UNIVERSITY of CALIFORNIA Riverside

Efficient Processing of XML Containment Queries using Partition-Based Schemes

---

Zografoula Vagena
MIRELLA M. MORO
Vassilis J. Tsotras

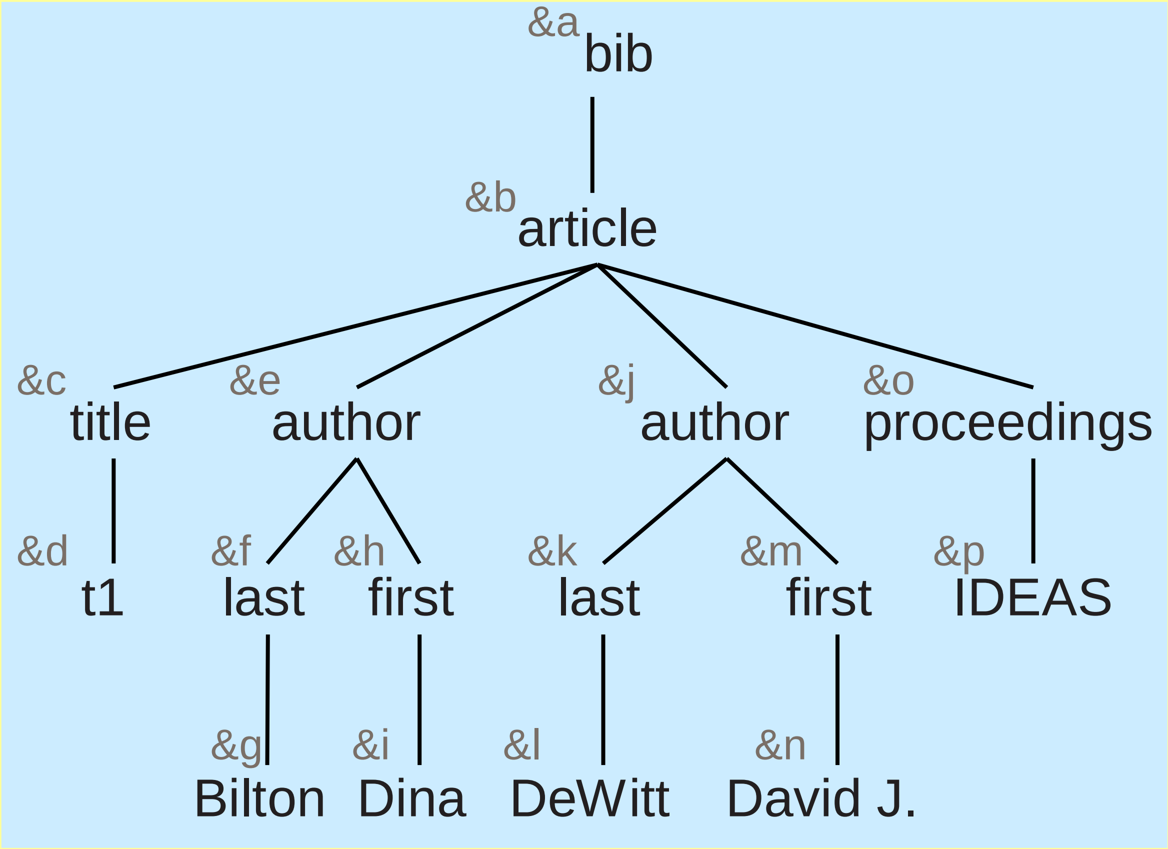
Research partially supported by CAPES and NSF IIS-0339032

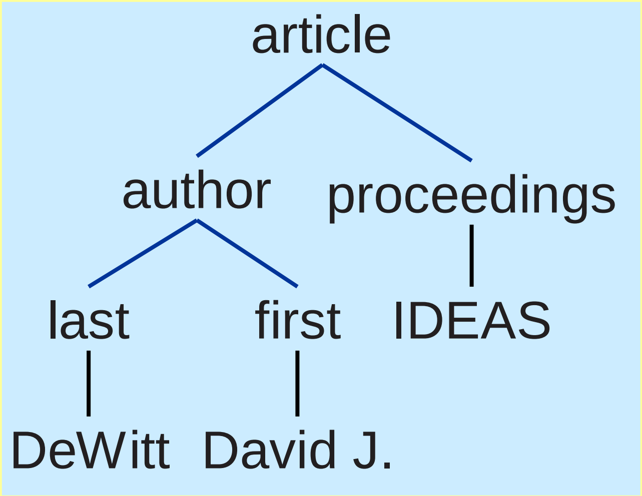
Outline

- Motivation
- Contributions
- PBiTree
- Our proposal
- Experimental evaluation
- Conclusions

Motivation

- XML as standard for e-commerce applications
  - Efficient storage, management, and retrieval
- Containment join
  - Given two sets of elements, return pairs of ancestor-descendant elements
- XML database
  - Forest of unranked, ordered, node-labeled trees
  - One tree = one document
  - Each node = element, attribute or value
  - Edges = immediate element-subelement or element-value relationships





//article [.author [@last=“DeWitt” and @first = “David J”]] //proceedings[./IDEAS]

Motivation

1. Navigation-based algorithms
    - One node at a time (Yfilter, ViST, …)
2. Set-based techniques
    - Range-based numbering scheme to identify relationships on input element sets (structural join, holistic twig joins, XR-tree…)

- Input accessed in particular order
    - Input is sorted or indexed
- Not necessarily available (e.g. intermediate)

Motivation

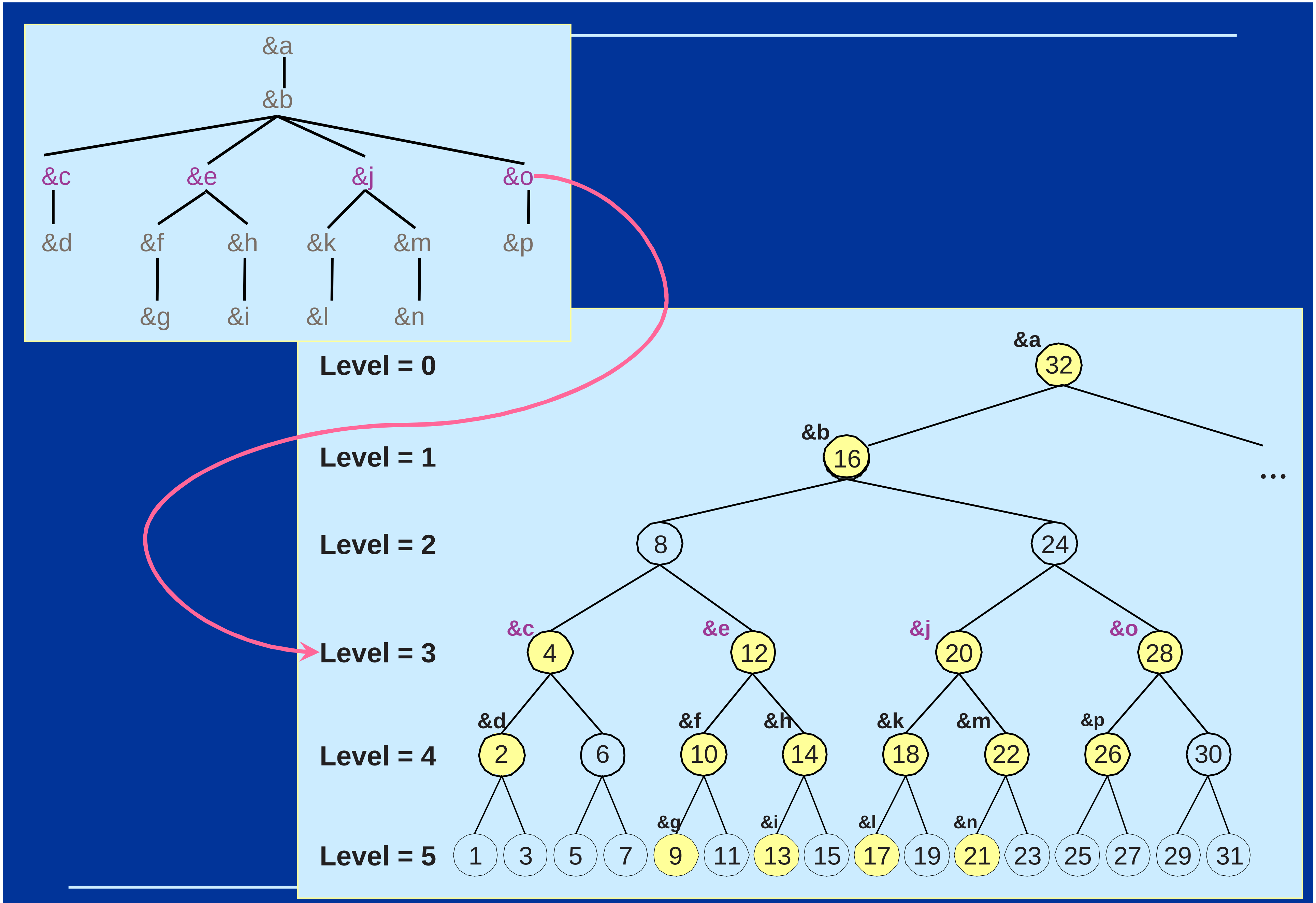
- No predefined access order
- Partition-based algorithms for processing containment join
- PBiTree encoding
    - No full use of the structural features
    - False positives (overhead)
    - Not scalable join algorithm

Contributions

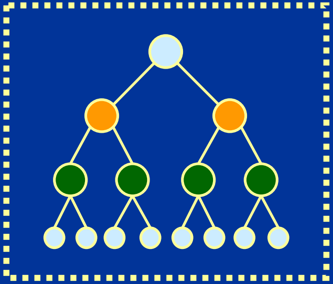
1. New scalable SAX-based mapping algorithm
2. Improved partition-based algorithm
    - Gracefully adaptable to large documents
    - Effects of partition overlaps relieved
3. More appropriate numbering scheme
4. Extensive experimental section

PBiTree

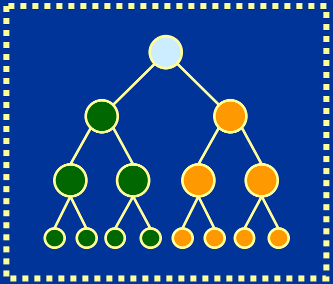
- Perfect binary tree, nodes with 2 numbers
  - Level within the tree
  - In-order traversal tag
- Tree binarization
  - Root, subtree s.t. children fit same level
- PBiTree not materialized, just numbers
- Checking ancestor-descendant: shifting and integer operators


&a
&b
&c
&e
&j
&o
&d
&f
&h
&k
&m
&p
&g
&i
&l
&n
Level = 0
Level = 1
Level = 2
Level = 3
Level = 4
Level = 5
&a 32
&b 16
…
8
24
&c 4
&e 12
&j 20
&o 28
&d 2
6
&f 10
&h 14
&k 18
&m 22
&p 26
30
1
3
5
7
&g 9
11
&i 13
15
&l 17
19
&n 21
23
25
27
29
31

PBiTree: Partitioning algorithms



1. Horizontal
   - According to the level
   - ✓ Specific (limited) scenarios, e.g. no recursion, equal #children
   - ✘ False positives, multiple scans, …



2. Vertical
   - According to residing path
   - ✘ No good use of size of available memory
   - ✘ Replication cost

Our Proposal

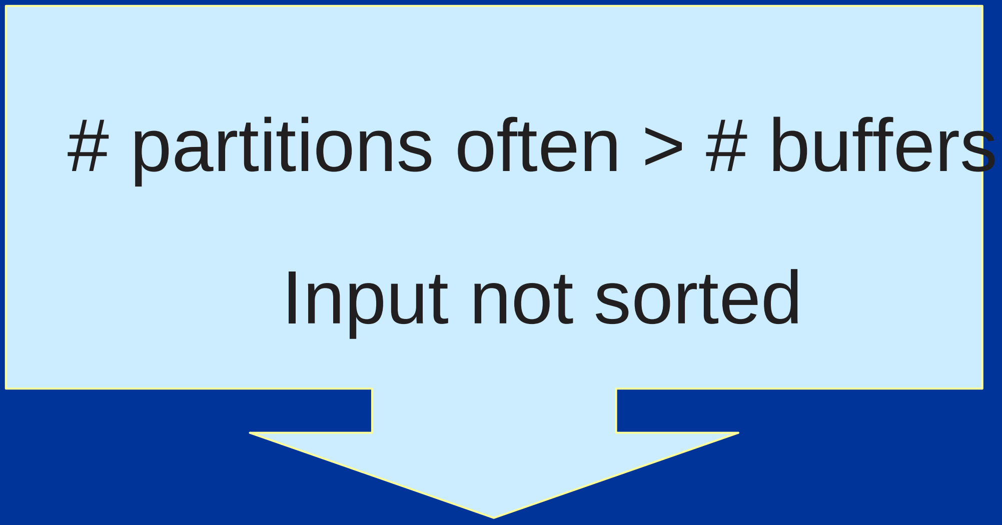
- Original mapping (binarization):
  - Height of PBiTree is known in advance
  - Number of children known before visiting node
  - Children placed to the first level can hold
  - Shortcoming: DOM interface: up to 5x document size larger

☀ New scalable SAX-based mapping algorithm

1. Compute number of children of each node (stack)
2. Find height of PBiTree
3. Identify level of each node and its code (stack for parent node)

Our Proposal

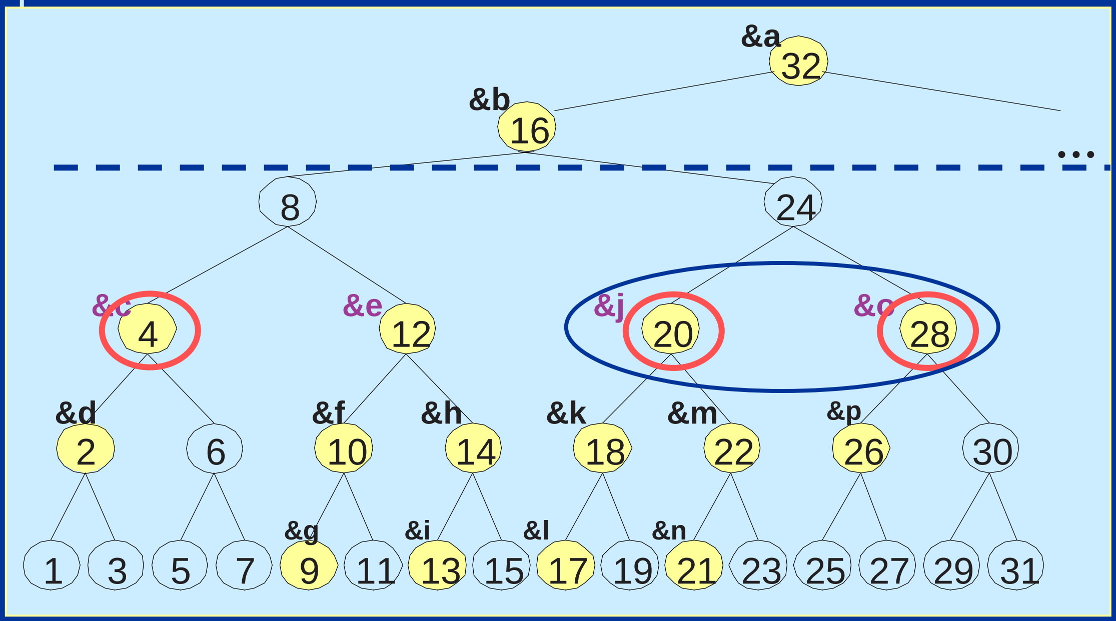
- Original vertical partitioning: Shortcoming 1
  - Each partition fits in memory

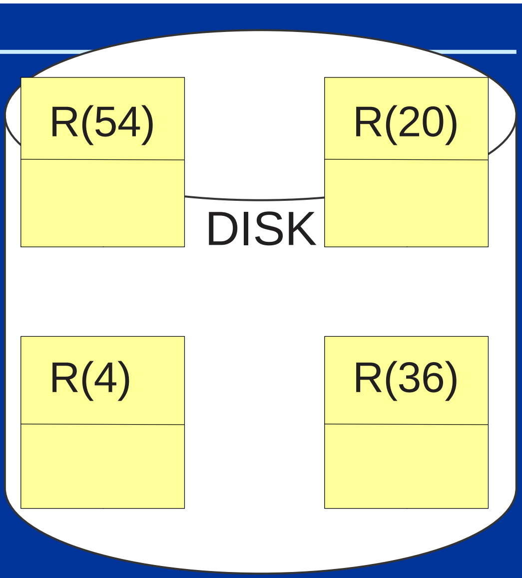


partitions often > # buffers

Input not sorted

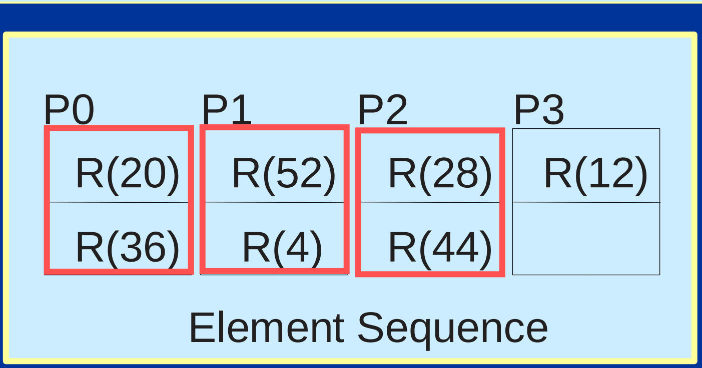
May cause overhead (as in next slide)





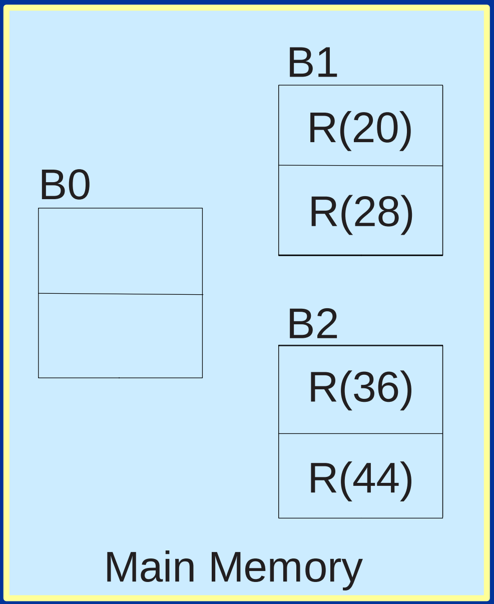
DISK

R(54)
R(20)
R(4)
R(36)



P0	P1	P2	P3
R(20)	R(52)	R(28)	R(12)
R(36)	R(4)	R(44)	

Element Sequence



B0

B1
R(20)
R(28)

B2
R(36)
R(44)

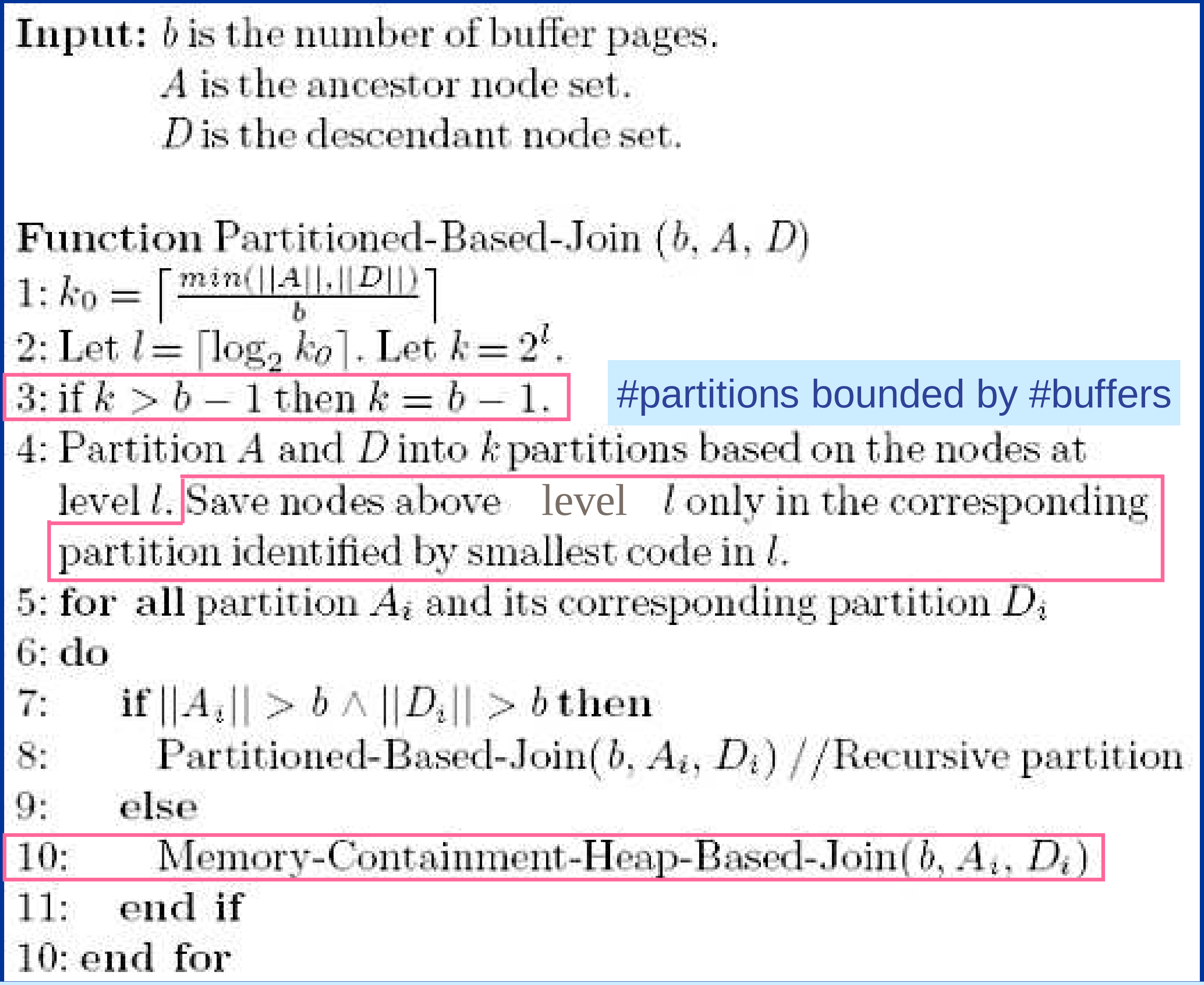
Main Memory

- Solution: # partitions <= # buffers
  - Buffer flushed out only when full
  - Partitions with larger size: repartitions

Our Proposal

- Shortcoming 2
  - Replicated elements
    - Elements above partition level are added to each partition they belong to (multiple times)
    - Size replication linear to the size of the elements that belong to more than one partition, creating space overhead
  - Descendants above partition level considered the same way as below level
- New Partition-Based Containment Join

Our Proposal



```
Input: $b$ is the number of buffer pages.
       $A$ is the ancestor node set.
       $D$ is the descendant node set.

Function Partitioned-Based-Join ($b$, $A$, $D$)
$k_0 = \lceil \frac{min(||A||,||D||)}{b} \rceil$
Let $l = \lceil \log_2 k_0 \rceil$. Let $k = 2^l$.
if $k > b - 1$ then $k = b - 1$.
Partition $A$ and $D$ into $k$ partitions based on the nodes at
  level $l$. Save nodes above level $l$ only in the corresponding
  partition identified by smallest code in $l$.
for all partition $A_i$ and its corresponding partition $D_i$
do
    if $||A_i|| > b \wedge ||D_i|| > b$ then
        Partitioned-Based-Join($b$, $A_i$, $D_i$) //Recursive partition
    else
        Memory-Containment-Heap-Based-Join($b$, $A_i$, $D_i$)
    end if
end for
```

#partitions bounded by #buffers

Ancestors saved once here. Then ancestors are cached to be used in next partitions.

One partition is probed by using a heap

Our Proposal

- Shortcoming 3: Virtual nodes
  - Whole levels not utilized
  - Element instances spread over different levels
    - More horizontal partitions
    - False positives, overhead of post-processing

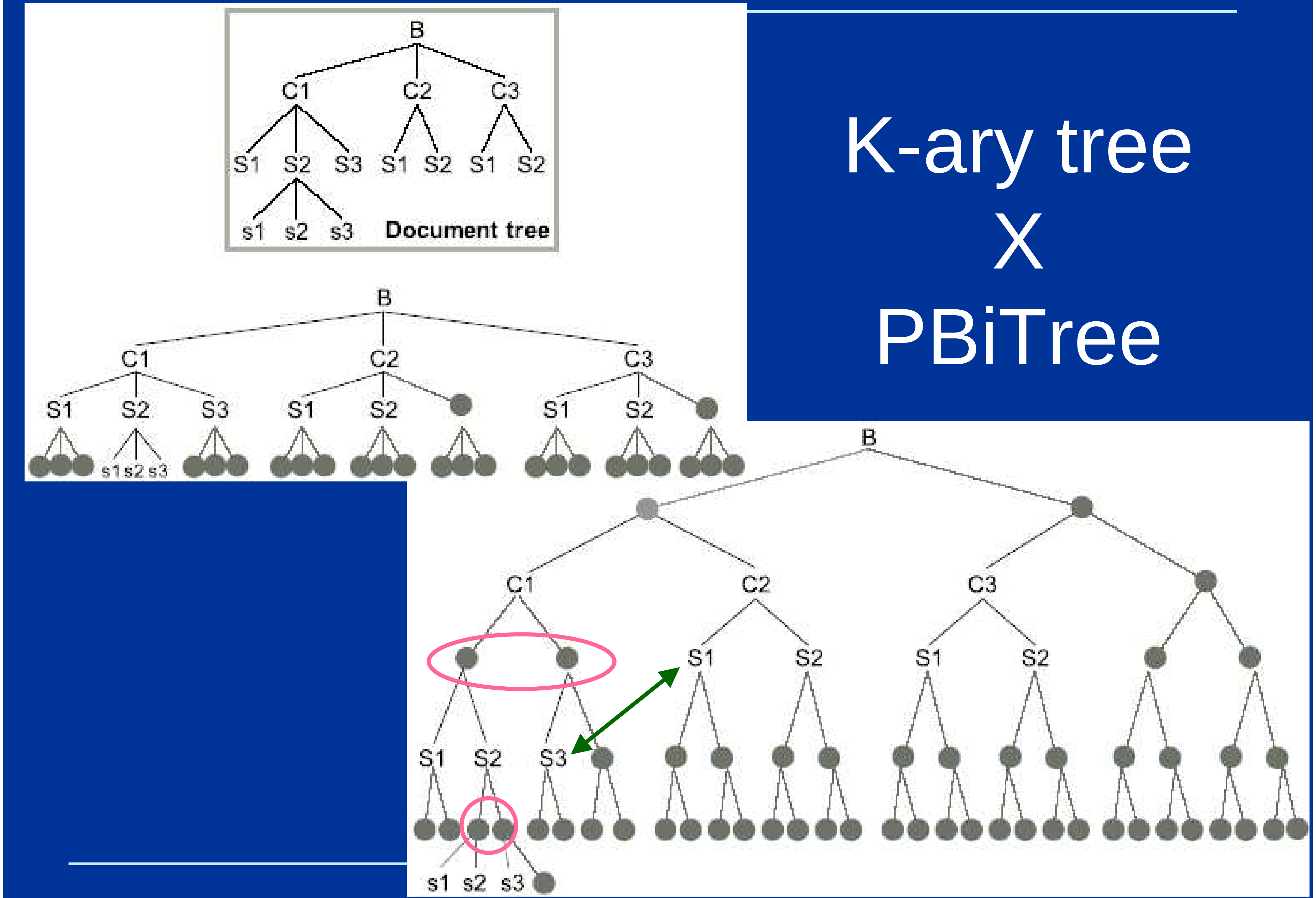
☀ New numbering scheme

  - *K-ary* tree, where k = largest number of children

  ✓ Similar functions to compare pairs

  ✓ More ancestors in the same level

K-ary tree
X
PBiTree



Experimental Evaluation

- Setup
  - Native storage manager implementing LRU
  - 2.6GHz Pentium 4 with 512Mb memory
  - Page size 8Kb, 100 buffers
  - Repeat each algorithm 10 times, take average t
- Synthetic and benchmark data

Experimental Evaluation

- VPJ: original vertical partition join
- XPJ: proposed Xml-data partition join

9 100% selectivity, no recursive elements
- XPJ adapts gracefully

10 More replicated elements over partitions
- XPJ smaller partitions
- No descendant without ancestor

Varying the data size

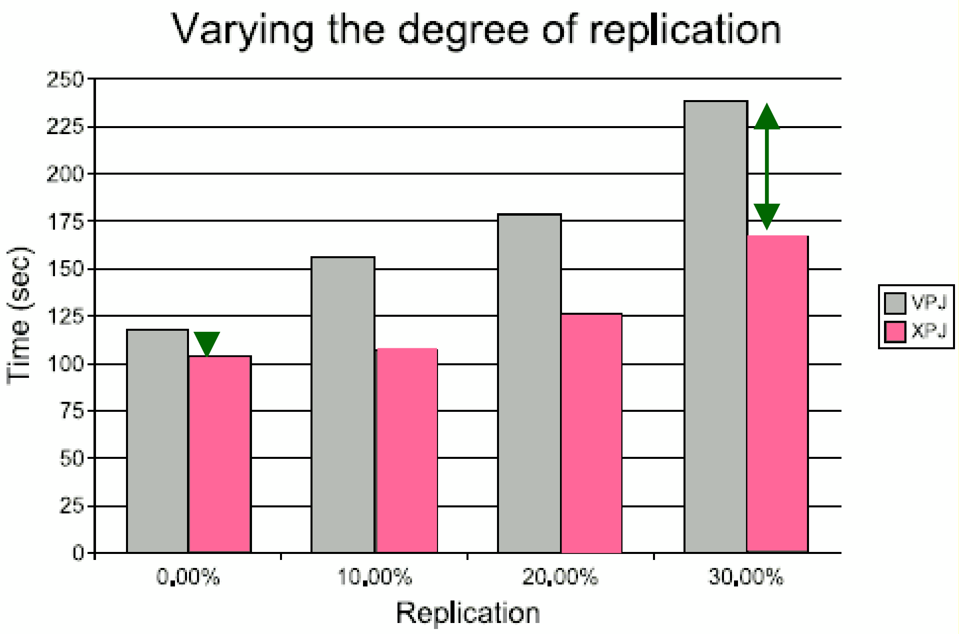


Figure 10. Performance of the partition-based algorithms with varying degree of replication

Experimental Evaluation

- Structural join algorithm + XB-tree
  - Discards both ancestor and descendant elements not in result
  - Replication not issue

11. XBJ better sorted XPJ better unsorted
12. 1 Gb XMark data

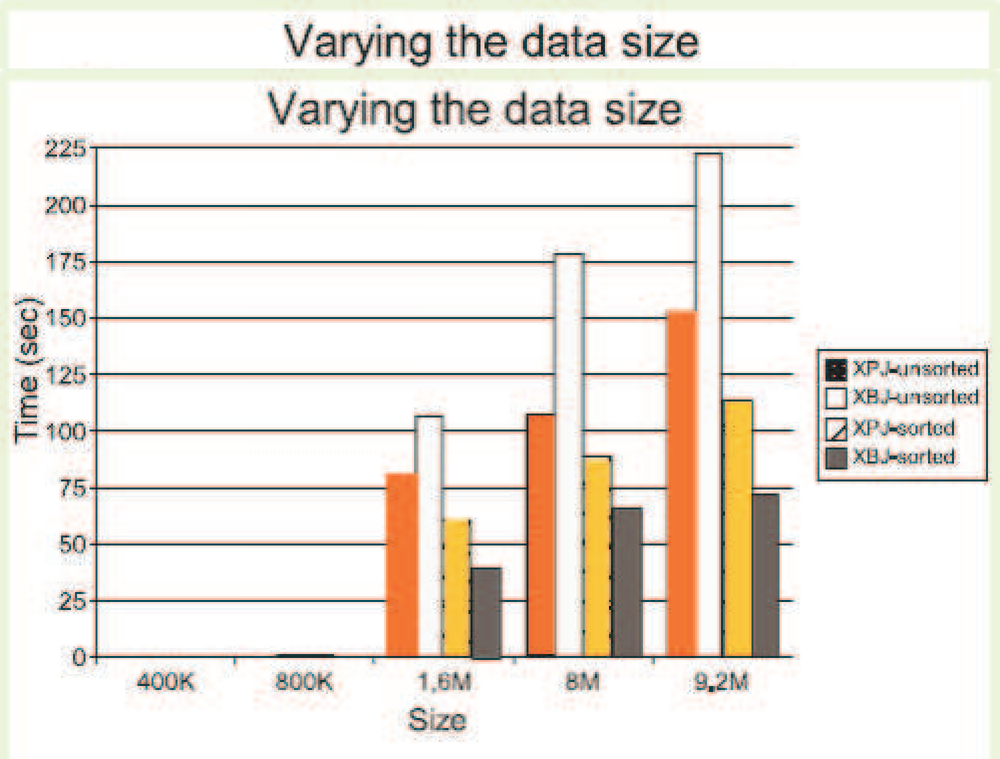


Figure 12. Partition-against set-based algorithms with varying data sizes (XMark data)

Conclusions

- Improved partition based algorithm (XPJ)
- Processing containment joins
- Data not clustered, ordered, or indexed
- Superior performance under these conditions
- Intermediate results

Questions