Efficient Structural Joins on Indexed XML Documents

Shu-Yao Chien
Zografoula Vagena
Donghui Zhang
Vassilis J. Tsotras
Carlo Zaniolo

Presented By: Samuel R. Collins
scollins@cc.gatech.edu

Georgia Tech College of Computing

What’s it all about?

- This paper proposes efficient structural join algorithms in the presence of tag indices
  - B+-tree based structural join algorithms
  - Introduce the utilization of sibling pointers to improve performance
  - Comparison with R-tree algorithm presented

Agenda

- Querying XML Documents
- Previous Work
  - Node Numbering Schemes
  - Structural Joins
  - XML Indexing
- Structural Joins Using B+-Trees
- Performance Analysis
- Summary

Querying XML Documents

- Combine selections on element contents and structural relationships (path expressions) between tagged elements
- Example Query:

section[title=“overview”]//figure[caption=“R-tree”]

finds all figures with caption=“R-tree” under sections whose title is “overview”

Querying XML Documents

- Traditional indexing schemes, such as B+-trees, can be extended to support value based queries
- Path expression queries pose a much harder problem
  - Require computation of **structural joins**

Querying XML Documents

- **Structural joins** are used to find all pairs of elements satisfying the primitive structural relationships specified in a query
  - ***parent-child*** relationship
    - Example: **section/figure**
  - ***ancestor-descendant*** relationship
    - Example: **section//figure**

Querying XML Documents

- Efficient support for structural joins is therefore needed for efficient implementation of XML queries
  - Quickly determine structural relationship among any pair of tree nodes
  - Efficiently find all occurrences of a structural relationship

Previous Work

- Numbering Schemes
  - Allows faster determination of structural relationships if embedded on the document's tree
  - One approach assigns ***preorder*** and ***postorder*** ranks as well as ***level*** in the XML tree
    - Affected by document updates: Node ranks change when inserting and deleting nodes
  - Another approach assigns ***(start, end)*** interval
    - Durable

Durable Numbering Scheme

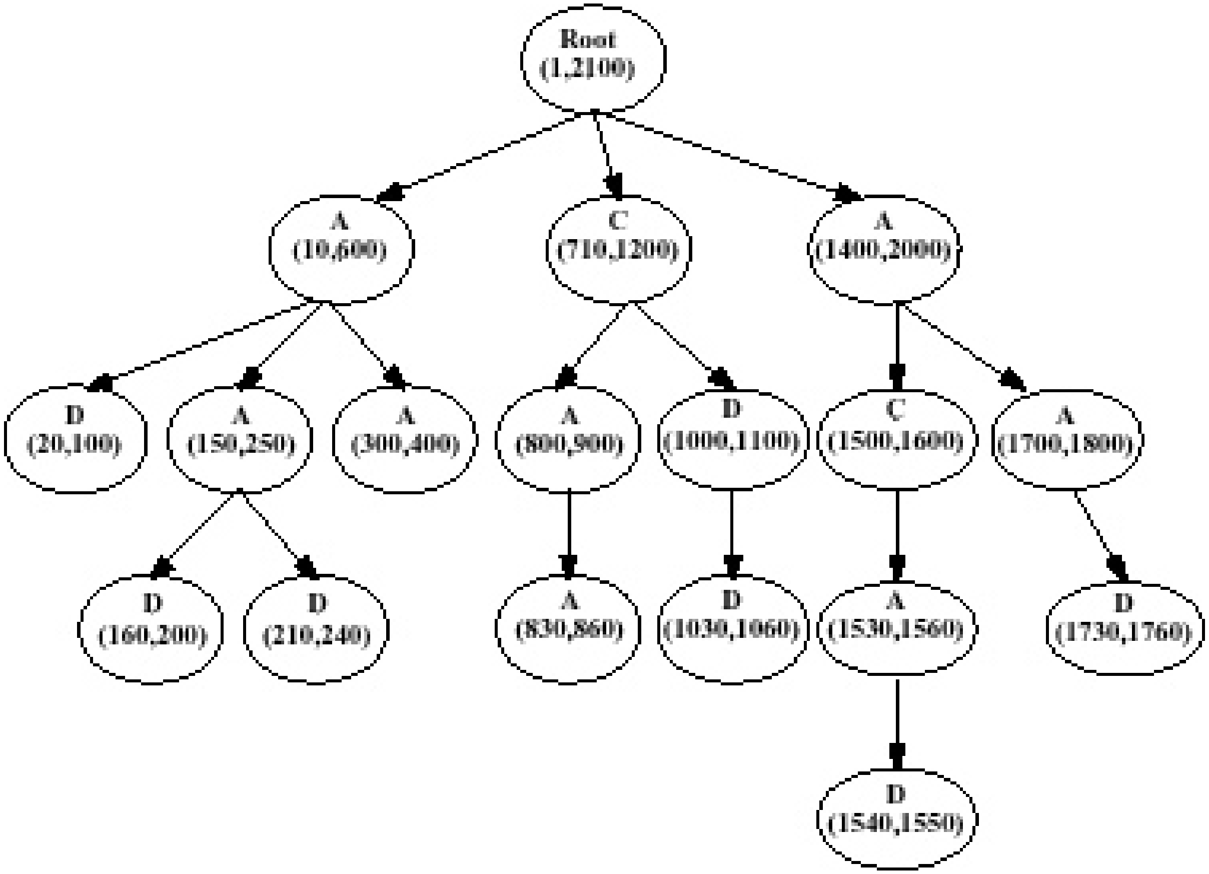


Figure 1: A sample XML document.

Previous Work

- Structural Joins
  - Can take advantage of numbering schemes to determine all pairs of ancestor-descendants
  - Considered core operations in optimizing XML queries
  - Various techniques have been proposed
    - Relational DBMS
    - Native XML Query Engines
    - **Stack-Tree-Desc** algorithm represents state-of-the-art in structural joins

Previous Work

- Indexing XML Data
  - Techniques have been proposed that do not facilitate a numbering scheme. These works create labeled directed graphs.
    - Unlike a schema they are not static, and thus may change with an update
  - Recent proposed node numbering schemes use **B+-trees** and **R-trees** to capture XML document structures

Structural Joins (B+-Trees)

- Authors propose structural join algorithm using B+-tree and Node Numbering
  - Consider a single large document
  - Concentrate on the ***ancestor-dependant*** join
  - Assume that a separate index is used to cluster elements from the same tag
  - In practice, multiple indices can be combined by adding the tag name in the search key

    (tag,start), See figure 2.

Structural Joins (B+-Trees)

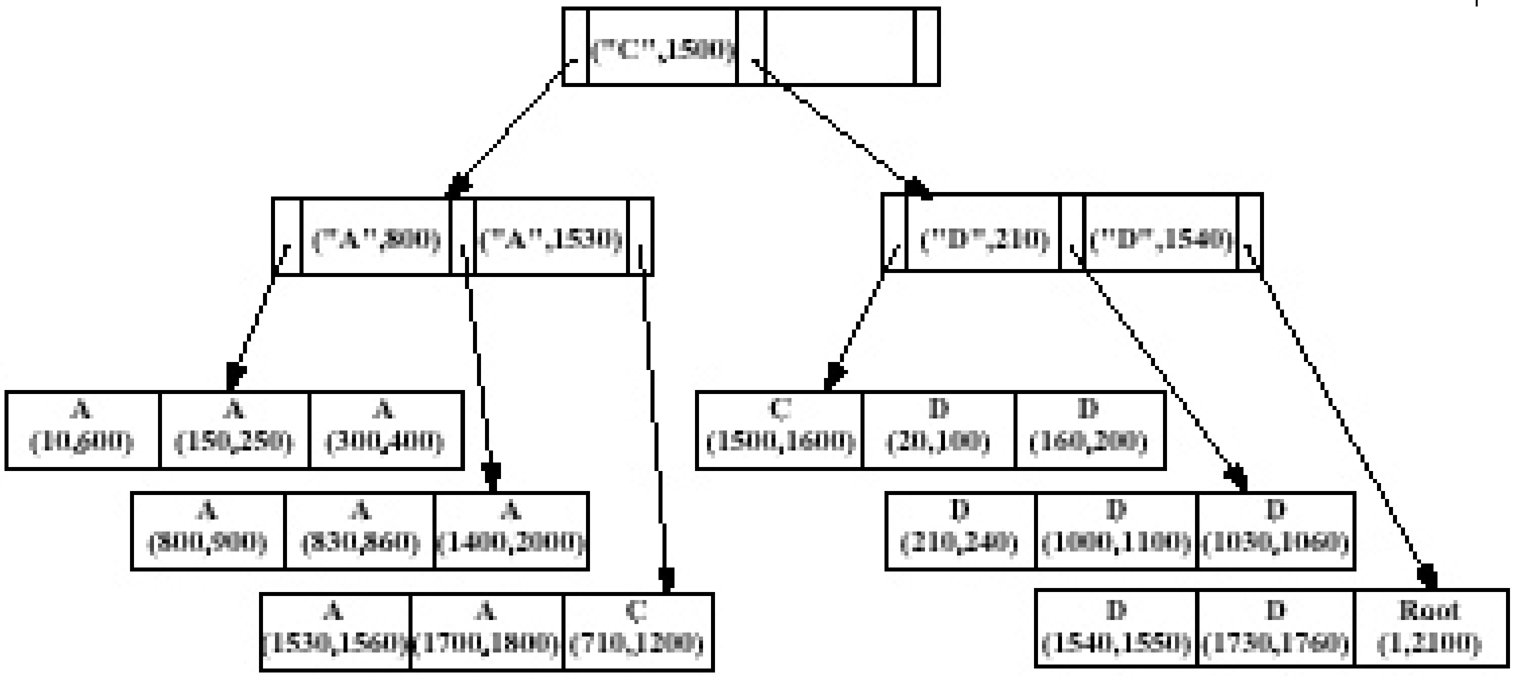
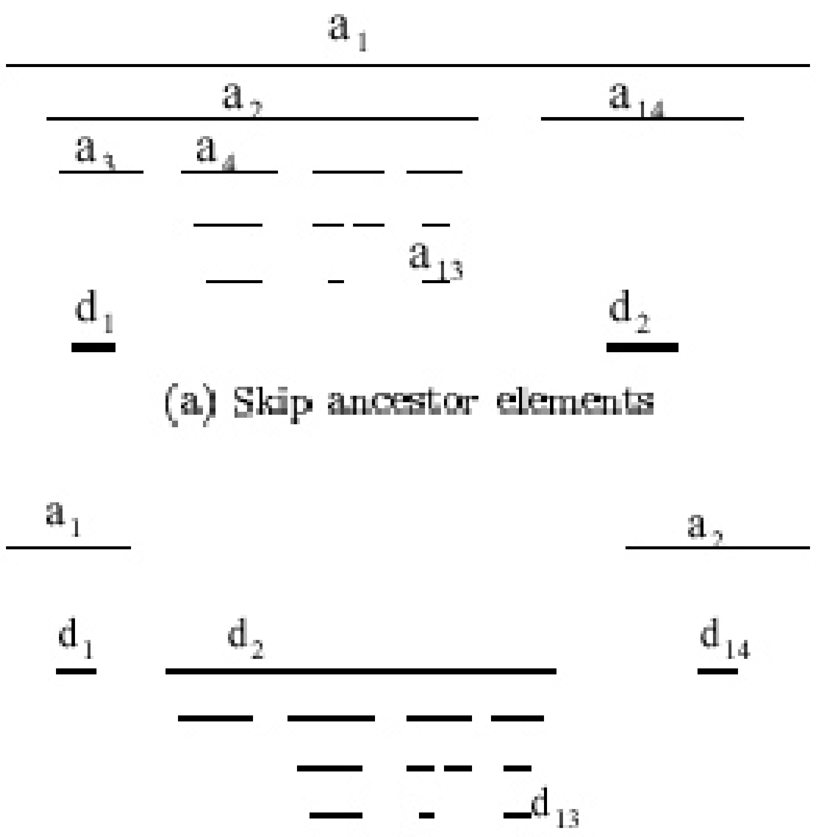


Figure 2: The B+-tree corresponding to the XML document of figure 1.

Georgia Tech College of Computing

Why Index for Structural Joins?

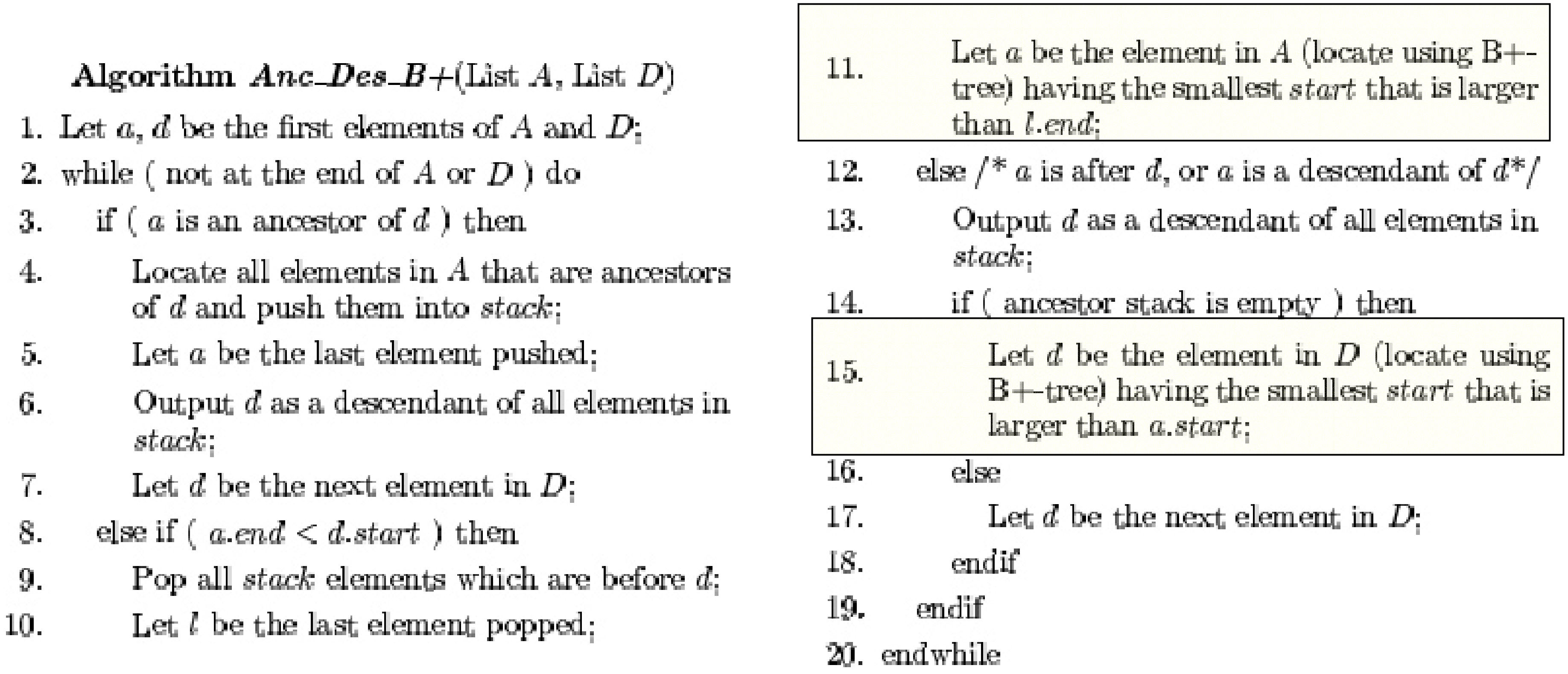


(a) Skip ancestor elements

(b) Skip descendant elements

Figure 3: Motivation for using the B+-tree index.

Algorithm *Anc_Desc_B+*



```
Algorithm $Anc\_Des\_B+$(List $A$, List $D$)
Let $a$, $d$ be the first elements of $A$ and $D$;
while ( not at the end of $A$ or $D$ ) do
    if ( $a$ is an ancestor of $d$ ) then
        Locate all elements in $A$ that are ancestors of $d$ and push them into $stack$;
        Let $a$ be the last element pushed;
        Output $d$ as a descendant of all elements in $stack$;
        Let $d$ be the next element in $D$;
    else if ( $a.end < d.start$ ) then
        Pop all $stack$ elements which are before $d$;
        Let $l$ be the last element popped;
        Let $a$ be the element in $A$ (locate using B+-tree) having the smallest $start$ that is larger than $l.end$;
    else /* $a$ is after $d$, or $a$ is a descendant of $d$*/
        Output $d$ as a descendant of all elements in $stack$;
        if ( ancestor stack is empty ) then
            Let $d$ be the element in $D$ (locate using B+-tree) having the smallest $start$ that is larger than $a.start$;
        else
            Let $d$ be the next element in $D$;
        endif
    endif
endwhile
```

Embedding Containment Forest

- Enhancement to B+-Tree to improve performance
- Each element corresponds to a node in the structure and is linked to other elements from same tag
  - Parent, first-child, sibling pointers

Embedding Containment Forest

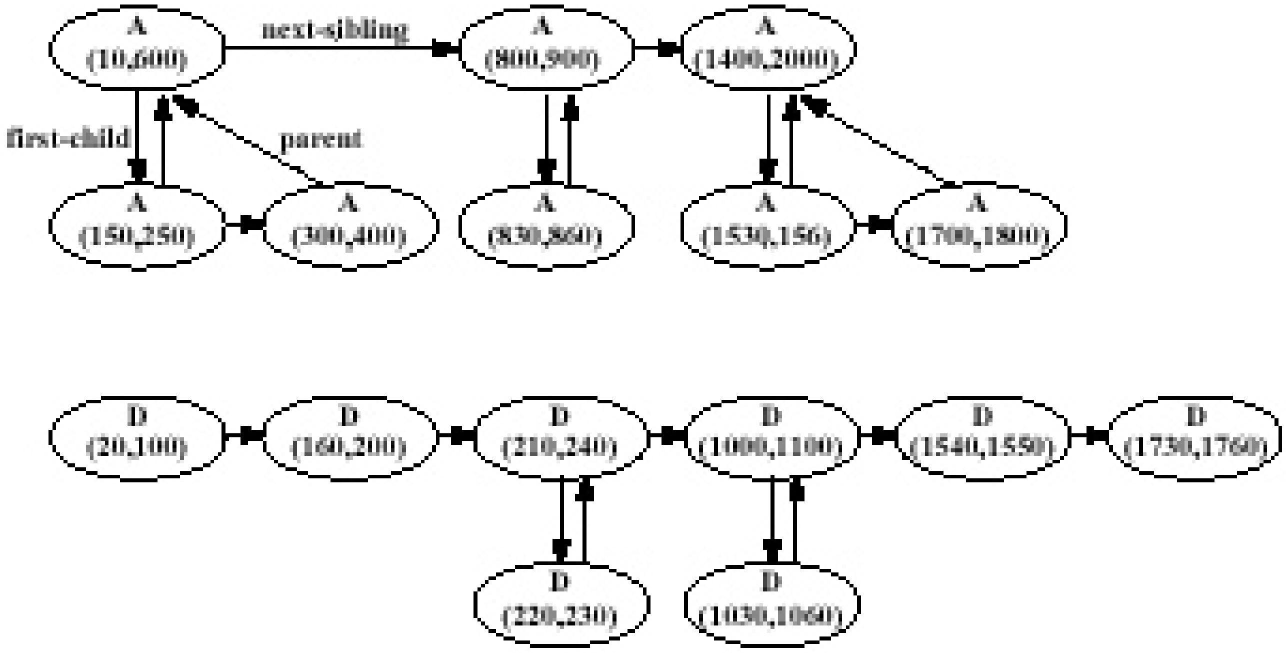
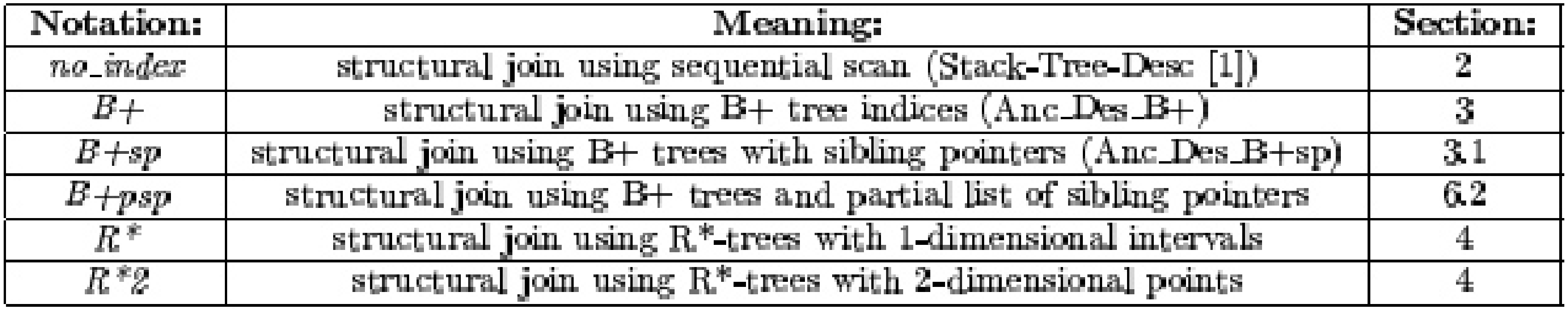


Figure 5: C-forests on tags *A* and *D* for the XML document of Figure 1.

Embedding Containment Forest

- Properties
  - The (start,end) interval of each node contains all intervals in its subtree
  - Start numbers in the forest follow a preorder traversal
  - The start (end) numbers of sibling nodes are in increasing order
- Embedding C-forest for a given tag can be accomplished by adding the C-forest parent and next-sibiling ponters amongst leaf records of B+-tree
  - Improves algorith. B+-tree traversal is avoided

Performance Analysis

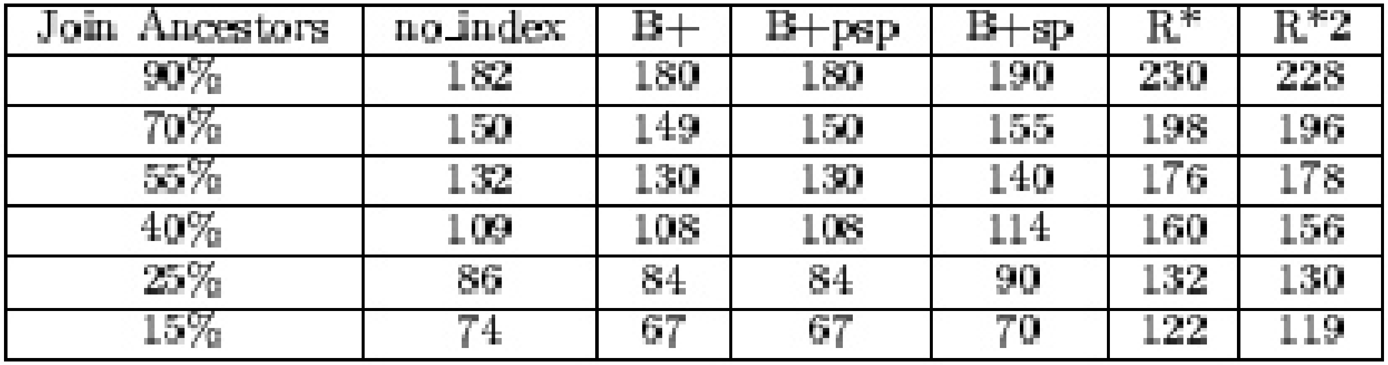


Notation:	Meaning:	Section:
no_index	structural join using sequential scan (Stack-Tree-Desc [1])	2
B+	structural join using B+ tree indices (Anc_Des_B+)	3
B+sp	structural join using B+ trees with sibling pointers (Anc_Des_B+sp)	3.1
B+psp	structural join using B+ trees and partial list of sibling pointers	6.2
*R**	structural join using R*-trees with 1-dimensional intervals	4
*R*2*	structural join using R*-trees with 2-dimensional points	4

Table 1: Implemented Algorithms.

- CPU time and number of I/0’s are used to measure performance

Performance Analysis



Join Ancestors	no_index	B+	B+psp	B+sp	R*	R*2
90%	182	180	180	190	230	228
70%	150	149	150	155	198	196
55%	132	130	130	140	176	178
40%	109	108	108	114	160	156
25%	86	84	84	90	132	130
15%	74	67	67	70	122	119

Table 2: Effect of skipping only ancestors in join performance.

- All the algorithms except, R-tree based ones, perform similarly

Performance Analysis

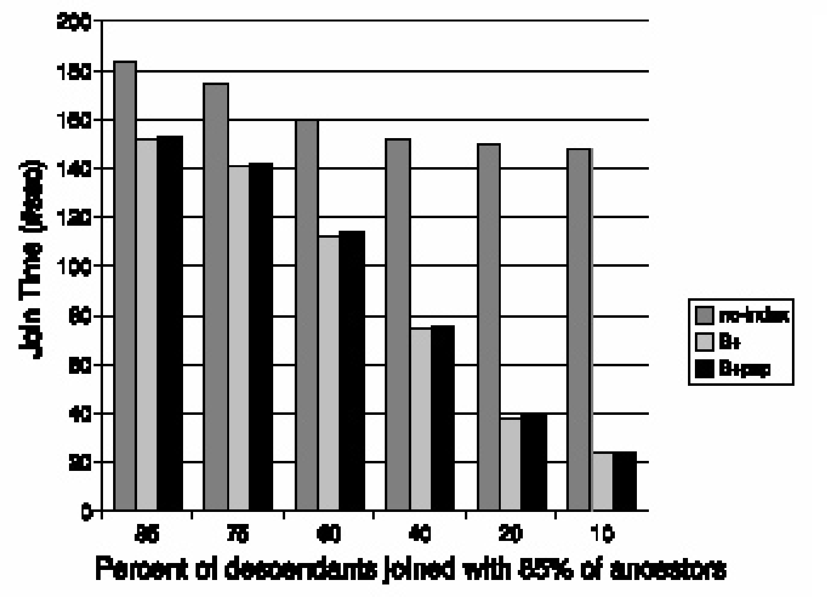


Figure 10: Effect of skipping only descendants.

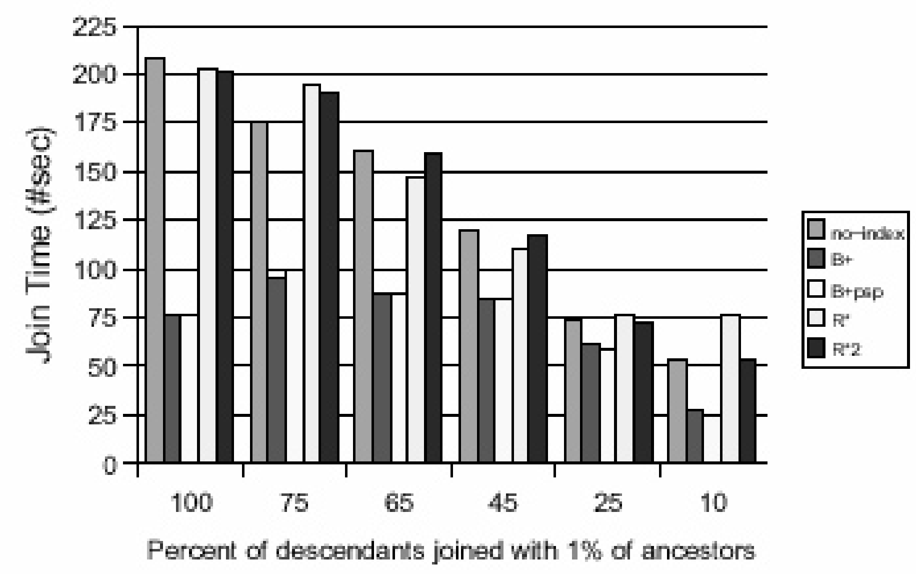


Figure 11: Effect of skipping both ancestors and descendants.

Summary

- Indexing schemes can be enhanced to support their structural join algorithm
- Indexing schemes can be made durable, thus support updateson XML documents
- Their indexed algorithms are more robust than the state-of-the-art algorithms