

Part XVII

Staircase Join—Tree-Aware Relational (X)Query Processing

Outline of this part

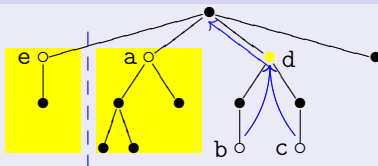
- 1 XPath Accelerator—Tree aware relational XML representation
 - Enhancing Tree Awareness
- 2 Staircase Join
 - Tree Awareness
 - Context Sequence Pruning
 - Staircases
- 3 Injecting \sqcup into PostgreSQL
- 4 Outlook: More on Performance Tuning in MonetDB/XQuery

Enhancing tree awareness

- We now know that the XPath Accelerator is a true **isomorphism** with respect to the XML skeleton **tree structure**.
 - Witnessed by our discussion of **shredder** (\mathcal{E}) and **serializer** (\mathcal{E}^{-1}).
- We will now see how the database kernel can benefit from a more elaborate **tree awareness** (beyond document order and semantics of the four major XPath axes).
- This will lead to the design of **staircase join** \sqcup , the core of MonetDB/XQuery's XPath engine.
 - We will also discuss issues of how to tune \sqcup to get the most out of modern CPUs and memory architectures.

Tree awareness?

Document order and XPath semantics aside, what are further **tree properties** of value to a relational XML processor?



- ① The **size of the subtree** rooted in node a is 5
- ② The leaf-to-root **paths** of nodes b, c **meet** in node d
- ③ The **subtrees** rooted in e and a are necessarily **disjoint**

Tree awareness ①: Subtree size

We have seen that tree property **subtree size** (① on previous slide) is implicitly present in a *pre/post*-based tree encoding:

$$post(v) - pre(v) = size(v) - level(v)$$

- To exploit property **subtree size**, we were able to find a means on the **SQL language level**, *i.e.*, **outside the database kernel**.
- ⇒ This led to *window shrink-wrapping* for the XPath descendant axis.

Tree awareness on the SQL level

Shrink-wrapping for the descendant axis

$$Q \equiv (c)/\text{following}::\text{node}()/\text{descendant}::\text{node}()$$

path(Q)

```
SELECT  DISTINCT v2.pre
FROM    accel v1, accel v2
WHERE   v1.pre > c.pre
        AND v1.pre < v2.pre
        AND v1.post > c.post
        AND v1.post > v2.post
        AND v2.pre <= v1.post + h AND v2.post >= v1.pre + h
ORDER BY v2.pre
```

Tree awareness ②: Meeting ancestor paths

- Evaluation of axis `ancestor` can clearly benefit from knowledge about the exact element node where several given **node-to-root paths meet**.
 - For example:
For context nodes c_1, \dots, c_n , determine their **lowest common ancestor** $v = lca(c_1, \dots, c_n)$.
 \Rightarrow Above v , produce result nodes once only.
(This still produces duplicate nodes below v .)
- This knowledge *is* present in the encoding but is **not as easily expressed on the level of commonly available relational query languages** (such as, SQL or relational algebra).

Flashback: XPath: Ensuring order is not for free

The strict XPath requirement to construct a result in document order may imply **sorting effort** depending on the actual XPath implementation strategy used by the processor.

```
(<x>  
  <x><y id="0"/></x>  
  <y id="1"/>  
</x>)/descendant-or-self::x/child::y
```

 \Rightarrow

```
(<y id="0"/>,  
<y id="1"/>)
```

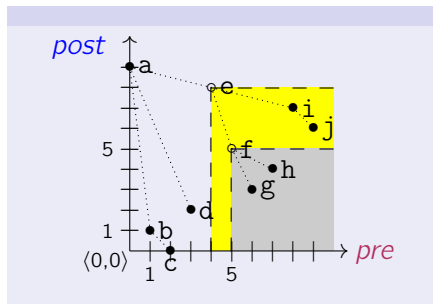
- In many implementations, the `descendant-or-self::x` step will yield the context node sequence `(<x>...</x>, <x>...</x>)` for the `child::y` step.
- Such implementations thus will typically extract `<y id="1"/>` before `<y id="0"/>` from the input document.

Flashback: (e,f)/descendant::node()

Context & frag. encodings

context		
<i>pre</i>	<i>post</i>	...
5	5	
4	8	

accel		
<i>pre</i>	<i>post</i>	...
0	9	
1	1	
2	0	
3	2	
4	8	
5	5	
6	3	
7	4	
8	7	
9	6	

SQL query with expanded *window()* predicate

```

SELECT    DISTINCT v1.*
FROM      context v, accel v1
WHERE     v1.pre > v.pre AND v1.post < v.post
ORDER BY v1.pre

```

Tree awareness ③: Disjoint subtrees

- An XPath location step cs/α is evaluated for a context node **sequence** cs .
 - This “*set-at-a-time*” processing mode is key to the efficient evaluation of queries against bulk data. We want to map this into **set-oriented operations** on the RDBMS.
(Remember: location step is translated into **join** between context node sequence and document encoding table *accel*.)
- But: If two context nodes $c_{i,j} \in cs$ are in α -relationship, **duplicates** and **out-of-order** results may occur.
 - Need efficient way to identify the $c_i \in cs$ which are *not* in α -relationship with any other c_j
(for $\alpha = \text{descendant}$: “ $c_{i,j}$ in disjoint subtrees?”).

Staircase Join: An injection of tree awareness

Since we fail to explain tree properties ② and ③ at the relational language level interface, we opt to **invade the database kernel** in a controlled fashion.⁴⁶

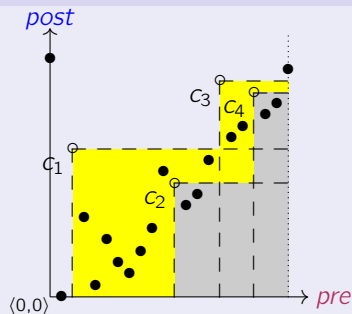
- Inject a new relational operator, **staircase join** \Join , into the relational query engine.
- Query translation and optimization in the presence of \Join continues to work like before (e.g., selection pushdown).
- The \Join algorithm encapsulates the necessary tree knowledge. \Join is a **local change** to the database kernel.

⁴⁶Remember: All of this is optional. XPath Accelerator is a purely relational XML document encoding, working on top of *any* RDBMS.

Tree awareness: Window overlap, coverage

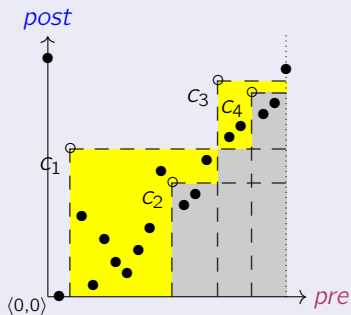
Location step (c_1, c_2, c_3, c_4) /descendant::node(). The pairs (c_1, c_2) and (c_3, c_4) are in descendant-relationship:

Window overlap and coverage (descendant axis)

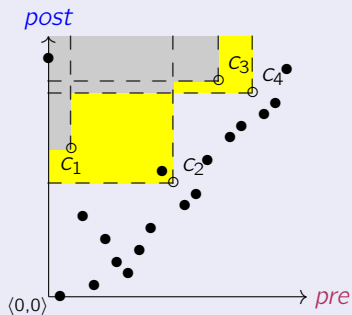


Tree awareness: Window overlap, coverage

Axis window overlap
(descendant axis)

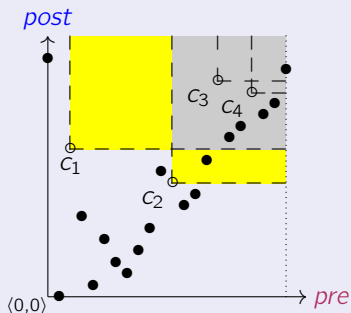


Axis window overlap
(ancestor axis)

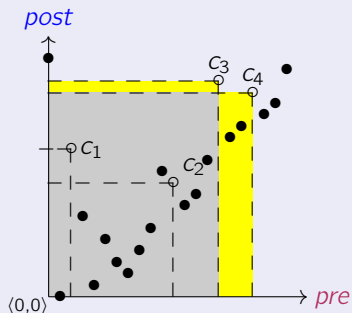


Tree awareness: Window overlap, coverage

Axis window overlap
(following axis)



Axis window overlap
(preceding axis)



Context node sequence pruning

We can turn these observations about axis window overlap and coverage into a simple strategy to **prune the initial context node sequence** for an XPath location step.

Context node sequence pruning

Given cs/α , determine minimal $cs^- \subseteq cs$, such that

$$cs/\alpha = cs^-/\alpha .$$

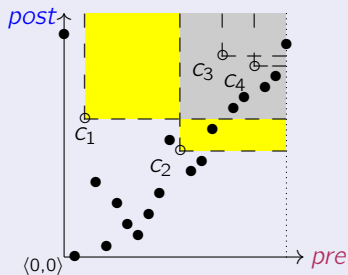
We will see that this minimization leads to axis step evaluation on the *pre/post* plane, which never emits duplicate nodes or out-of-order results.⁴⁷

⁴⁷The ancestor axis needs a bit more work here.

Context node pruning: following axis

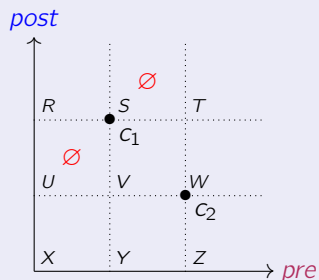
Once context pruning for the following axis is complete, all remaining context nodes relate to each other on the ancestor/descendant axes:

Covering nodes $c_{1,2}$ in descendant relationship



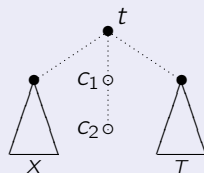
Empty regions in the *pre/post* plane

Relating two context nodes (c_1, c_2) on the plane

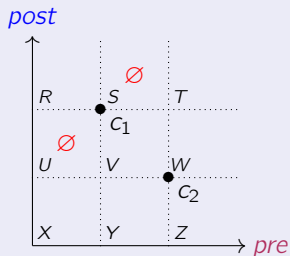


Empty regions?

Given $c_{1,2}$ on the left, why are the regions U, S marked \emptyset guaranteed *to not hold any nodes*?



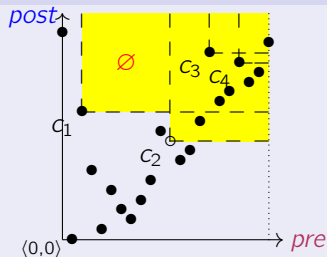
Context pruning (following axis)

 $(c_1, c_2)/\text{following}::\text{node}()$ 

$$\begin{aligned}
 (c_1, c_2)/\text{following}::\text{node}() &\equiv SUTUW \\
 &\equiv T UW \\
 &\equiv (c_2)/\text{following}::\text{node}()
 \end{aligned}$$

Context pruning (following axis)

Context pruning (following axis)



Context pruning (following axis)

Replace context node sequence cs by singleton sequence (c) , $c \in cs$, with $post(c)$ minimal.

Context pruning (preceding axis)

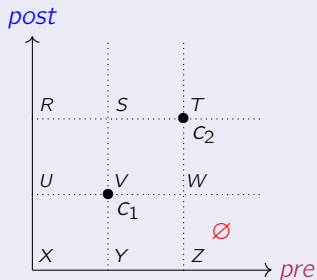
Context pruning (preceding axis)

Replace context node sequence cs by singleton sequence (c) , $c \in cs$, with $pre(c)$ maximal.

- Regardless of initial context size, axes following and preceding yield simple **single region queries**.
- We focus on descendant and ancestor now.

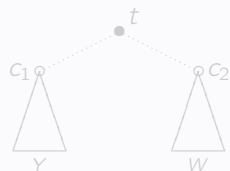
More empty regions

Remaining context nodes
 c_1, c_2 after pruning for
descendant axis



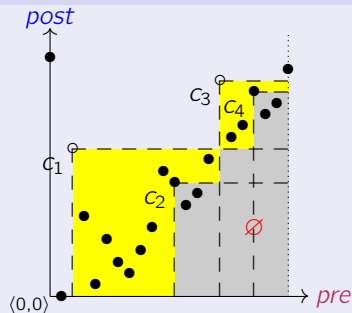
Empty region?

Why is region Z marked \emptyset
guaranteed to be empty?



Context pruning (descendant axis)

Context pruning (descendant axis)



- The region marked \emptyset above is a region of type Z (previous slide). In general, a **non-singleton sequence remains**.

Context pre-processing: Pruning

```
prune_contextdesc(context : TABLE(pre, post))
```

```
begin
```

```
  result ← CREATE TABLE(pre, post);
```

```
  prev ← 0;
```

```
  foreach c in context do
```

```
    /* retain node only if post rank increases */
```

```
    if c.post > prev then
```

```
      APPEND c TO result;
```

```
      prev ← c.post;
```

```
  /* return new context table */
```

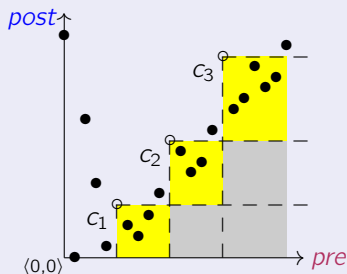
```
  return result;
```

```
end
```

“Staircases” in the *pre/post* plane

Note that after context pruning, the remaining context nodes form a **proper “staircase”** in the plane. (This is an important assumption in the following.)

Context pruning & “staircase”



Flashback: Intersecting ancestor paths

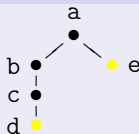
Even with pruning applied, duplicates and out-of-order results may still be generated due to **intersecting ancestor paths**.

- We have observed this before: apply function `ancestors(c_1, c_2)` where c_1 (c_2) denotes the element node with tag d (e) in the sample tree below.
(Nodes $c_{1,2}$ would *not* have been removed during pruning.)

Simulate XPath ancestor via parent axis

```
declare function
  ancestors($n as node(*) as node()*
{ if (fn:empty($n)) then ()
  else (ancestors($n/..), $n/..)
}
```

Sample tree



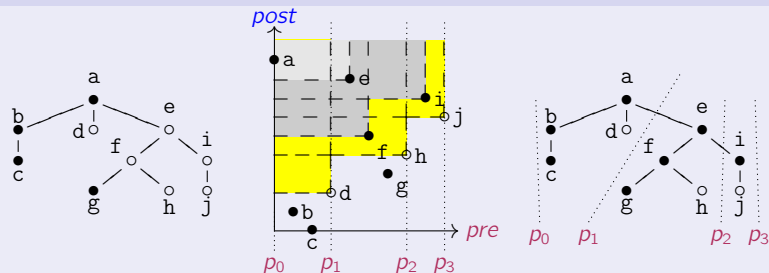
Remember: `ancestors((d,e))` yielded (a, b, a, c) .

Separation of ancestor paths

Idea: try to **separate** the ancestor paths by defining suitable **cuts** in the XML fragment tree.

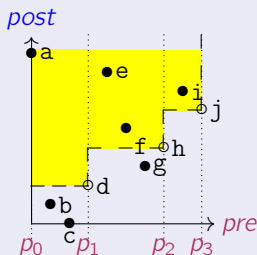
- Stop node-to-root traversal if a cut is encountered.

Path separation (ancestor axis)



Parallel scan along the *pre* dimension

Separating ancestor paths



Scan partitions (intervals): $[p_0, p_1)$, $[p_1, p_2)$, $[p_2, p_3)$.

- Can scan in **parallel**. Partition results may be concatenated.
- Context pruning reduces numbers of partitions to scan.

Basic Staircase Join (descendant)

\sqcup_{desc} (*accel* : TABLE(*pre*, *post*), *context* : TABLE(*pre*, *post*))

begin

result \leftarrow CREATE TABLE(*pre*, *post*);

foreach successive pair (*c*₁, *c*₂) **in** *context* **do**

 | *scanpartition*(*c*₁.*pre* + 1, *c*₂.*pre* - 1, *c*₁.*post*, <);

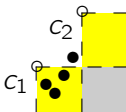
c \leftarrow last node in *context*;

n \leftarrow last node in *accel*;

scanpartition(*c*.*pre* + 1, *n*.*pre*, *c*.*post*, <);

return *result*;

end



Partition scan (sub-routine)

```
scanpartition(pre1, pre2, post,  $\theta$ )
```

```
begin  
  for i from pre1 to pre2 do  
    if accel[i].post  $\theta$  post then  
      APPEND accel[i] TO result;  
end
```

Notation *accel*[*i*] does not imply random access to document encoding:

- Access is strictly **forward sequential** (also *between* invocations of *scanpartition*(·)).

Basic Staircase Join (ancestor)

 $\sqcup_{\text{anc}}(\text{accel} : \text{TABLE}(\text{pre}, \text{post}), \text{context} : \text{TABLE}(\text{pre}, \text{post}))$

begin

$\text{result} \leftarrow \text{CREATE TABLE}(\text{pre}, \text{post});$

$c \leftarrow \text{first node in context};$

$n \leftarrow \text{first node in accel};$

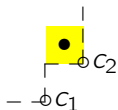
$\text{scanpartition}(n.\text{pre}, c.\text{pre} - 1, c.\text{post}, >);$

foreach successive pair (c_1, c_2) **in** context **do**

$\text{scanpartition}(c_1.\text{pre} + 1, c_2.\text{pre} - 1, c_2.\text{post}, >);$

return result;

end



Basic Staircase Join: Summary

- The operation of **staircase join** is perhaps most closely described as **merge join** with a **dynamic range predicate**: the join predicate traces the staircase boundary:
 - \Join scans the *accel* and *context* tables and populates the *result* table sequentially in document order,
 - \Join scans both tables once for an entire context sequence,
 - \Join never delivers duplicate nodes.
- \Join works correctly only if *prune_context*(\cdot) has previously been applied.
 - *prune_context*(\cdot) may be **inlined** into \Join , thus performing context pruning *on-the-fly*.



Pruning on-the-fly

```
⌋desc(accel:TABLE(pre, post), context:TABLE(pre, post))
```

begin

```
  result ← CREATE TABLE(pre, post);
```

```
  c1 ← first node in context;
```

```
  while (c2 ← next node in context) do
```

```
    if c2.post < c1.post then
```

```
      | /* prune */
```

```
    else
```

```
      | scanpartition(c1.pre + 1, c2.pre - 1, c1.post, <);
```

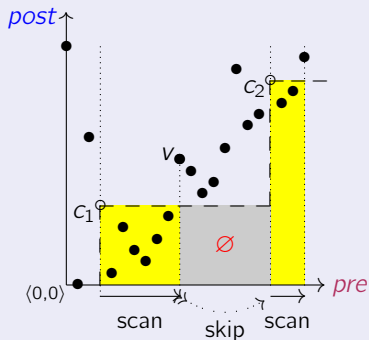
```
      | c1 ← c2;
```

```
  return result;
```

end

Skip ahead, if possible

$(c_1, c_2)/\text{descendant}::\text{node}()$



- While scanning the partition associated with $c_{1,2}$:
 - v is outside staircase boundary, thus not part of the result.
 - No node beyond v in result (\emptyset -region of type Z).
- ⇒ Can **terminate scan early and skip ahead** to $pre(c_2)$.

Skipping for the descendant axis

*scanpartition*_{desc}(*pre*₁, *pre*₂, *post*)

```
begin  
  for i from pre1 to pre2 do  
    if accel[i].post < post then  
      | APPEND accel[i] TO result;  
    else  
      | /* on the first offside node, terminate scan */ break;  
end
```

Note: keyword **break** transfers control out of innermost enclosing loop (cf. C, Java).

Effectiveness of skipping

- Enable skipping in $scanpartition(\cdot)$. Then, for each node in $context$, we either
 - ① hit a node to be copied into table $result$, or
 - ② encounter an offside node (node v on slide 780) which leads to a skip to a known pre value (\rightarrow positional access).
- To produce the final result, \sphericalangle thus never touches more than

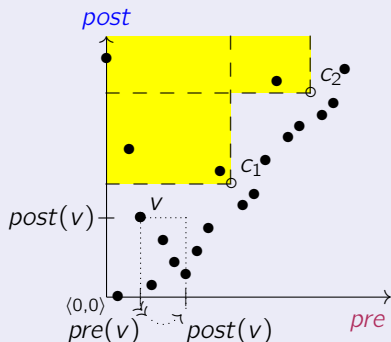
$$| context | + | result |$$

nodes in the plane (without skipping: $| context | + | accel |$).

- In practice: $> 90\%$ of nodes in table $accel$ are skipped.

Skipping for the ancestor axis

Skipping over the subtree of v



Encounter v outside staircase boundary

↓

v **and subtree below** v in preceding axis of context node.

How far to skip?

Conservative estimate:

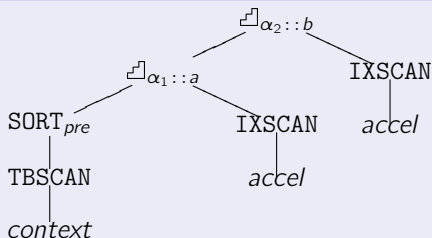
$$size(v) \geq post(v) - pre(v)$$

Injecting \sqcup into PostgreSQL

PostgreSQL (<http://postgresql.org/>): Conventional disk-based RDBMS, SQL interface.

- Detection of \sqcup applicability on SQL level (self-join with conjunctive range selection on columns of type `tree`⁴⁸).

Algebraic query plan for two-step XPath location path



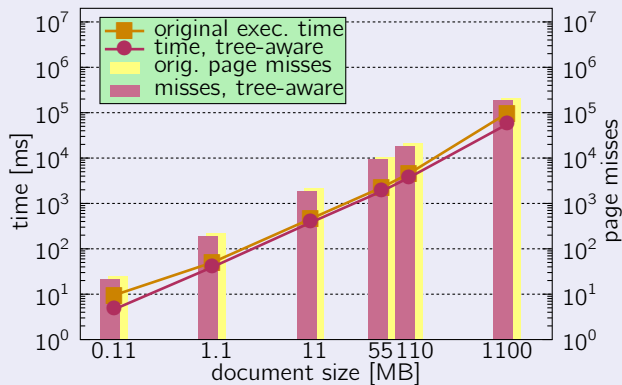
⁴⁸PostgreSQL is highly extensible, also permits introduction of new column types.

Injecting \surd into PostgreSQL

- Create clustered ascending B-tree index on column *pre* of table *accel*.
 - Standard no-frills PostgreSQL B-tree index, entered with search predicates of the form $pre \underset{?}{\geq} c.pre$ (*c* context node).
 - B-tree on column *pre* also used for **skipping**.
- Following performance figures obtained on a 2.2 GHz Dual Intel™ Pentium 4, 2 GB RAM, PostgreSQL 7.3.3.
 - Compares \surd -enabled (tree-aware) PostgreSQL with vanilla PostgreSQL instance.
 - Evaluate XPath location path `/descendant::a/α::b` on document instances of up to 1.1 GB serialized size.

Injecting λ into PostgreSQL

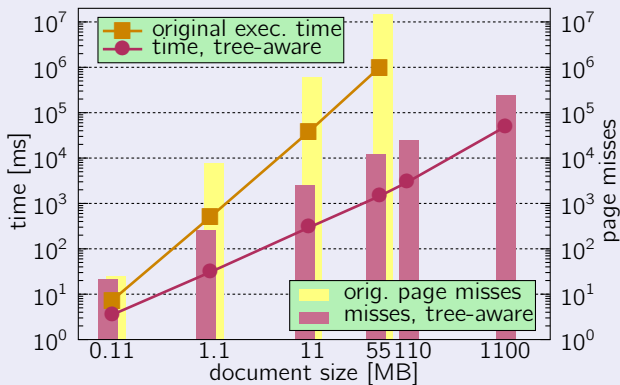
/descendant::a/descendant::b



Injecting α into PostgreSQL

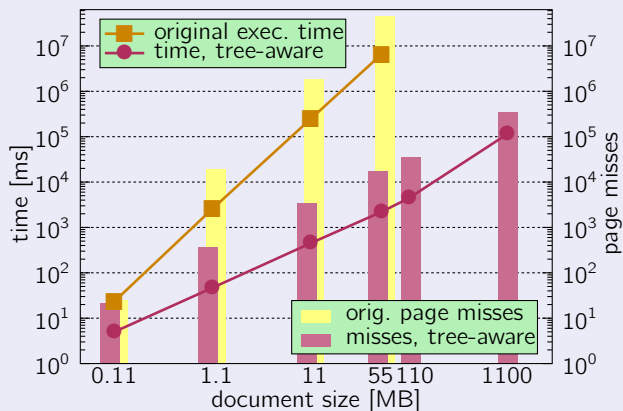
For $\alpha = \text{descendant}$ observe:

- For *both* PostgreSQL instances, query evaluation time grows **linearly** with the input XML document size (since the results size grows linearly).
- For the original instance, this is due to **window shrink-wrapping** (expressible at the SQL level).

Injecting λ into PostgreSQL`/descendant::a/ancestor::b`

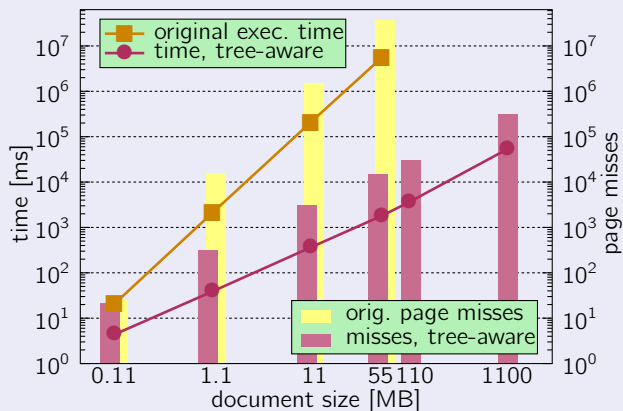
Injecting \bowtie into PostgreSQL

- **For** $\alpha \in \{\text{ancestor, preceding, following}\}$ **observe:**
 - For the \bowtie -**enabled** PostgreSQL instance, query evaluation time grows **linearly** with the input XML document (and result) size.
For the **original** instance, query evaluation time grows **quadratically** ($| accel |$ scans of table $accel$ performed).
 - Original instance is incapable of completing experiment in reasonable time (> 15 mins for XML input size of 55 MB).
- **Generally:**
 - The number of **buffer page misses** (= necessary I/O operations) determines evaluation time.

Injecting ξ into PostgreSQL
$$/descendant::a/preceding::b$$


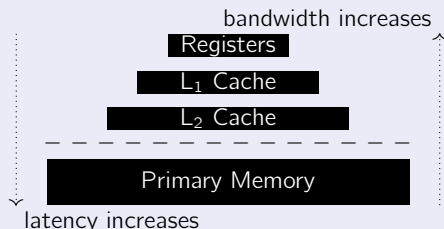
Injecting ξ into PostgreSQL

```
/descendant::a/following::b
```



MonetDB/XQuery: Targetting modern CPU/memory architectures

Memory Hierarchy



- Computation performed with CPU registers only.
- **Cache miss may escalate:** L₁ → L₂ → RAM, data transport all the way back: L₁ ← L₂ ← RAM.
- Data transport in **cache line granularity**.

CPU/cache characteristics

Intel™ Dual Pentium 4 (Xeon)⁴⁹

CPU/Cache Characteristics

Clock frequency		2.2 GHz
L ₁ /L ₂ cache size		8 kB/512 kB
L ₁ /L ₂ cache line size	LS _{L₁} /LS _{L₂}	32 byte/128 byte
L ₁ miss latency	L _{L₁}	28 cycles $\hat{=}$ 12.7 ns
L ₂ miss latency	L _{L₂}	387 cycles $\hat{=}$ 176 ns

- For this CPU, a **full cache miss** implies a **stall of the CPU** for $28 + 387 = 415$ cycles (cy).



⁴⁹Measure these characteristics for your CPU with Stefan Manegold's *Calibrator*, <http://monetdb.cwi.nl/Calibrator/>.

Staircase join: Wrap-up

- Standard B⁺-tree implementation suffices to support \sqsubset .
 - A **single** B⁺-tree indexes the *pre/post* plane as well as the context node sequence.
 - ⇒ Less index pages compete for valuable buffer space.
- \sqsubset derives pruning and skipping information from the plane itself, using **simple integer arithmetic and comparisons**.
 - Simple \sqsubset logic leads to **simple memory access pattern and control flow**.
 - ⇒ Branches in inner \sqsubset loops are highly predictable, facilitating **speculative execution** in the CPU.

Predictable branches?

Explain why!