Part XIV

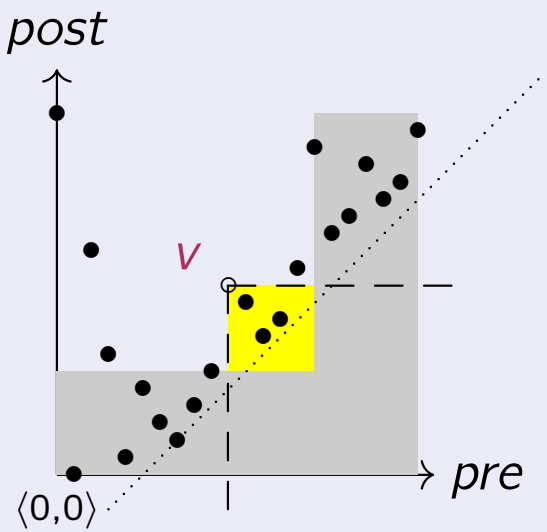
XPath Accelerator Optimization

Outline of this part

1. Scan Ranges
   - `descendant` Axis
2. Streched *Pre*/*Post* Plane
3. XPath Symmetries

Scan ranges: descendant axis

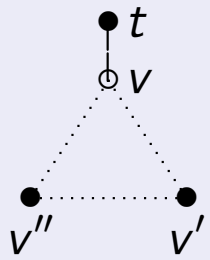
Consider a descendant step originating in context node v:



A significant fraction of the ipre and ipost 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

- $size(v) = |v/\texttt{descendant::node()}|$

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

$$pre(v) = |\, v/\texttt{preceding::node()} \,| + \overbrace{|\, v/\texttt{ancestor::node()} \,|}^{= level(v)} + 1$$
$$post(v) = |\, v/\texttt{preceding::node()} \,| + \underbrace{|\, v/\texttt{descendant::node()} \,|}_{= size(v)} + 1$$

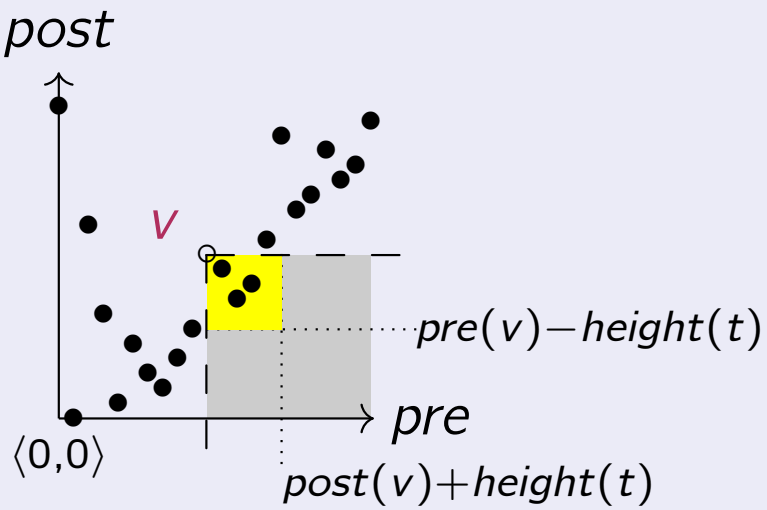
$$\Downarrow$$

$$post(v) - pre(v) = size(v) - \underbrace{level(v)}_{\leqslant\ height(t)}$$

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

$$pre(v') \leqslant post(v) + height(t) \;\Big|\; post(v'') \geqslant pre(v) - height(t)$$

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 $height(t)$.

Overestimation of descendant window size

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

Streched *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.**

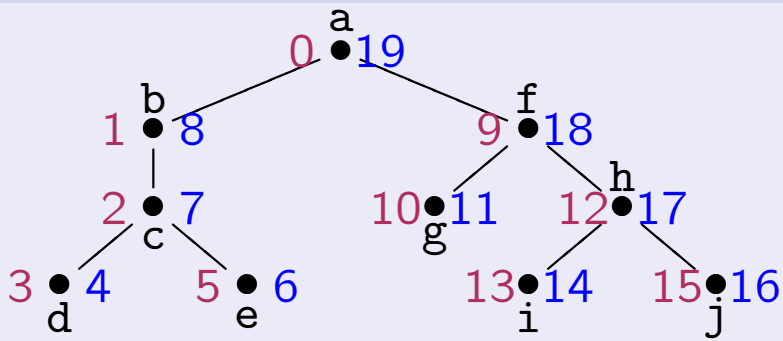
Streched *pre*/*post* plane

"Stretched" (or coupled) *preorder*/*postorder* ranks

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

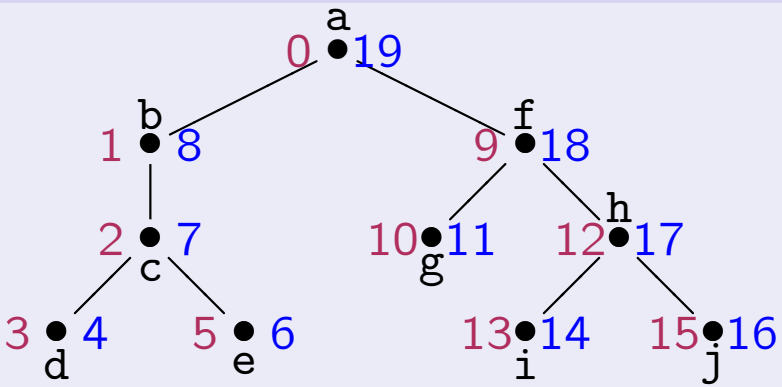
Example



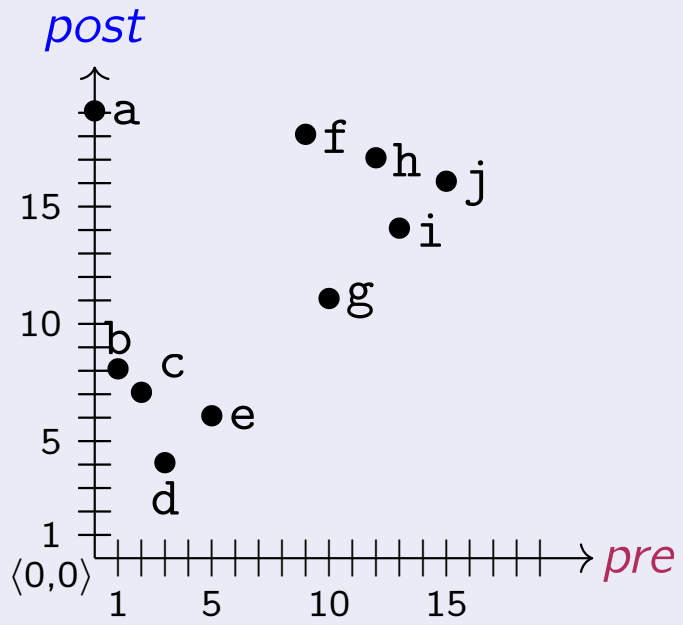
This encoding is also known as "start–end" numbering.

Streched *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 streched *pre*/*post* plane

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

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

Further:

Characterization of `descendant` axis

Node v is selected by c/`descendant::node()`, iff

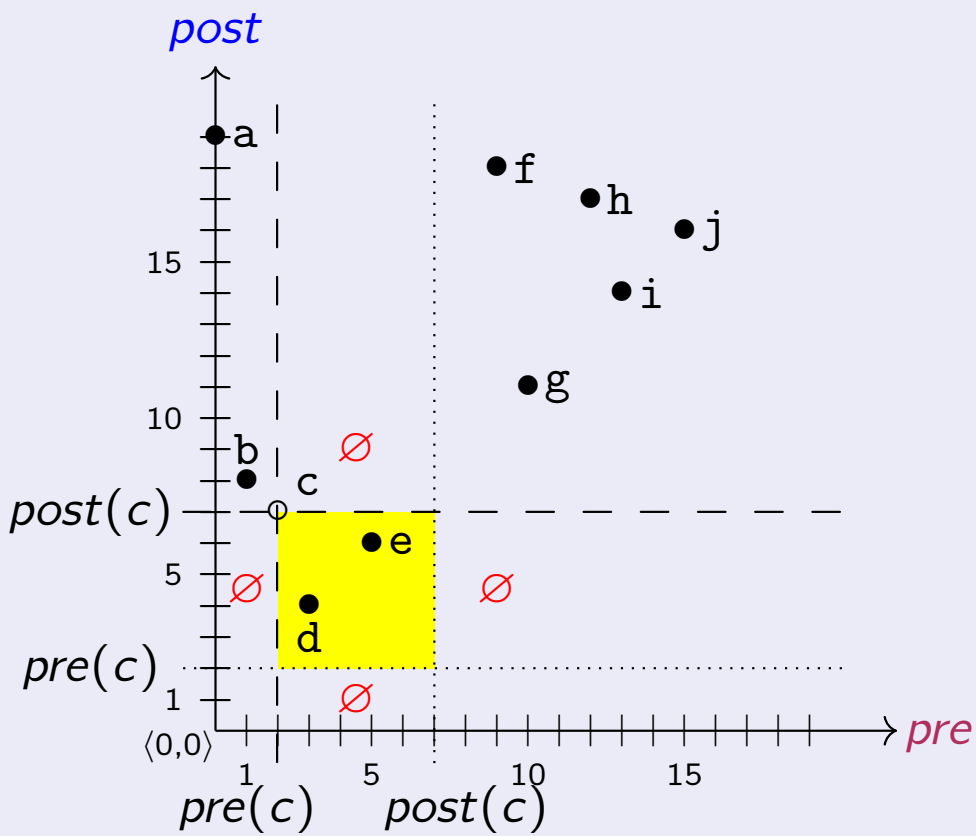
$$pre(v) \in (pre(c), post(c)) \quad \textbf{or} \quad post(v) \in (pre(c), post(c))$$

Subtree size (exact, no estimation)

For any node v:

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

c/descendant::node()



XPath axes in the streched *pre*/*post* plane

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

Axis `descendant` in the stretched plane

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

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 statically[44] 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())]`

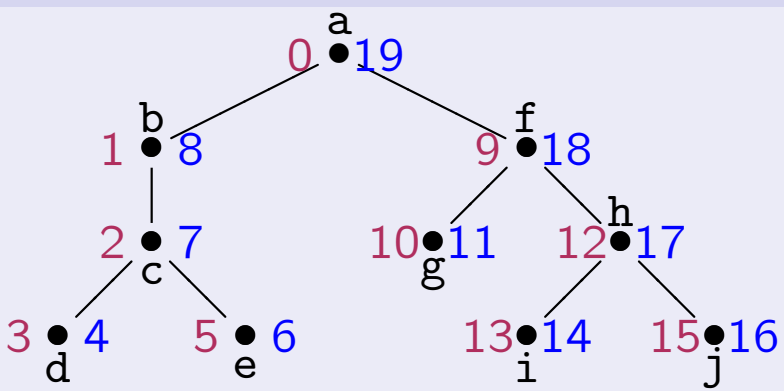
Characterization of any leaf node ℓ

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

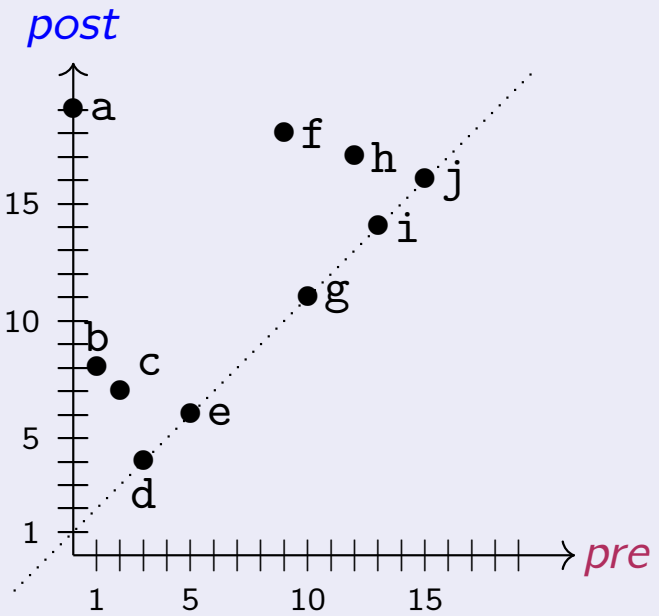
$$post(\ell) = pre(\ell) + 1$$

[44] At 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

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

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

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

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

$$size(v) = 1/2 \cdot (post(v) - pre(v) - 1)$$
$$\Updownarrow$$
$$post(v) = 2 \cdot size(v) + pre(v) + 1$$

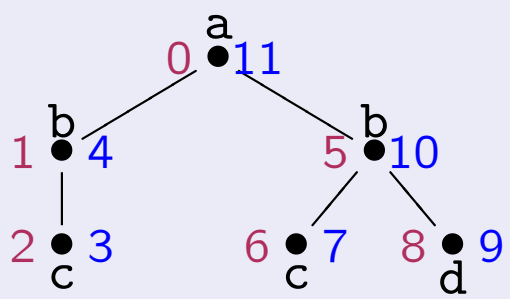
we can establish a **stripe** in the stretched *pre*/*post* plane which is guaranteed to contain all elements with tag t.

Exploiting schema/DTD information

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

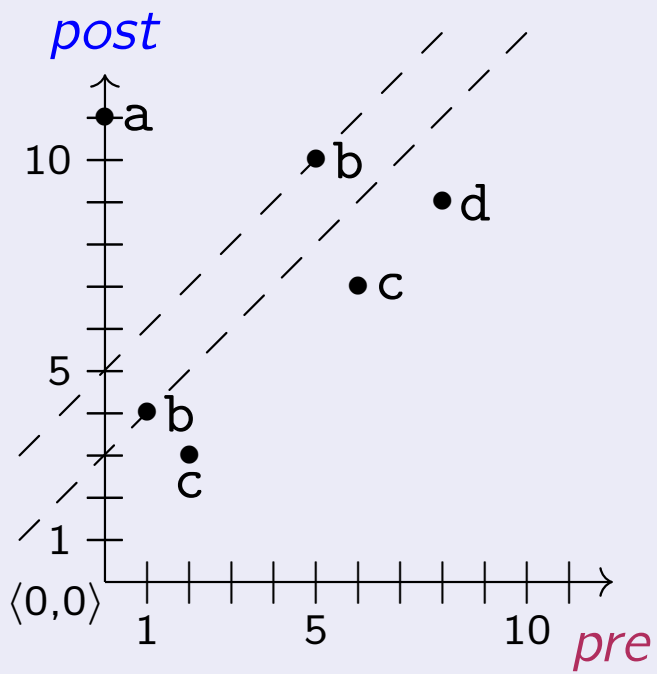
```
<!ELEMENT a (b+)>
<!ELEMENT b (c,d?)>
<!ELEMENT c EMPTY>
<!ELEMENT d EMPTY>
```

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



⇒ All `b` elements in stripe

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



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?

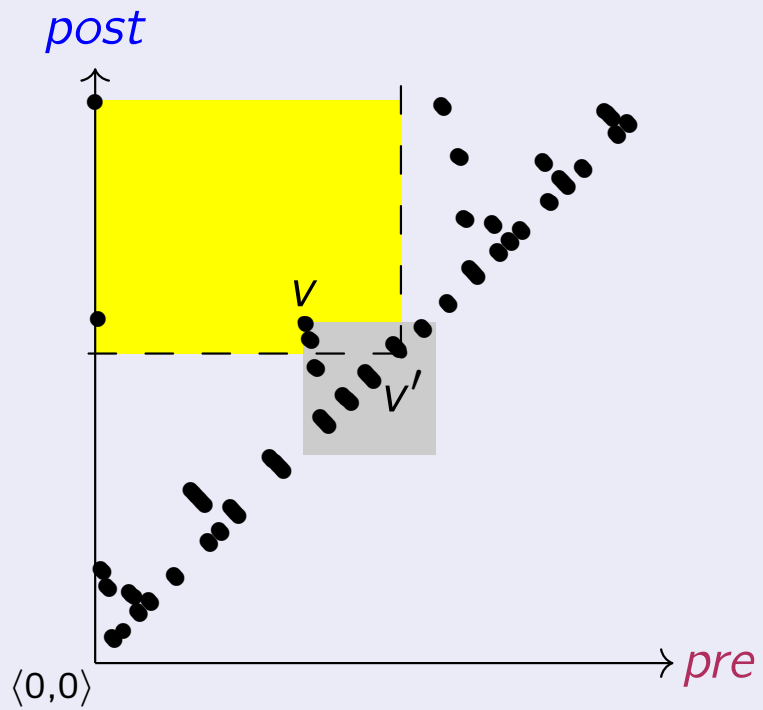
XPath symmetries

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

↕

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 → ② could also be initiated on the XQuery (XPath) source level.

More XPath symmetries

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

$$
\begin{array}{r}
\vdots \\
\texttt{descendant::}t\texttt{/parent::}s \leftrightarrow \texttt{descendant-or-self::}s\texttt{[child::}t\texttt{]} \\
\texttt{child::}t\texttt{/parent::}s \leftrightarrow \texttt{self::}s\texttt{[child::}t\texttt{]} \\
c\texttt{/child::}t\texttt{/ancestor::}s \leftrightarrow c\texttt{[child::}t\texttt{]/ancestor-or-self::}s \\
\texttt{/descendant::}t\texttt{/preceding::}s \leftrightarrow \texttt{/descendant::}s\texttt{[following::}t\texttt{]} \\
\vdots
\end{array}
$$