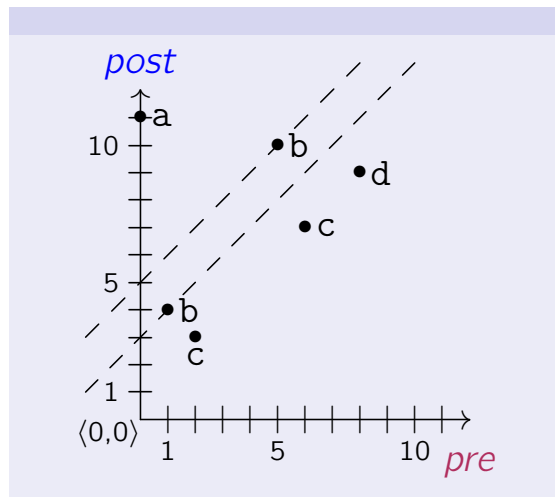
```
<!ELEMENT a (b+)>
<!ELEMENT b (c,d?)>
<!ELEMENT c EMPTY>
<!ELEMENT d EMPTY>
```

⇒ Minimum (maximum) subtree size of b elements in a valid fragment is 1 (2).



⇒ All b elements in stripe

$$3 \leq post(v) - pre(v) \leq 5$$



XPath symmetries

- Clearly, *pre/post* plane **window size** is the dominating cost factor for the XPath Accelerator.
 - The window size determines the stride of B-tree range scans and thus the amount of secondary memory touched (affects # I/O operations necessary).

(We could even try to derive a **cost model** from window size.)

- How can we benefit from this observation?

XPath symmetries

Plan choices: `/descendant::t/ancestor::s`

① **Forward mode.**

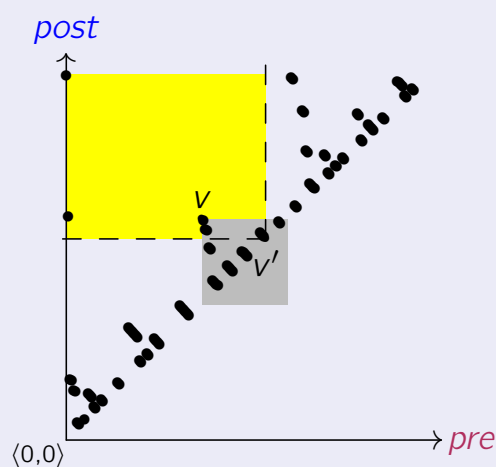
Find intermediary context node sequence of elements with tag t . Then, for each node v' in this sequence, evaluate $window(ancestor::s, v')$.

② **Backward mode.**

Find intermediary context node sequence of elements with tag s . Then, for each node v in this sequence, check whether $window(descendant::t)$ yields at least one node v' . If no such v' is found, drop v .

NB. Based on the descendant \leftrightarrow ancestor symmetry.

XPath symmetries and window size



Note: plan ① evaluates the , plan ② the $window(s)$.

XPath symmetries

- Note that plan ② corresponds to the **symmetrical equivalent** of the original location path:

XPath Symmetry

① `/descendant::t/ancestor::s`
 \updownarrow
 ② `/descendant-or-self::s[descendant::t]`

Can you suggest a proof for the symmetry?

Why is axis descendant-or-self used in ②?

- The **query rewrite** ① \rightarrow ② could also be initiated on the XQuery (XPath) source level.

More XPath symmetries

XPath Symmetries (due to Dan Olteanu, *et.al.*)

```

      :
descendant::t/parent::s  $\leftrightarrow$  descendant-or-self::s[child::t]
  child::t/parent::s  $\leftrightarrow$  self::s[child::t]
  c/child::t/ancestor::s  $\leftrightarrow$  c[child::t]/ancestor-or-self::s
/descendant::t/preceding::s  $\leftrightarrow$  /descendant::s[following::t]
      :

```
