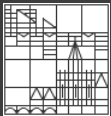


EDBT Summerschool 2004

Santa Margherita di Pula - Sardinia (Italy)
September 6-10, 2004

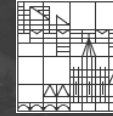
Universität Konstanz



University of Twente
The Netherlands

Fully (or Purely) Relational XPath and XQuery Processors

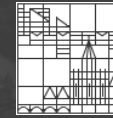
Maurice van Keulen, University of Twente, The Netherlands
Torsten Grust, University of Konstanz, Germany



Turn an ordinary RDBMS into an efficient XQuery engine for high volume XML document collections

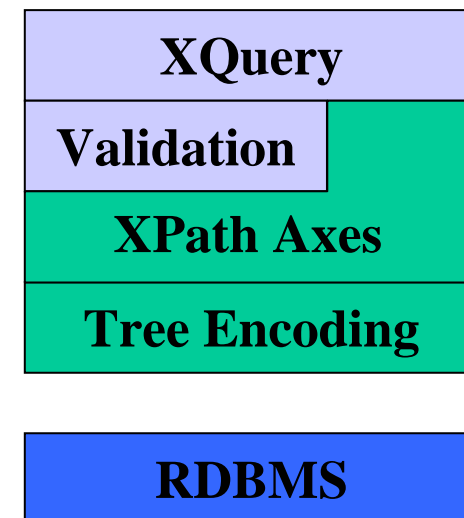
- Existing relational DBMS as XQuery backend
 - Complete relational representation of XML
 - Efficient evaluation of XQuery inside the DBMS
 - By making RDBMS more *tree aware*, significant improvements in efficiency can be obtained
 - Support full XQuery exactly in accordance with XQuery semantics
 - Keep change to RDBMS engine minimal

A purely relational XQuery processing stack

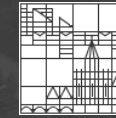


Need for **efficient XQuery evaluation** drives the construction of the following development stack:

Compilation of XQuery core to relational algebra
(wednesday morning by Torsten Grust) →
Validation of encoded XML against schema definition
(wednesday morning by Torsten Grust) →
Relational XPath evaluation (also **today**) →
Relational XML encoding (**now**) →
SQL, relational algebra



Data- vs. Document-centric XML



- Data-centric

- Highly structured
- Usage stems from exchange of data from databases

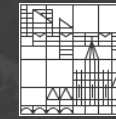
```
<site>
  <item ID="I001">
    <name>Chair</name>
    <description>This chair is in good condition ...
  </description>
</item>
  <item ID="I002">
    <name>Table</table>
    <description>...</description>
  </item>
  ...
</site>
```

- Document-centric

- Semi-structured
- Embedded tags
- Fulltext search
- Usage stems from exchange of formatted text

```
<memo>
  <author>John Doe</author>
  <title>...</title>
  <body>
    This memo is meant for all persons responsible for
    <list bullets=1>
      <item>either <em>customers</em> abroad,</item>
      <item>or <em>suppliers</em> abroad.</item>
    </list>
  ...
</memo>
```

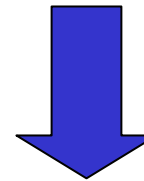
Mapping a DTD to a relational schema: Inlining



```
<!ELEMENT site (item*)>
<!ELEMENT item (name,description*)>
<!ATTLIST item (id ID #REQUIRED)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
```

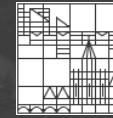
Disadvantages

- Suitable for data-centric XML only
- DTD or XML schema required

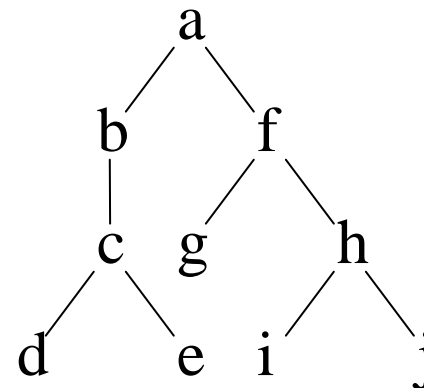


```
CREATE TABLE item (
  id INTEGER NOT NULL PRIMARY KEY,
  name VARCHAR(250) NOT NULL
)
CREATE TABLE description (
  itemID INTEGER NOT NULL,
  pcdData TEXT NOT NULL
)
```

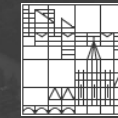
XML document and its tree shape



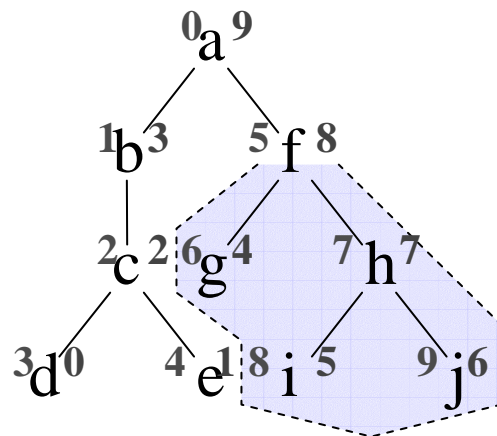
```
<a>  
  <b>  
    <c>  
      <d/> <e/>  
    </c>  
  </b>  
  <f>  
    <g/>  
    <h>  
      <i/> <j/>  
    </h>  
  </f>  
</a>
```



Document encoding

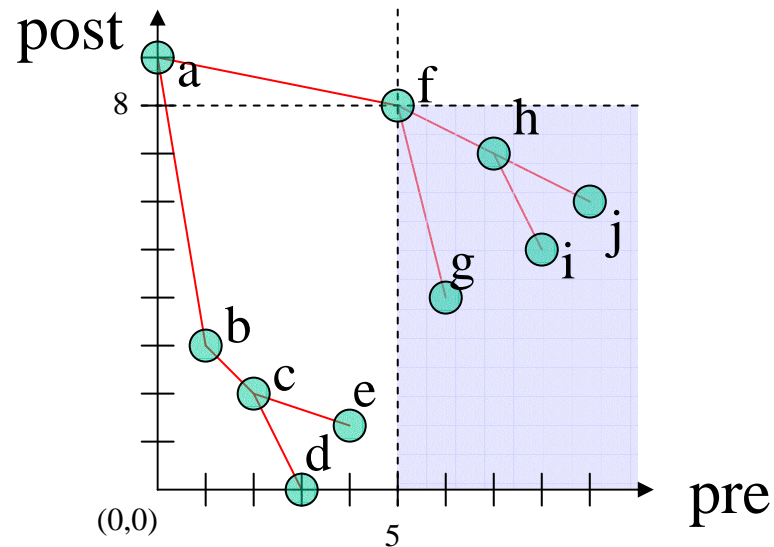


XML tree



Preorder/postorder
rank assignment

Pre/Post Plane Encoding

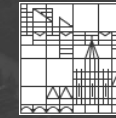


RDBMS table

| | pre | post |
|---|-----|------|
| a | 0 | 9 |
| b | 1 | 3 |
| c | 2 | 2 |
| d | 3 | 0 |
| e | 4 | 1 |
| f | 5 | 8 |
| g | 6 | 4 |
| h | 7 | 7 |
| i | 8 | 5 |
| j | 9 | 6 |

Preorder coincides with order of opening tags
Postorder coincides with order of closing tags

Document storage



document

| pre | post | level | kind | name |
|-----|------|-------|------|-------|
| 0 | 9 | 0 | doc | a.xml |
| 1 | 3 | 1 | elem | b |
| 2 | 2 | 2 | elem | c |
| 3 | 0 | 3 | attr | id |
| 4 | 1 | 3 | text | |

text

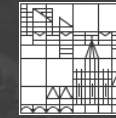
| pre | text |
|-----|-------|
| 4 | "..." |

attribute

| pre | name | value |
|-----|------|-------|
| 3 | id | "10" |

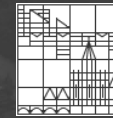
One node in tree =
one row in table

Loading & Serialization



- **Loading:** “how to convert (textual) XML document to rows in table structure?”
 - Only one pass needed!
- *At startElement:*
 - (1) assign pre + increase
 - (2) push v on stack
 - (3) process attributes
 - *At endElement:*
 - (1) pop v from stack
 - (2) assign post + increase
 - (3) insert into table
- **Serialization:** “how to convert table contents to (textual) XML-doc?”
 - Only one pass needed!
- foreach v in table do
 - (1) for each node on stack with $post < post(v)$, pop from stack & print end tag
 - (2) if v is element, push on stack, print start tag; else ..
 - perform (1) for all remaining nodes on stack

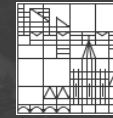
Exercise 1



```
<?xml version="1.0" encoding="iso-8859-1"?>
<orders>
  <order total="10.89">
    <line>
      <article id="10">Pencil</article>
      <price>1.95</price>
    </line>
    <line>
      <article id="23">
        Paper (<weight>80gr</weight>)
      </article>
      <price>6.99</price>
    </line>
  </order>
  <order total="1.95">
    <line>
      <article id="10">Pencil</article>
      <price>1.95</price>
    </line>
  </order>
</orders>
```

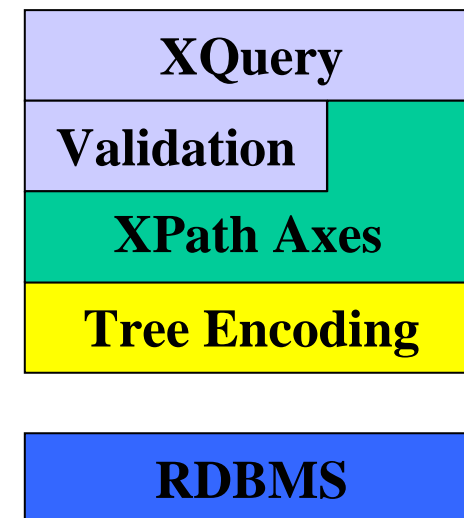
1. Draw tree for this XML document
2. Assign pre-order and postorder ranks
3. Fill table *document*

A purely relational XQuery processing stack



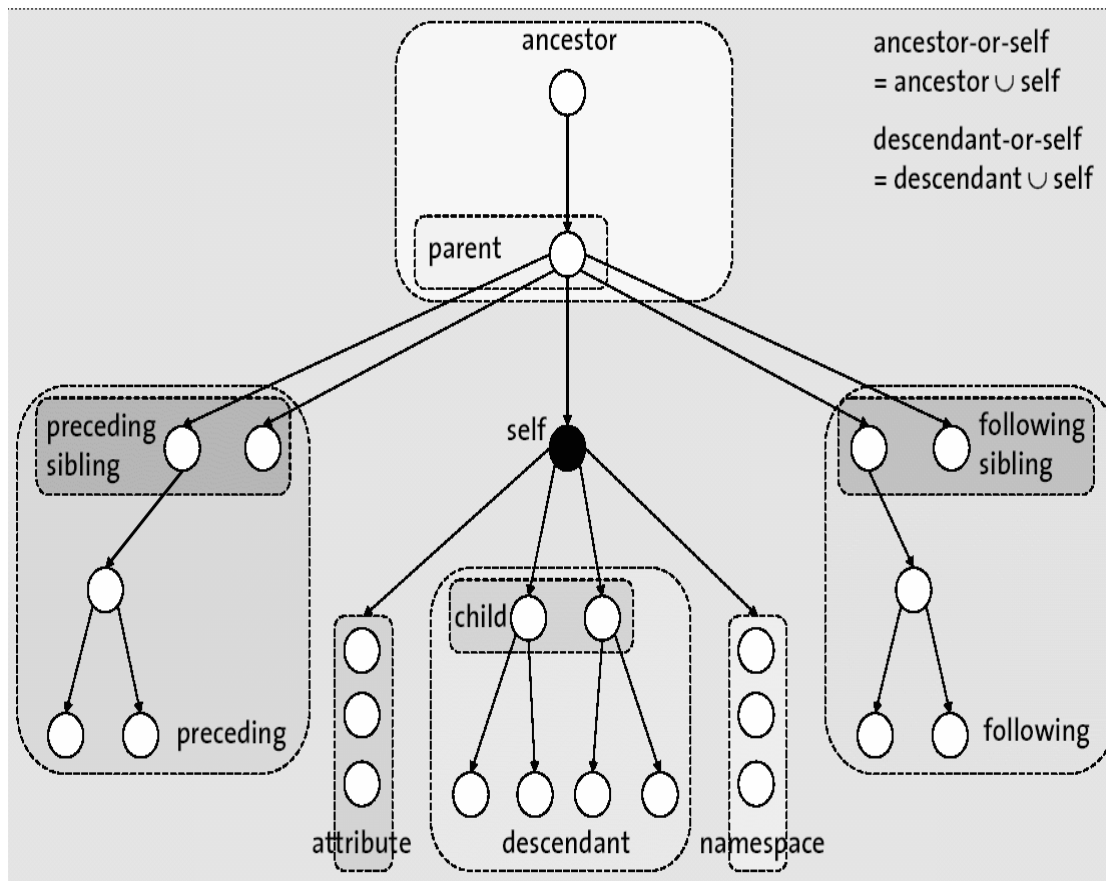
Need for **efficient XQuery evaluation** drives the construction of the following development stack:

Compilation of XQuery core to relational algebra
(wednesday morning by Torsten Grust) →
Validation of encoded XML against schema definition
(wednesday morning by Torsten Grust) →
Relational XPath evaluation (**now**) →
XPath accelerator ✓
SQL, relational algebra

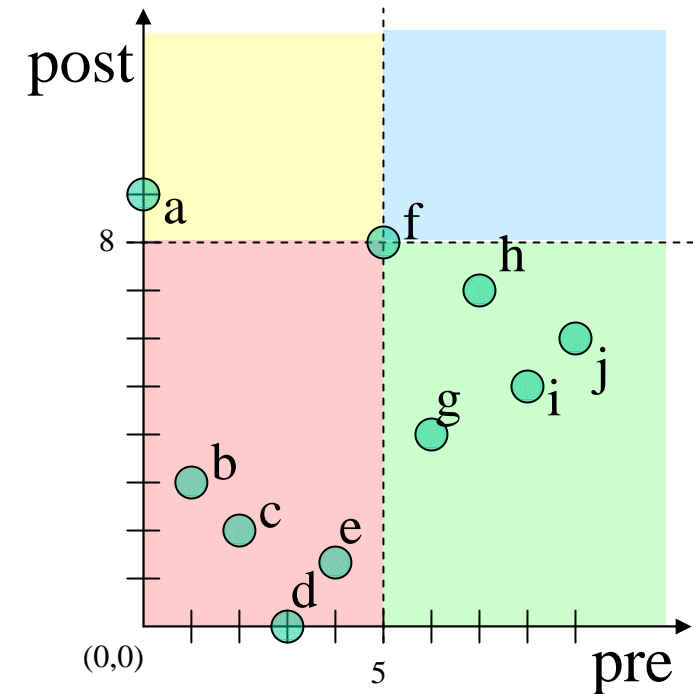


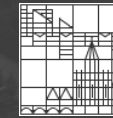
XPath axes

XPath axes



in the pre/post plane

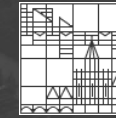




Window(α, v)

| Axis α | pre | post | Level | kind |
|---------------|---|--|-------------------------|------|
| child | $\langle \text{pre}(v), \infty \rangle$ | $[0, \text{post}(v) \rangle$ | $= \text{level}(v) + 1$ | elem |
| descendant | $\langle \text{pre}(v), \infty \rangle$ | $[0, \text{post}(v) \rangle$ | * | elem |
| desc-or-self | $[\text{pre}(v), \infty \rangle$ | $[0, \text{post}(v)]$ | * | elem |
| parent | $[0, \text{pre}(v) \rangle$ | $\langle \text{post}(v), \infty \rangle$ | $= \text{level}(v) - 1$ | elem |
| ancestor | $[0, \text{pre}(v) \rangle$ | $\langle \text{post}(v), \infty \rangle$ | * | elem |
| anc-or-self | $[0, \text{pre}(v)]$ | $[\text{post}(v), \infty \rangle$ | * | elem |
| following | $\langle \text{pre}(v), \infty \rangle$ | $\langle \text{post}(v), \infty \rangle$ | * | elem |
| preceding | $[0, \text{pre}(v) \rangle$ | $[0, \text{post}(v) \rangle$ | * | elem |
| foll-sibling | $\langle \text{pre}(v), \infty \rangle$ | $\langle \text{post}(v), \infty \rangle$ | $= \text{level}(v)$ | elem |
| prec-sibling | $[0, \text{pre}(v) \rangle$ | $[0, \text{post}(v) \rangle$ | $= \text{level}(v)$ | elem |
| attribute | $\langle \text{pre}(v), \infty \rangle$ | $[0, \text{post}(v) \rangle$ | $= \text{level}(v) + 1$ | attr |

XPath evaluation (approach)



XPath path expression $s_1/s_2/.../s_n$

- Each step s_i is of the form *axis::nodetest*
- Each step s_i results in **context node sequence** for step s_{i+1}

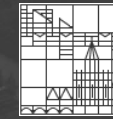
Evaluation in DBMS

- Apply each location step to *all* nodes in context (bulk-oriented query processing)
- **Preserve document order and not produce duplicate nodes**

Initial focus on *major axis steps*

- Major: *ancestor, descendant, preceding, and following*
- Other axes define efficiently computable subsets of the four major axes

XPath evaluation (SQL)



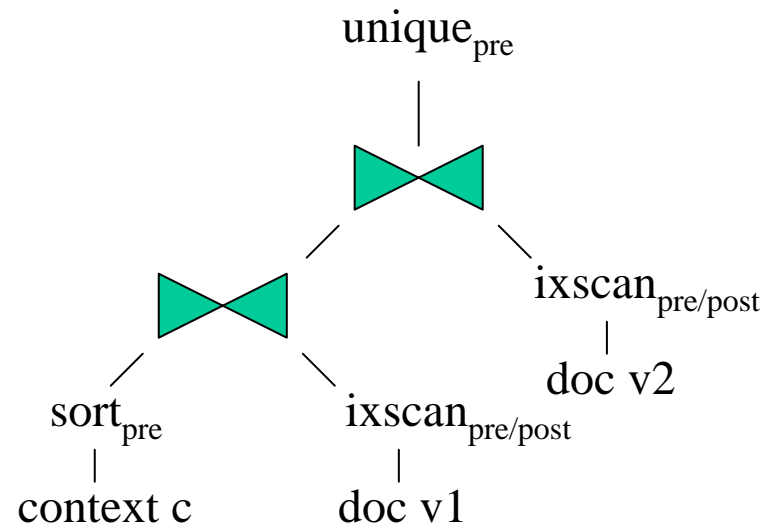
Example query:

`context/following::node()/descendant::node()`

SQL Query

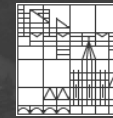
```
SELECT DISTINCT v2.pre
  FROM context c, doc v1, doc v2
 WHERE v1.pre > c.pre
       AND v1.post > c.post
       AND v2.pre > v1.pre
       AND v2.post < v1.post
 ORDER BY v2.pre
```

IBM DB2 Query Plan

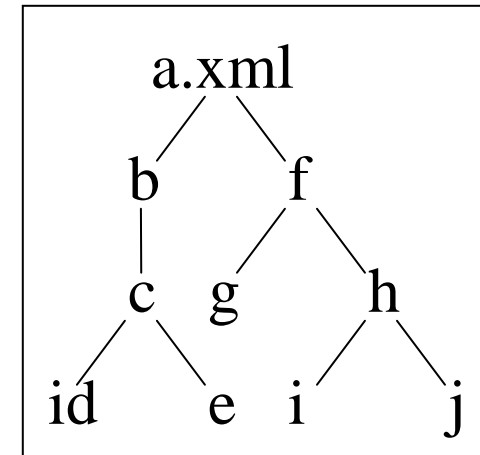


Efficient with concatenated *pre/post* B-tree over *doc*,
but remains **ignorant of many useful tree properties**

Exercise 2



- Write the SQL-query for
`/descendant::*[./child::e]`
(all descendant elements
that have a child with name 'e')



Example query:

```
context/following::node()/descendant::node()
```

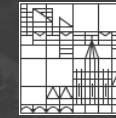
SQL Query

```
SELECT DISTINCT v2.pre
FROM context c, doc v1, doc v2
WHERE v1.pre > c.pre
      AND v1.post > c.post
      AND v2.pre > v1.pre
      AND v2.post < v1.post
ORDER BY v2.pre
```

document

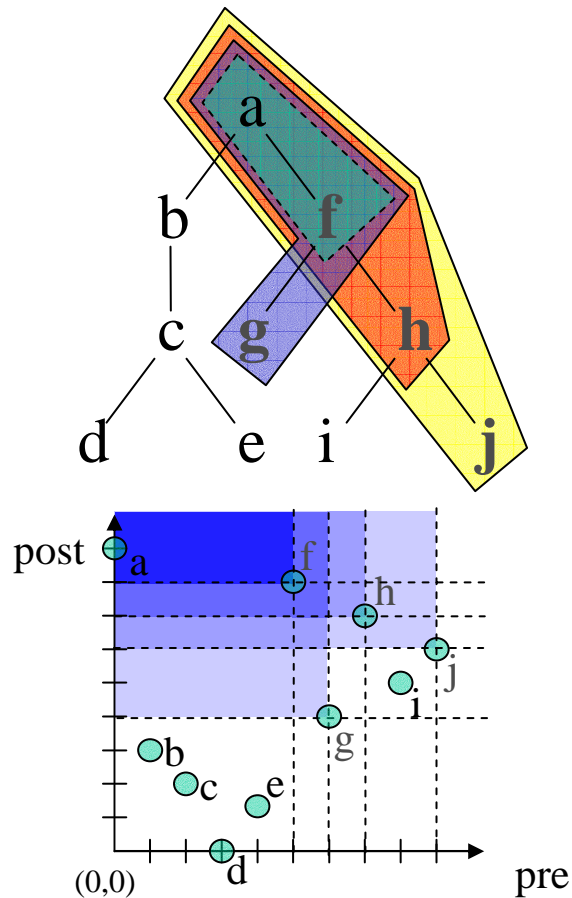
| pre | post | level | kind | name |
|-----|------|-------|------|-------|
| 0 | 9 | 0 | doc | a.xml |
| 1 | 3 | 1 | elem | b |
| 2 | 2 | 2 | elem | c |
| 3 | 0 | 3 | attr | id |
| 4 | 1 | 3 | elem | e |

Tree knowledge 1: pruning

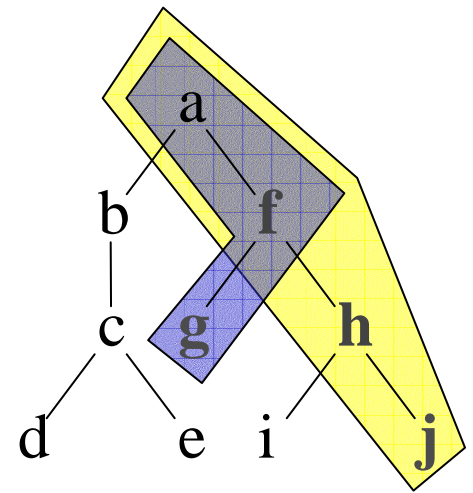


Avoiding duplicates: $(f, g, h, j) / \text{ancestor-or-self}::*$

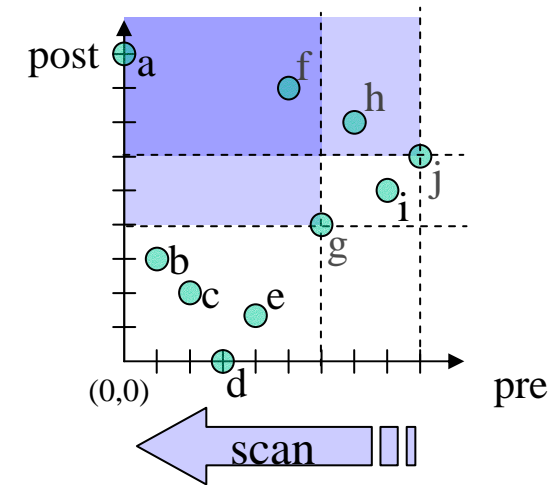
$= (g, j) / \text{ancestor-or-self}::*$



a
f
a
f
g
a
f
h
a
f
h
j

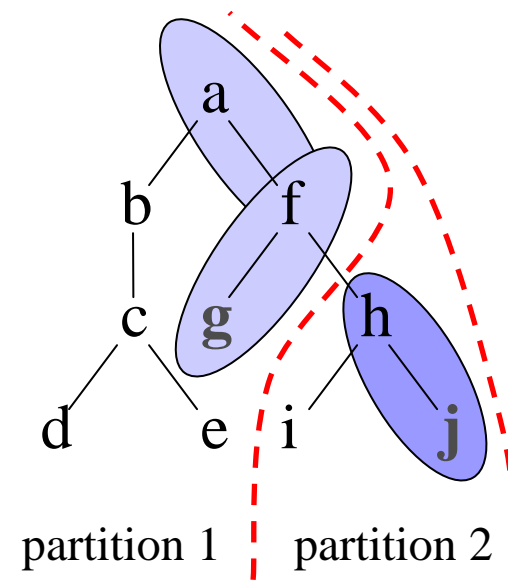
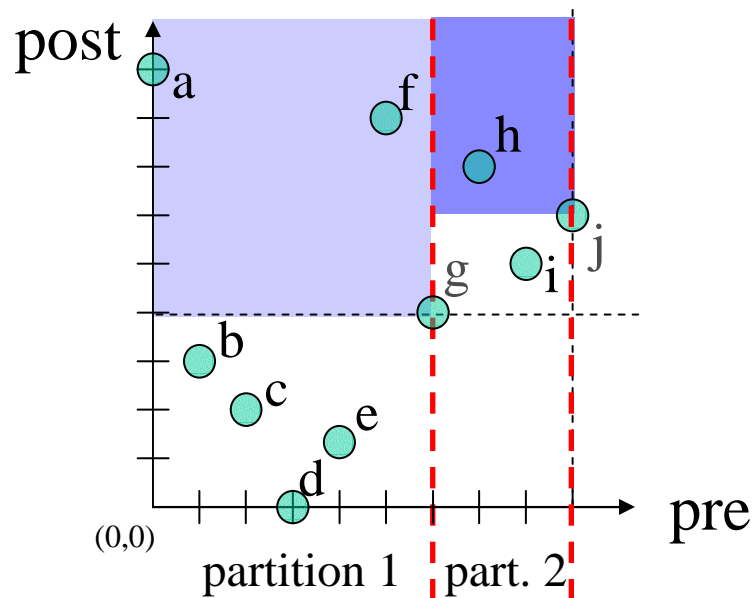
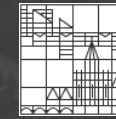


a
f
g
a
f
h
j

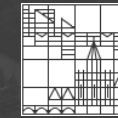



Pruning requires only a single sequential scan over the *pre/post* table.

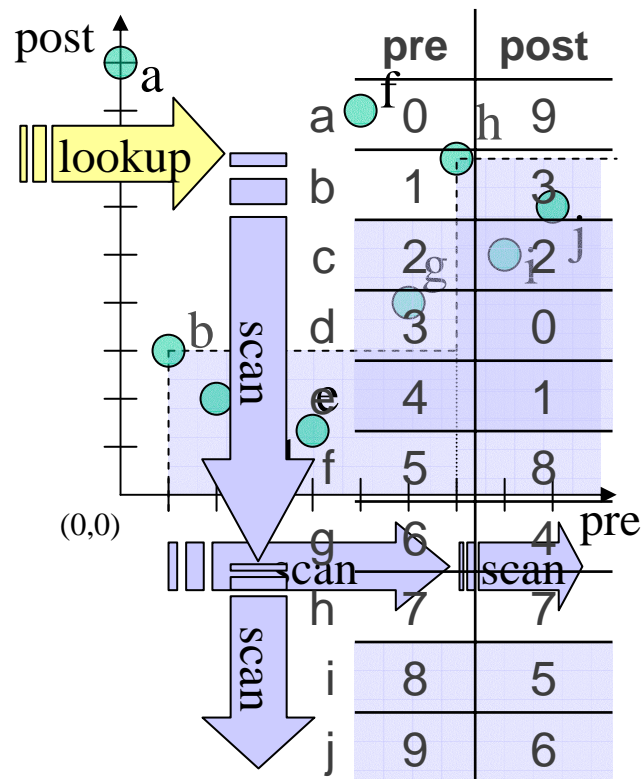
Tree knowledge 2: partitioning




Basic Staircase Join Algorithm



(b,c,h)  *doc*
desc

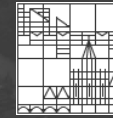


```

context  doc =
begin desc
  result=new table(pre,post);
  /* partition c_from ... c_to */
  c_from = first node in context;
  while (c_to = next node in context) do
    if c_to.post < c_from.post then
      /* prune */
    else
      scanpartition_desc(c_from.pre+1,c_to.pre-1,
        c_from.post);
      c_from = c_to;
  n=last node in doc;
  scanpartition_desc(c_from.pre+1,n.pre,c_from.post);
  return result;
end

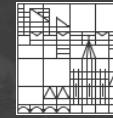
scanpartition_desc(pre1,pre2,post)
begin
  for i from pre1 to pre2 do
    if doc[i].post < post then
      append doc[i] to result;
    end
  end
end
    
```

Characteristics of basic algorithm



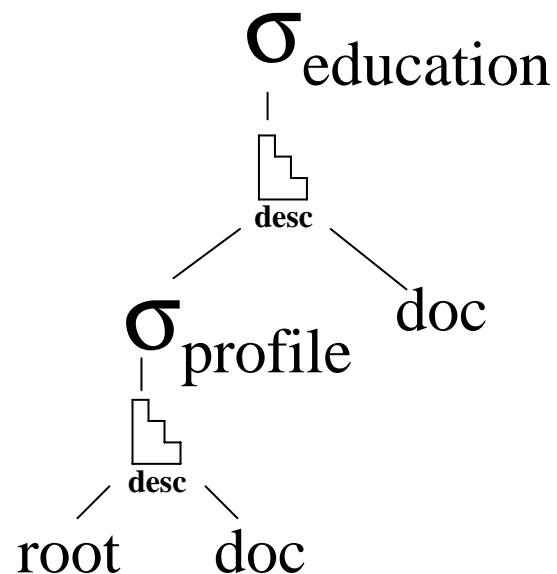
- 1) It scans doc and context tables *sequentially*
 - 2) It scans both tables only *once* for an entire context node sequence
 - 3) It *never* delivers duplicate nodes
 - 4) Result nodes are produced in *document order*
 - 5) Input for staircase join can be *any* node sequence
- (3) + (4)
⇒ no post-processing (unique/sort) is needed to comply to XPath semantics

Query plan with staircase join

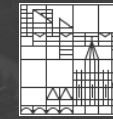


Example query: `/descendant::profile/descendant::education`

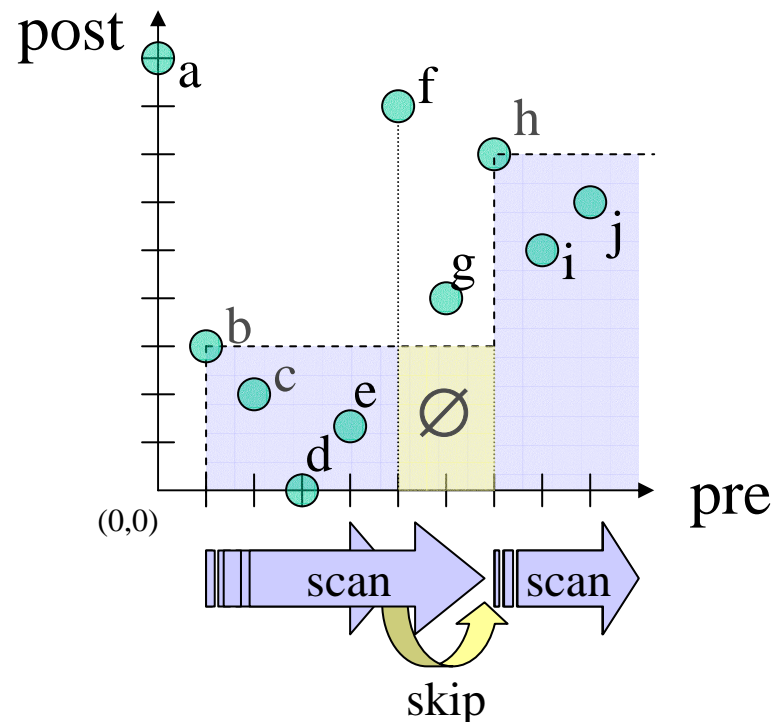
Query plan



Tree Knowledge 3: Skipping



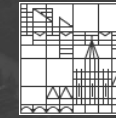
$(b, c, h) / \text{descendant} :: *$



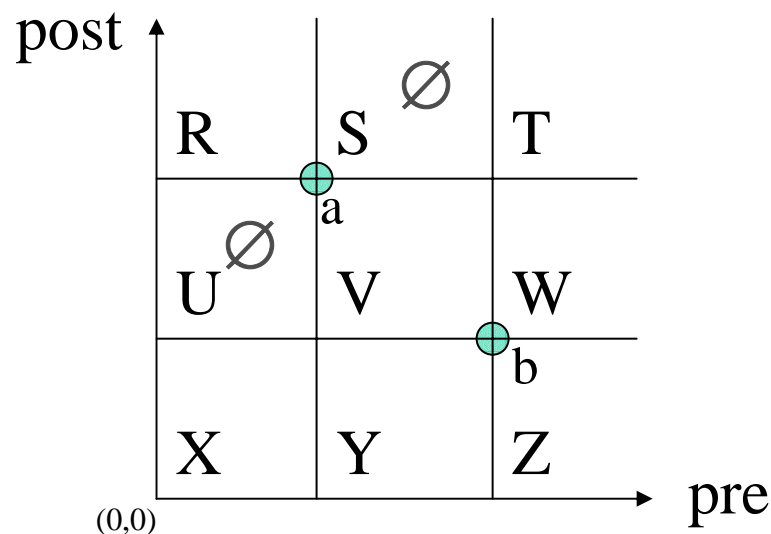
```
scanpartition_desc(pre1,pre2,post)
begin for i from pre1 to pre2 do
  if doc[i].post < post then
    append doc[i] to result;
  else break; /* skip */
end
```

- Never touch more than $|context| + |result|$ nodes
- A characterization of the location of 'f' can be found **before** scan starts (*estimated skipping*)

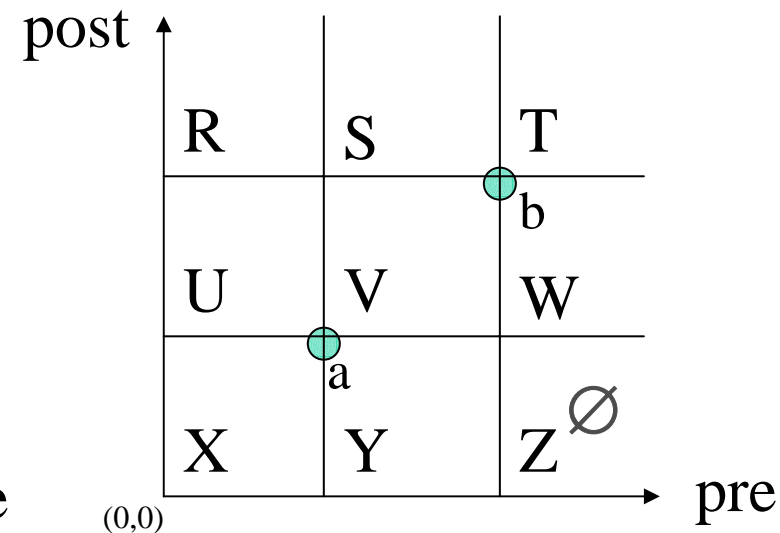
Tree Knowledge basis: Empty regions in plane



For *any* two nodes 'a' and 'b'

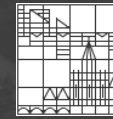


(1) Nodes *a* and *b* relate to each other on the ancestor/descendant axis.



(2) Nodes *a* and *b* relate to each other on the preceding/following axis.

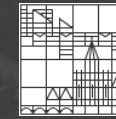
Tree Knowledge 4: Shrink-wrapping the //-axis



There is a very useful law that relates preorder rank, postorder rank, and the number of descendants of a node.

- **$\text{pre}(v) = |v/\text{ancestor}| + |v/\text{preceding}| + 1$**
(in a preorder traversal of a tree, you encounter *all* ancestors and preceding nodes of a node v)
- **$\text{post}(v) = |v/\text{descendant}| + |v/\text{preceding}| + 1$**
(in a postorder traversal of a tree, you encounter *all* descendants and preceding nodes of a node v)
- \Rightarrow **$\text{post}(v) - \text{pre}(v) = |v/\text{descendant}| - |v/\text{ancestor}|$**
- $|v/\text{descendant}|$ is often called **size(v)**
 $|v/\text{ancestor}|$ is often called **level(v)**
- Conclusion: **$\text{size}(v) = \text{post}(v) - \text{pre}(v) + \text{level}(v)$**

Tree knowledge 4: Shrink-wrapping the // -axis



For any v : $\text{post}(v) - \text{pre}(v) + \text{level}(v) = \text{size}(v)$

If $\text{level}(v)$ is unavailable, we can estimate the number of descendants, since $\text{level}(v) \leq \text{height}$:

$$\text{post}(v) - \text{pre}(v) \leq \text{size}(v) \leq \text{post}(v) - \text{pre}(v) + \text{height}$$

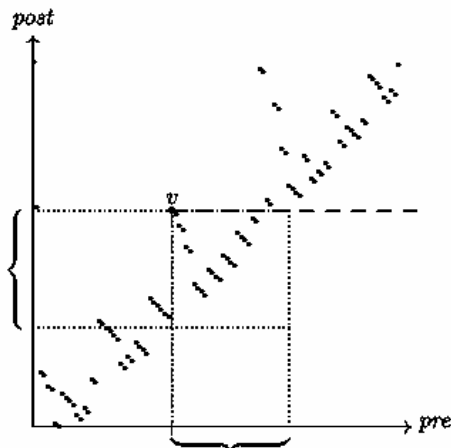
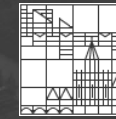


Figure 7: Original (---) and shrunk (.....) pre and post scan ranges for a // -step to be taken from v .

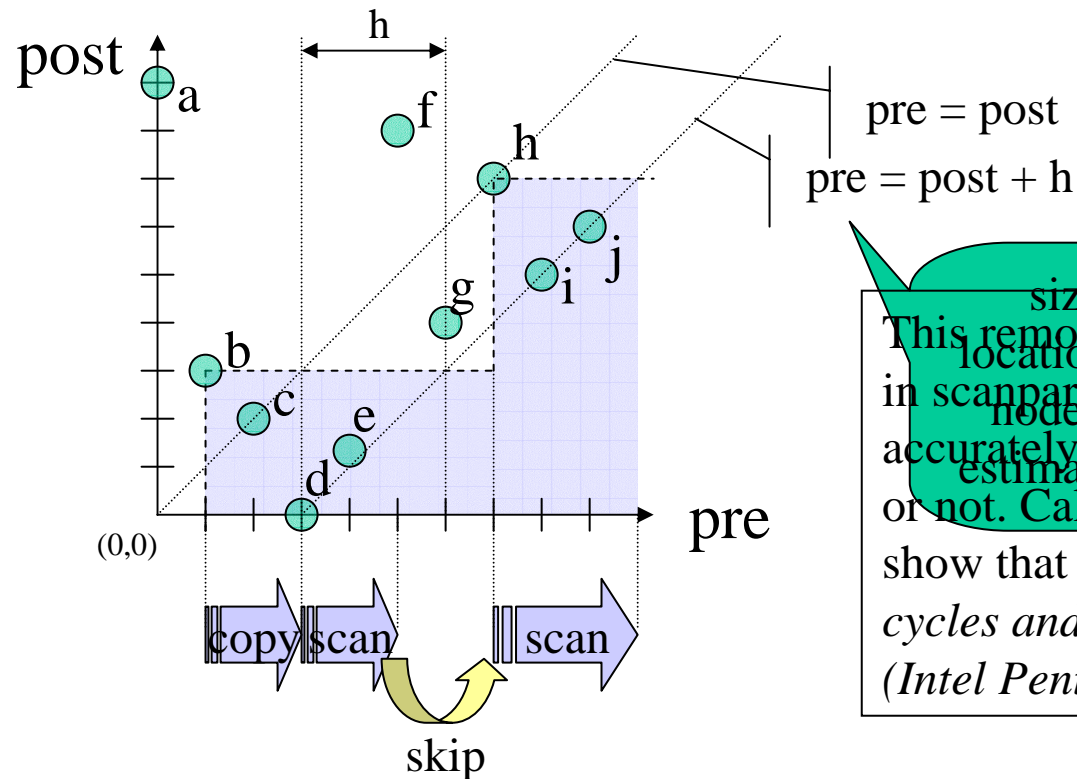
Hence: $\text{window}(\text{descendant}, v) =$
 $[\text{pre}(v), \text{post}(v) + \text{level}(v)]$,
 $[\text{pre}(v) - \text{level}(v), \text{post}(v)]$
or (estimated):
 $[\text{pre}(v), \text{post}(v) + \text{height}]$,
 $[\text{pre}(v) - \text{height}, \text{post}(v)]$

Estimation-based skipping



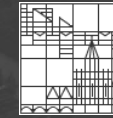
$$size(v) = post(v) - pre(v) + level(v)$$

$$\Rightarrow post(v) - pre(v) \leq size(v) \leq post(v) - pre(v) + h$$

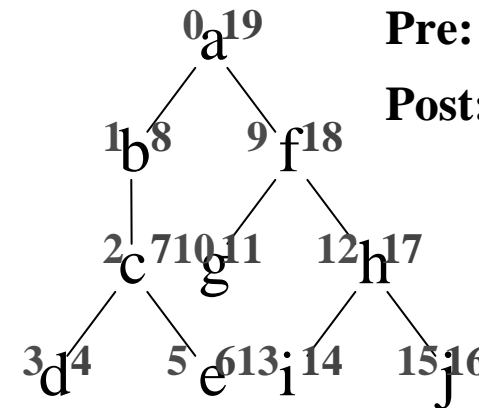


size(b) determines location of first following node (f), which can be estimated with these lines. This removes an IF from the inner loop in scanpartition. This allows a CPU to accurately predict if branches are taken or not. Calculations with CPU cycles show that *scan-loop* takes 17 CPU cycles and *copy-loop* only 5! (Intel Pentium 4)

A stretched pre/post plane

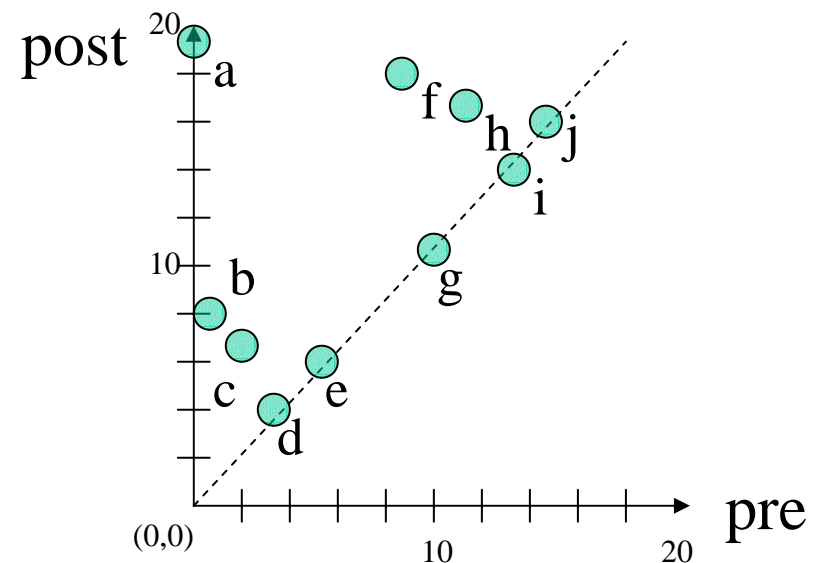


- The exact pre and post ranks are unimportant, only the order
 - Couple pre and post
- For descendants:
 - $\text{pre}(c) \leq \text{pre}(v) \leq \text{post}(c)$
 - $\text{pre}(c) \leq \text{post}(v) \leq \text{post}(c)$
- We can choose query window:
 - pre between $\text{pre}(c)$, $\text{post}(c)$
 - post between $\text{pre}(c)$, $\text{post}(c)$
- Furthermore
 - $\text{size}(v) = \frac{1}{2} (\text{post}(v) - \text{pre}(v) - 1)$

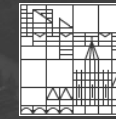


Pre: $0 < 1 < 2 < 3 < 5 < 9 < \dots$

Post: $4 < 7 < 8 < 11 < 14 < \dots$

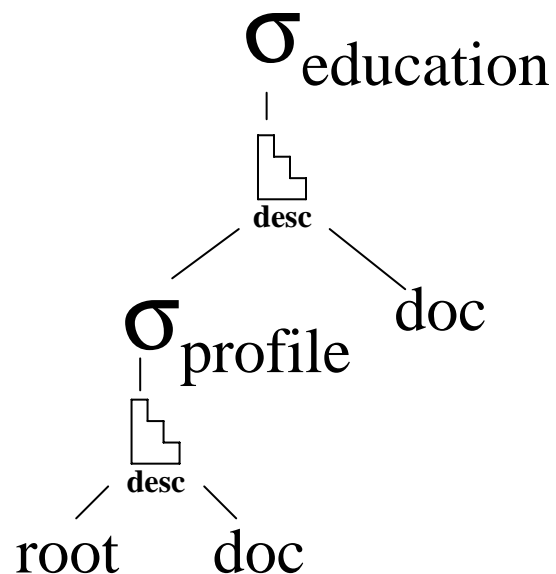


Query optimization



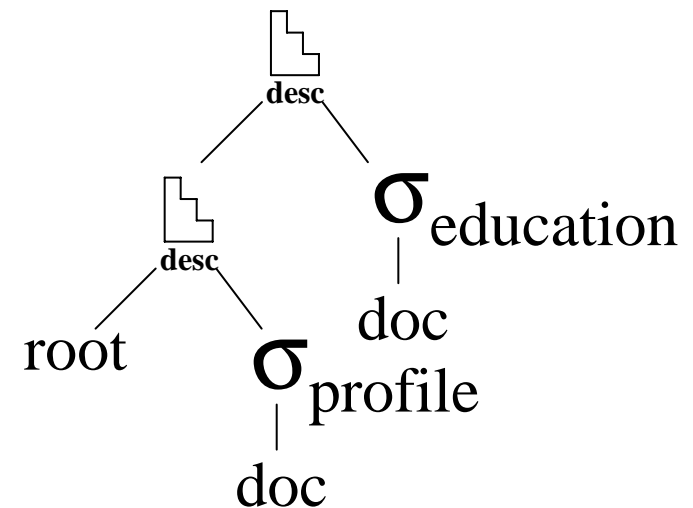
Example query: `/descendant::profile/descendant::education`

Query plan 1

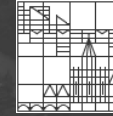


**push
selection
through
staircase
join**

Query plan 2

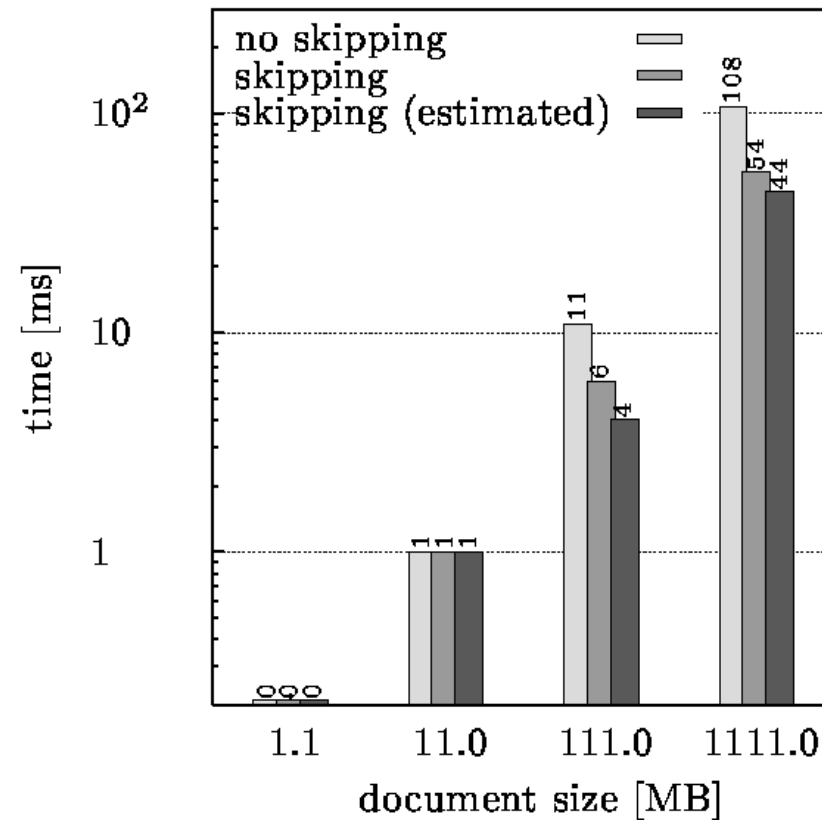
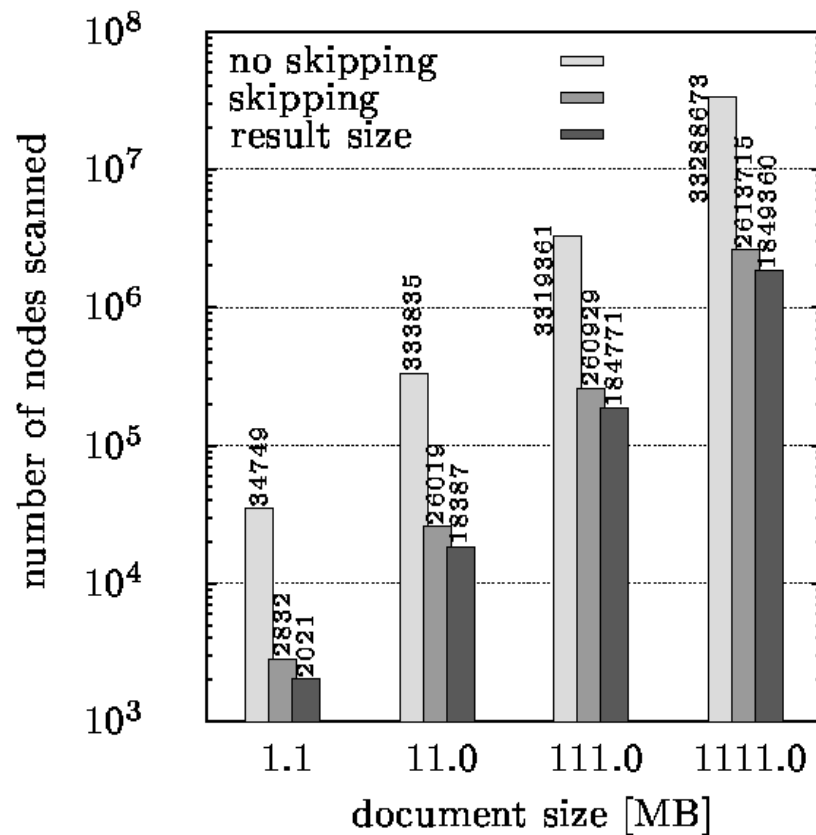
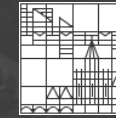


Query optimization



- How to choose among alternative query plans?
 - Cost estimation!
- Result size estimation:
 - Given context sequence CS
Approximate $|CS/axis|$ or $|nodetest(CS)|$
- Execution model of staircase join algorithm:
 - Context: main-memory access patterns
 - Sequential traversal of context sequence
 - Sequential traversal of document sequence with average-length skips
 - Sequential traversal of result sequence
 - Consider CPU cache misses as if all three happen in parallel

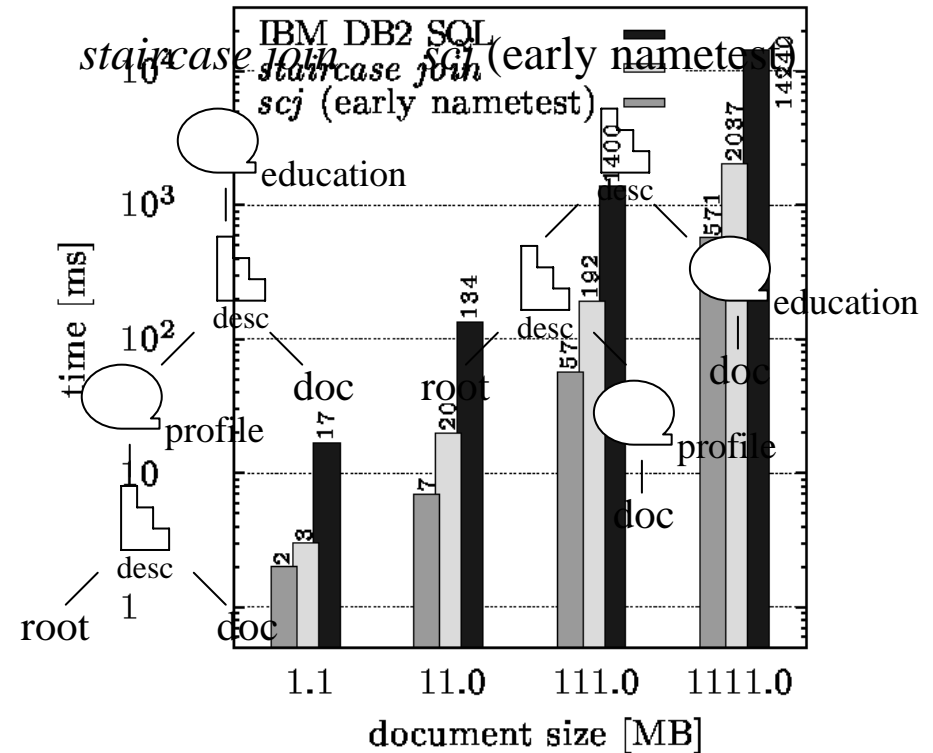
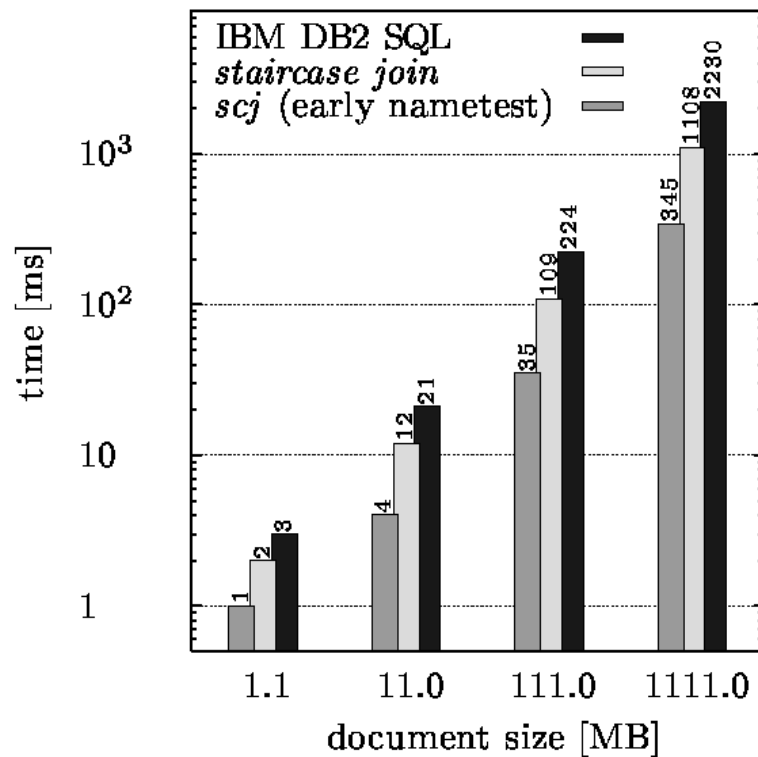
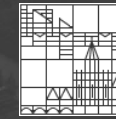
Experiments: effect of skipping



Query: `/descendant::profile/descendant::education`

Interm.results(1GB XMark) 47,015,212 127,984 1,849,360 63,793

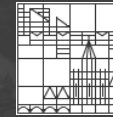
Experiments: Tree-unaware vs. Tree-aware



Query 1: /descendant::profile/descendant::education

Query 2: /descendant::increase/ancestor::bidder

Conclusions



- XPath accelerator
 - Complete relational representation of XML
- Staircase join
 - Encapsulates “tree knowledge” by using pre/post encoding, pruning, partitioning, and skipping
 - Local change to RDBMS kernel
 - Provides efficient evaluation of XPath axis steps

Compilation of XQuery core to relational algebra

(wednesday morning by Torsten Grust) →

Validation of encoded XML against schema definition

(wednesday morning by Torsten Grust) →

Staircase join ✓

XPath accelerator ✓

SQL, relational algebra

