Join Algorithms for XML Query Processing

Structural Joins: A Primitive for Efficient XML Query Pattern Matching [ICDE 2002]

Holistic Twig Joins: Optimal XML Pattern Matching [ACM SIGMOD 2002]

Presented by Juyong Jin

Overview

- Two approaches about indexing and query processing
  - Use specialized methods for *semi-structured* data
  - Represent XML data in relational tables
- Join algorithms for XML Query Processing
  - Structural Joins
    - The tree-merge join, The stack-tree join
  - Holistic Twig Joins
    - The path join and twig join (extend the stack-tree join)

How should we organize and query XML data?

- Solution 1
  - Use specialized storage methods, and query evaluation techniques for semi-structured data.
    - 1-index, A(k) index, APEX, etc.

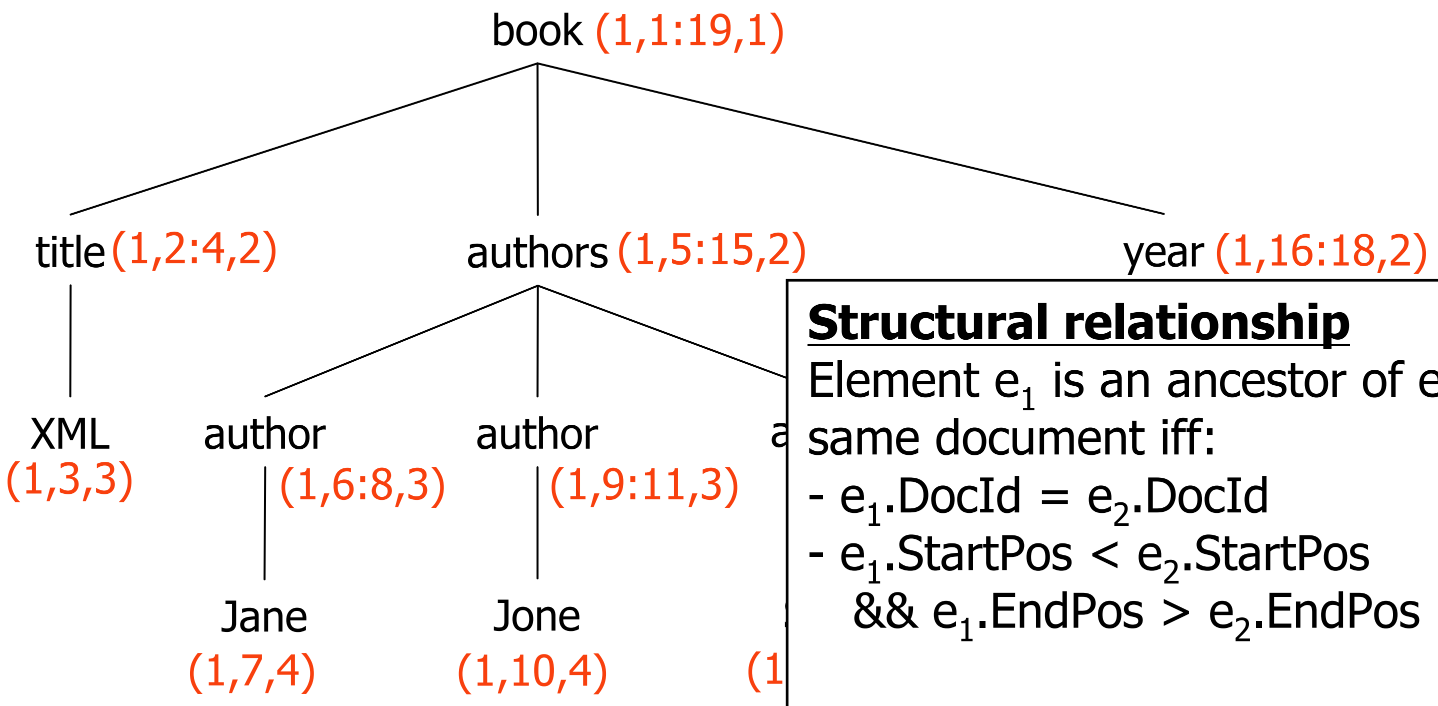
- Solution 2
  - Represent XML data in relational tables
    - (DocId, StartPos, EndPos, LevelNum) representation
    - decompose XML data into tables and use join algorithms to answer queries.

Storing and indexing XML data in relational databases

- Decompose the structural information into tables and use them to answer queries
  - off-the-shelf query processing
  - reduce the volume of accessed data
  - mature relational DB technology
    - optimization techniques
- Index the elements and text of the XML data by their position
  - (DocId, StartPos : EndPos, LevelNum) encoding
  - **expensive joins during query processing**

Encoding elements and text based on their positions

- **(DocId, StartPos : EndPos, LevelNum) representation**



<u>Structural relationship</u>

Element e_1 is an ancestor of element e_2 in the same document iff:

- e_1.DocId = e_2.DocId
- e_1.StartPos < e_2.StartPos && e_1.EndPos > e_2.EndPos

If the above hold and, in addition, e_1.LevelNum+1 = e_2.LevelNum, then e_1 is the parent of e_2

Answering queries using the encoding

- The encoding is used to index the position of each element and text
- Example of database mapping (single edge table)
  - Each table is sorted by (DocId, StartPos)

book table

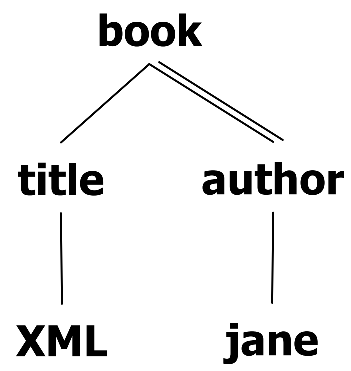
DocId	StartPos	EndPos	LevelNum
1	1	19	1

author table

DocId	StartPos	EndPos	LevelNum
1	6	8	3
1	9	11	3
1	12	14	3

Answering queries using the encoding (cont'd)

- The query is broken into binary parent-child or ancestor-descendent relationships
- Example of XML Query
  - book[title='XML']//author[.='jane']
  - Broken to:
    - book/title
    - title/XML
    - book//author
    - author/jane



- Each binary query is executed as a join, and their results are "stitched" together

How to process the binary joins

- The “heart” of XML query processing is the algorithm that joins the elements table to retrieve the result for each individual query component
- Structural Joins [ICDE 2002]
  - The tree-merge join algorithm
  - The stack-tree join algorithm

The tree-merge join algorithm

- Query: A//D
  - Let AList, DList be the lists of each element
    - e.g., paper//author
    - AList = { APEX, XTRACT, …}, DList = { Shim, Min, …}
- Extension of relational merge joins with the multiple inequality conditions

AList

1	23	34	2
1	25	33	3
1	45	56	2
2	15	28	3
2	45	57	4

DList

1	30	32	4
1	47	49	5
1	68	72	6
2	17	18	5
2	20	22	5

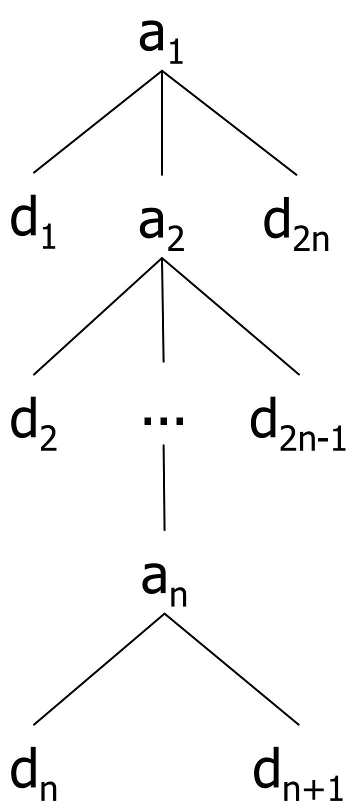
Analysis of the tree-merge join algorithm

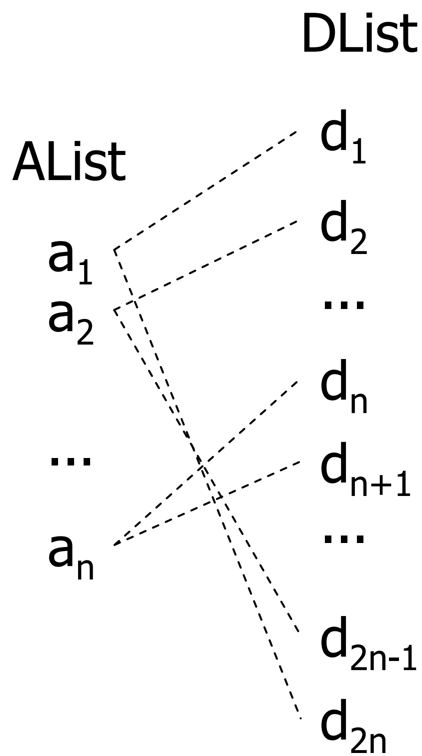
- Does NOT guarantee O(|AList| + |DList|)
  - Buffer is not considered for convenience

- O(|AList|+|DList|)
  - where no two nodes in AList are themselves related by an ancestor-descendant relationship
- O(|AList|*|DList|)
  - where multiple nodes in AList are themselves related by an ancestor-descendant relationship

Worst case for the tree-merge join algorithm

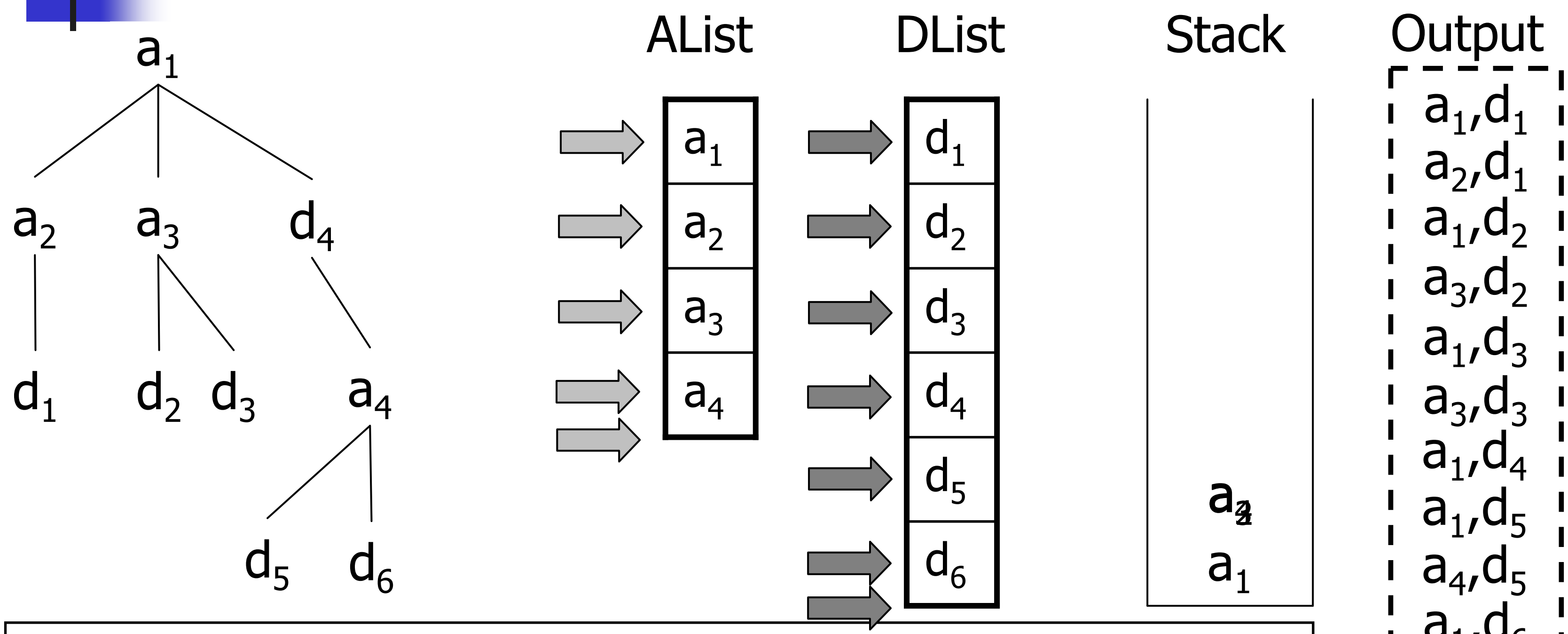
- Example Query: A/D





The tree-merge join algorithm does not have worst-case time complexity linear.

The stack-tree join algorithm



The stack always has a sequence of ancestor nodes

1. If current AList & DList are not a descendant of the current top of the stack → **Pop**
2. Else if current AList is an ancestor of the current DList → **Push & advance AList**
3. Else → **Output & advance DList**

Analysis of the stack-tree join algorithm and …

- Guarantee O(|AList| + |DList|)
  - better worst-case complexity than the tree-merge join algorithm

- Limitation of the binary join algorithms
  - If a query is complex (contains many binary relationship), intermediate result sizes can get large.
    - even when the input and output sizes are more manageable

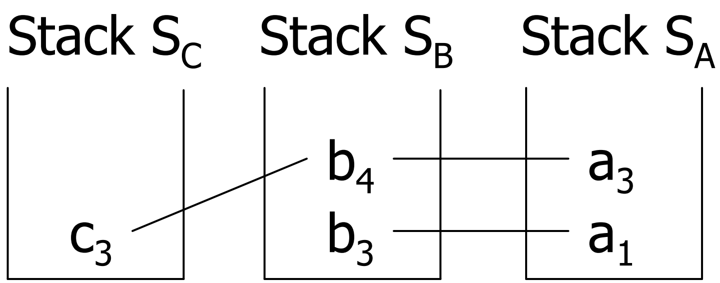
Extension of the stack-tree join algorithm

- Holistic Twig Joins [SIGMOD 2002]
  - The path join and twig join algorithms
    - extend the basic stack-join algorithm
  - Multiple stacks are used to avoid merging the intermediate results.
  - The path join algorithm for path queries only
    - e.g., book//author//name
  - The twig join algorithm for branching expressions
    - e.g., book[title='XML']//author[.='jane']

Stack encoding and query results

- Stack encoding and query results

  Query: a//b//c



Query results

a_3, b_4, c_3

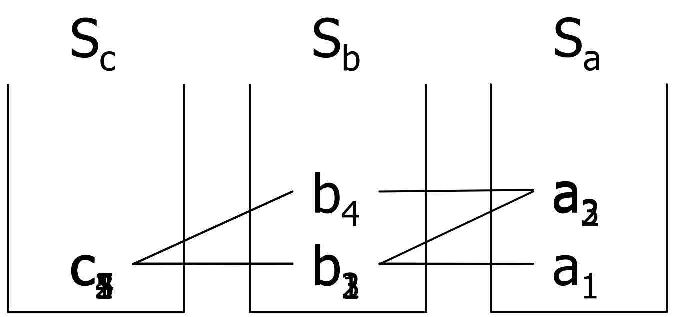
a_1, b_4, c_3

a_1, b_3, a_3

- Key idea
  - Repeatedly construct stack encoding of partial and total answers to the query path pattern
  - In the constructing process, remove partial answers from the stacks that cannot be extended to total answers

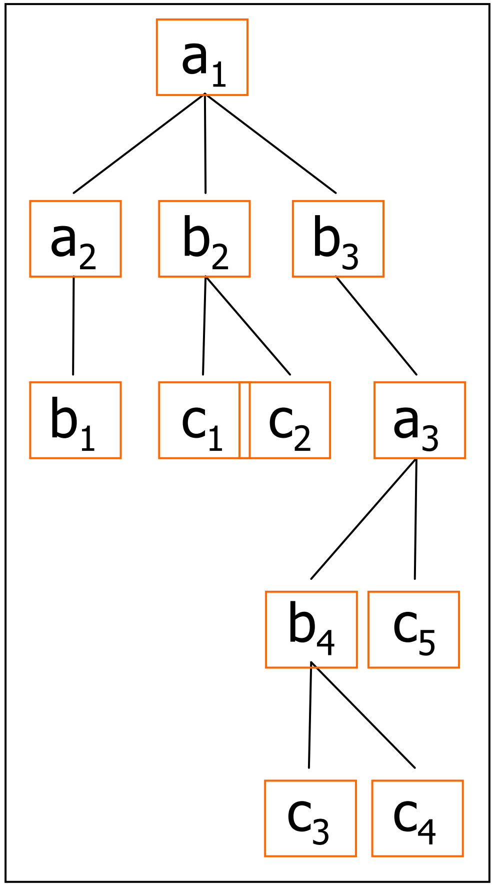
The path join algorithm

- Example query : a//b//c



Output

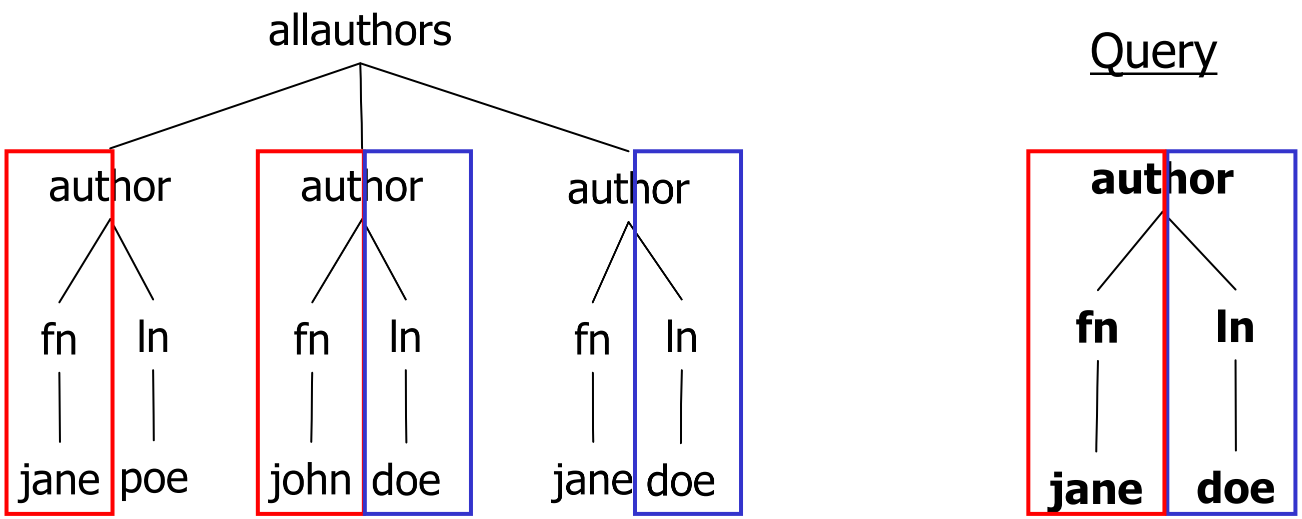
c_1, b_2, a_1
c_2, b_2, a_1
c_3, b_4, a_3
c_3, b_4, a_1
c_3, b_3, a_1
c_4, b_4, a_3
c_4, b_4, a_1
c_4, b_3, a_1
c_5, b_3, a_1



1. If current element is not a descendent of my stack→ **Pop from S_{me}**
2. **Push to $S_{current\ element}$**
3. If current element is a leaf → **Output and Pop from S_{leaf}**

Limitation of the path join

- If a query is a *twig* of multiple paths
  - The path join may decompose the twig into multiple root-to-leaf pattern.
  - Many intermediate results may not be part of any final answer.



The twig join algorithm

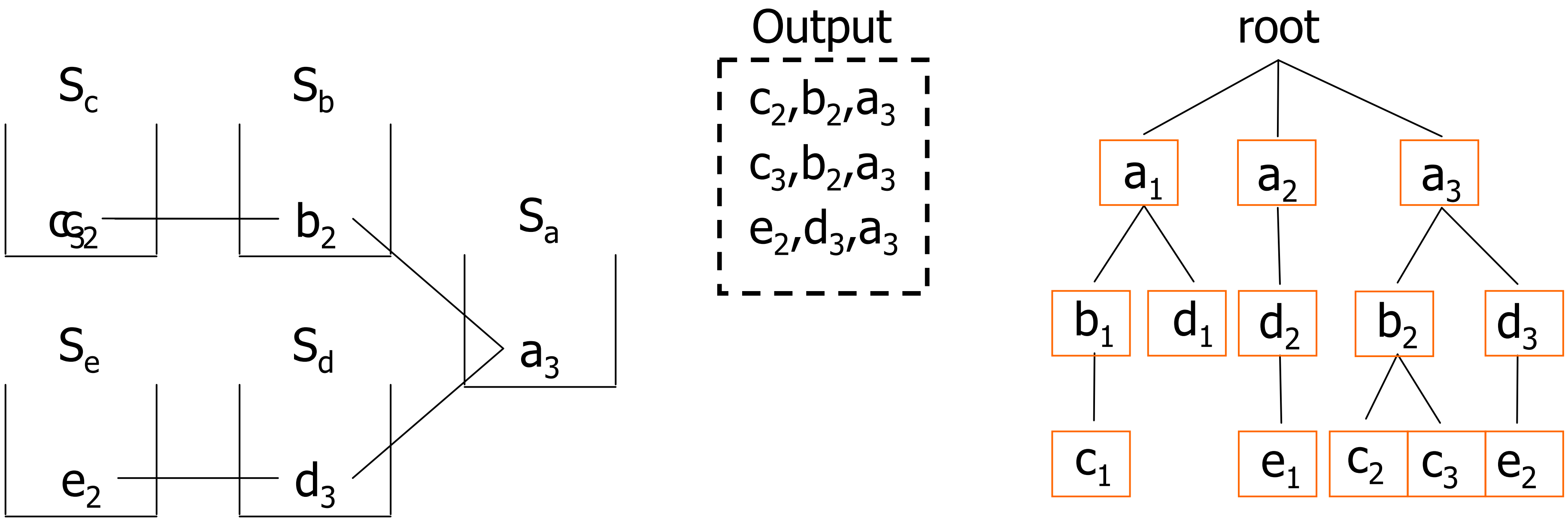
- the twig join applies multiple path-join at the same time.
- *Key difference* between the path join and twig join
  - When a element is pushed on its stack, it should have all the descendent elements satisfying the query.
    - A element which can not be a final result should not be pushed on the stack
  - No intermediate solution is large than the final answer

    → **optimal in the size of intermediate results**

Each individual root-to-leaf path is *guaranteed* to be merge-joinable with at least one of the other root-to-leaf paths.

The twig join algorithm (cont'd)

- Example query: a[//b//c][//d//e]



Experimental Evaluation

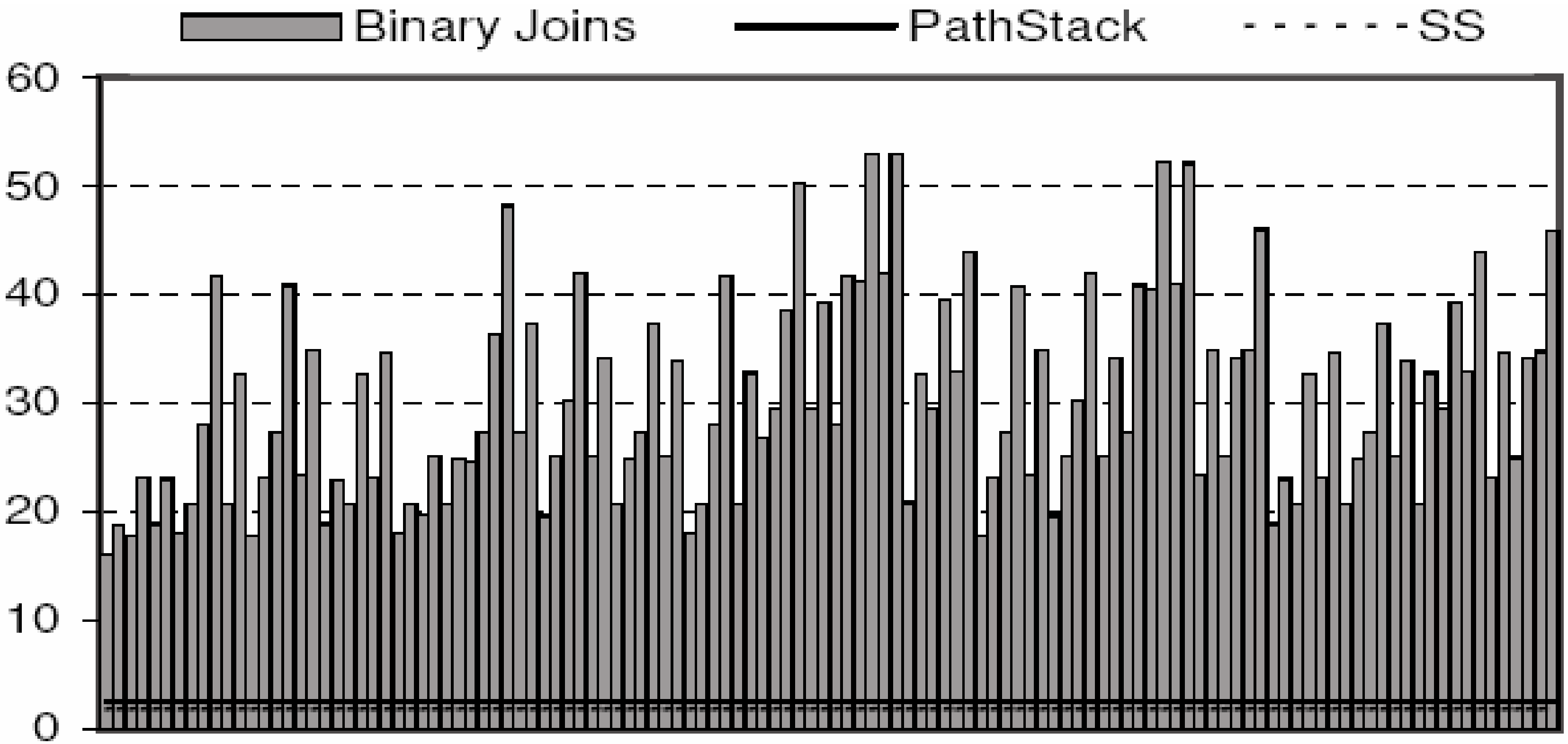
- Experimental Setting
  - PIII 550Mhz, RAM 768MB, disk 2GB
  - synthetic data
    - 1,000,000 nodes, 6 labels (A_1,A_2…,A_6)
  - real-world data
    - "unfolded" DBLP data set
    - For each paper, coauthor name is replaced with the actual information for that author
    - depth 805 and around 3 million nodes

Binary structural joins VS. the path join

- Execution time (synthetic data)

Query

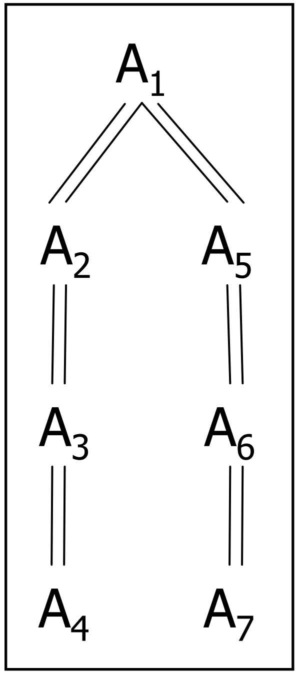
A_1
||
A_2
||
A_3
||
A_4
||
A_5
||
A_6

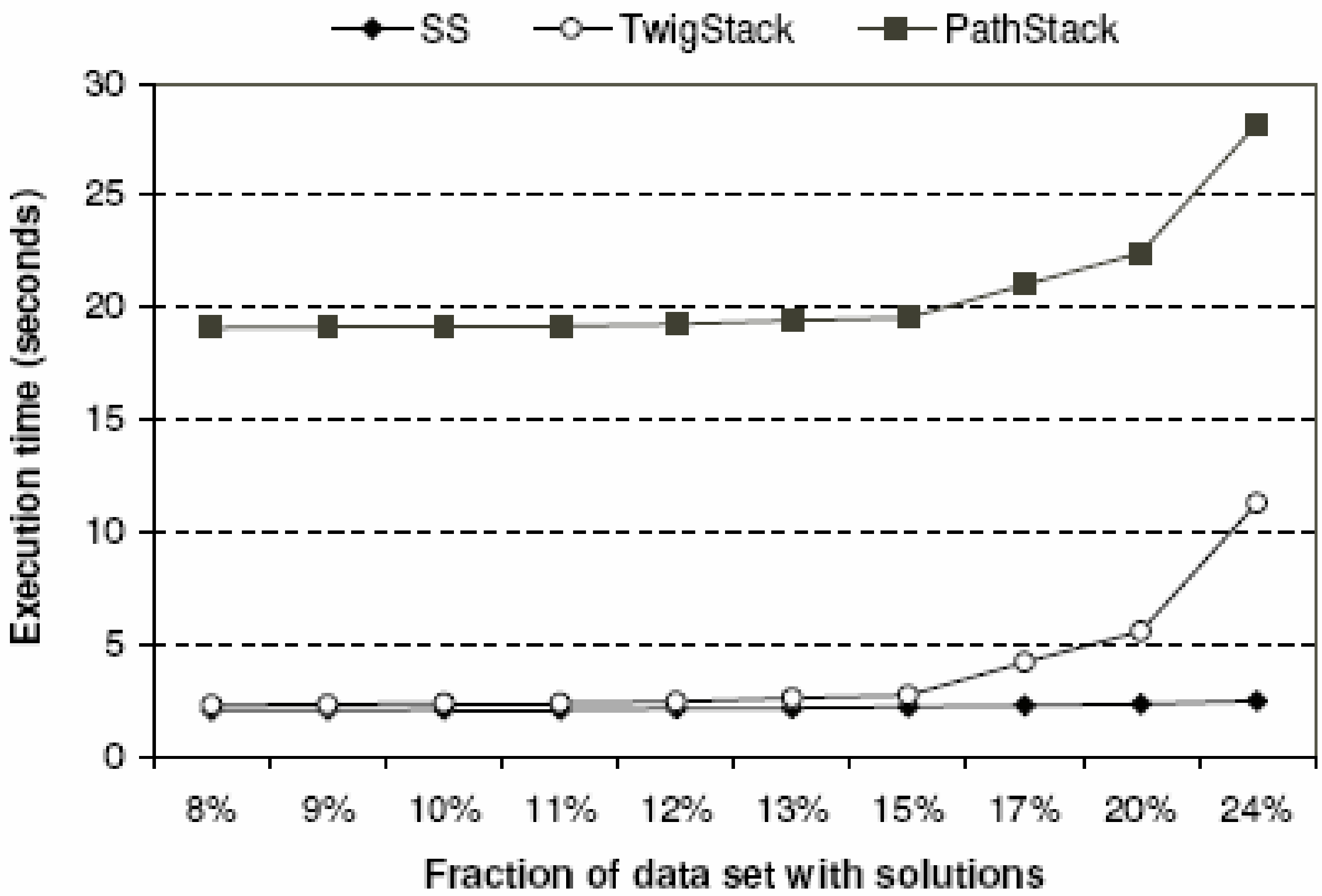


Twigs: the path join VS. the twig join

- Execution time (synthetic data)

Query

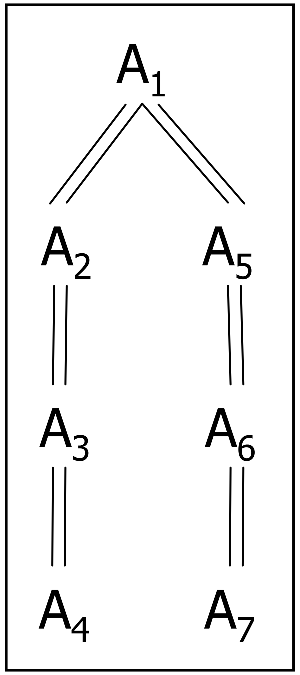


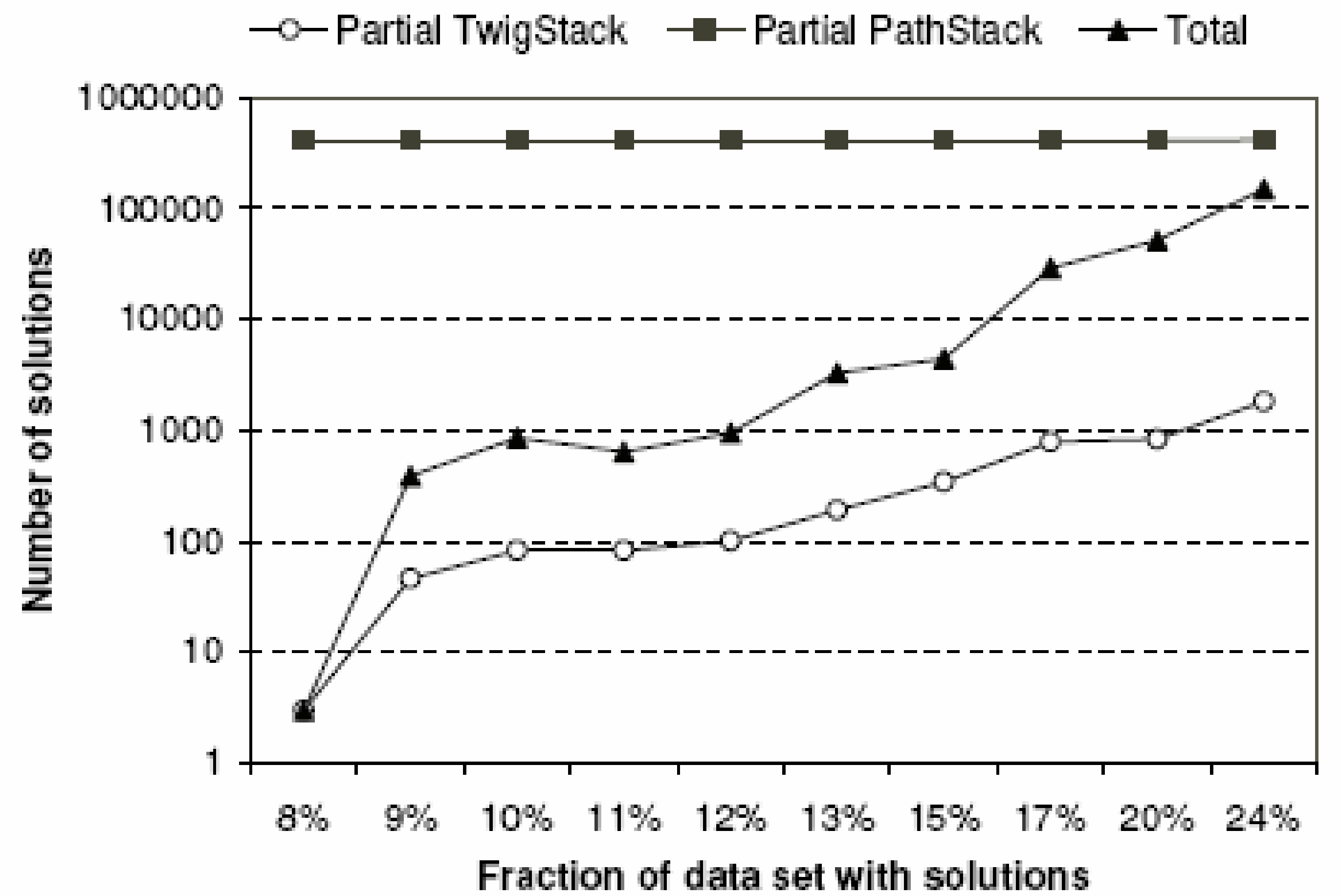


Twigs: the path join VS. the twig join (cont'd)

- Number of solutions (synthetic data)

Query





Conclusion

- Structural Joins (2-way join)
  - The tree-merge join algorithm
  - The stack-tree join algorithm
    - worst case complexity better than the tree-merge join
- Holistic Twig Joins (n-way join)
  - The path join algorithm
    - superior to any binary structural joins
  - The Twig join algorithm
    - optimal in the size of intermediate results