# XML Query Processing

CPS 216
Advanced Database Systems

## Announcements (March 31)



- Course project milestone 2 due today
  - Hardcopy in class or otherwise email please
- I will be out of town next week
  - No class on Tuesday (April 5); will make up during reading period
  - Badrish Chandramouli will give the lecture on Thursday (April 7)
- Homework #3 in less than two weeks (April 12)
- Reading assignment for next week will be assigned through email

## Overview



- Recall that XML queries based on path expressions can be expressed by joins
- Node/edge-based representation (graphs)
  - Equi-join on *id*'s
  - Chasing pointers ≈ index nested-loop joins
    - ☞"Navigational" approach
- Interval-based representation (trees)
  - "Containment" joins involving *left* and *right*
  - Sort-merge joins, zig-zag joins with indexes
    - ☞"Structural" approach

# Navigational processing in Lore

*VLDB* 1999

- Lore data model peculiarity: labels on edges instead of labels on nodes
- Access paths in Lore
  - Base representation: (parent, label) → child
  - Label index: (child, label) → parent
  - Edge index: label → (parent, child)
  - Value index: (value, label) → node
  - Path index: path expression → node
- Correspond to the following in a label-on-node model
  - label/value → node
  - (parent, label) → child
  - child → parent



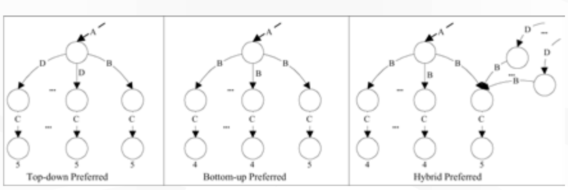
# Navigational plans in Lore

`//A/B/C[.=5]`

- Top down: pointer chasing
  - Start with `//A`, navigate down to `//A/B` and then to `//A/B/C`, and then check values of `C`
- Bottom up: reverse pointer chasing
  - Start with `//C[.=5]`, navigate up to `//B[/C[.=5]]` and then to `//A[/B/C[.=5]]`
- Hybrid: top down and bottom up, meet in middle
  - Start with `//A`, navigate down to `//A/B`
  - Start with `//C[.=5]`, navigate up to `//B[/C[.=5]]`
  - Intersect `B` nodes
  - ☞In general, hybrid can combine multiple top-down and bottom-up plans starting from anywhere in the path expression



# Comparison of Lore navigational plans

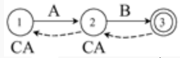



- Which plan is best depends on the size of the intermediate results it generates
  - Choose the optimal join order!
- Top down and bottom up are essentially index nested-loop joins (“pure” navigation)
- Hybrid can use any join strategy to combine subplans

# Niagara unnest

*VLDB* 2003

- Unnest: navigation-style processing using finite state machines

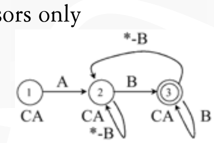



- Example: A/B
  - Given a list of elements for which A/B needs to be evaluated
  - Each state maintains a cursor
  - For each given element, state 1 uses a CA (child-axis) cursor with label A to iterate through all A children
  - For each A child, state 2 uses a CA cursor with label B to iterate through all B children of the A child
- Essentially a sequence of indexed nested-loop joins
  - Top down or bottom up, but not hybrid



# Alternative unnest strategies for //

- Example: A//B
- Using CA cursors only

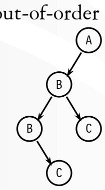



- Using DA (descendent-axis) cursor
  - Given node *n* and label A, a DA cursor iterates through all *n*//A nodes in document order




# Surprise with the DA cursor

- Recall that XPath expressions are supposed to return result nodes in document order
- Example: /A//B/C
  - DA enumerates descendents in document order
  - But subsequent steps may produce out-of-order results
- Duplicates are also an issue (e.g., query //A//B//C on data /A/B/B/C/C)



10

## Structural approach

- Binary containment joins (Al-Khalifa et al., *ICDE* 2002)
  - Given *Alist* and *Dlist*, two lists of elements encoded with (*left*, *right*), with each list sorted by *left*
  - Find all pairs of ($a$, $e$), where $a \in Alist$ and $e \in Dlist$, such that $a$ is a parent (or ancestor) of $e$
- Example query processing scenario: `//book/author`
  - Using an inverted-list index, retrieve the list of `book` elements sorted by *left*, and the list of `author` elements sorted by *left*
  - Find pairs that actually form parent-child relationships

11

## Tree-based algorithms

<u>Algorithm *Tree-Merge-Anc*</u>

*BeginJoinable* = 0;

For each $a$ in *Alist*:

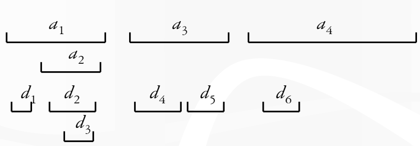
Start from *BeginJoinable* and skip *Dlist* until the first element with *left* > *a.left*; update *BeginJoinable*;

Start from *BeginJoinable* and join each $d$ from *Dlist* with $a$; stop at the first $d$ with *left* > *a.right*;

- An alternative algorithm, *Tree-Merge-Desc*, uses *Dlist* as the outer table instead of *Alist*, and requires minor tweaks to conditions

12

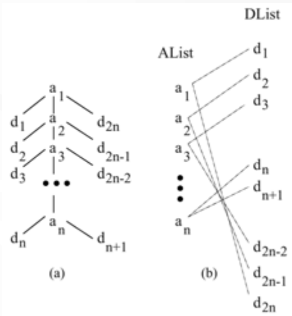
## *Tree-Merge-Anc* example




- $a_1$: *BeginJoinable* = $d_1$; stops at $d_4$
- $a_2$: *BeginJoinable* = $d_2$; stops at $d_4$
- $a_3$: *BeginJoinable* = $d_4$; stops at $d_6$
- $a_4$: *BeginJoinable* = $d_6$

☞ Further optimization is possible to avoid unnecessary rescanning; though in general rescanning cannot be avoided

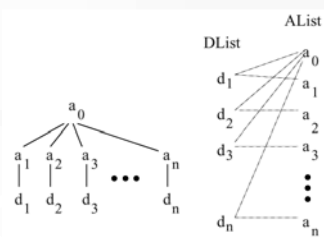
# Worst case of *Tree-Merge-Anc*




- Optimal (up to a constant factor) for //
- Not optimal for /



# Worst case of *Tree-Merge-Desc*




- Not even optimal for //
- Problem: linear access to *Alist* forces unnecessary scanning
- Idea: create another representation that corresponds more closely to a tree traversal



# Stack-based algorithms

<u>Algorithm *Stack-Tree-Desc*</u>

Start with an empty stack *Astack*

While *Astack* or *Alist* or *Dlist* is not empty:

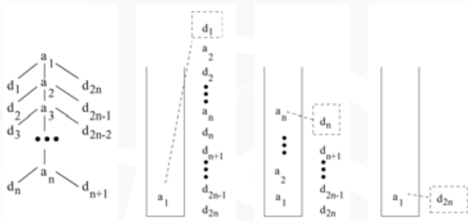
If heads of both *Alist* and *Dlist* come after the top of *Astack*, pop *Astack*;

Else if the head of *Alist* is contained by the top of *Astack*, push it onto *Astack* and advance *Alist*;

Else join the head of *Dlist* with everything on *Astack* and advance *Dlist*;

- Output is ordered by *Dlist*
- An alternative algorithm, *Stack-Tree-Anc*, orders output by *Alist* but requires more bookkeeping

# *Stack-Tree-Desc* example



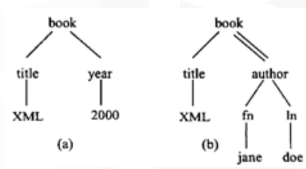
☞ Copying from *Alist* to *Astack* avoids the worst case of *Tree-Merge-Anc*



# Twigs

❖ "Twigs" represent longer and possibly branching XPath expressions
  - Problem: find all instances of a given twig in a document
    - More what XPath requires

```
//book[title="XML" and year="2000"]
//book[title="XML" and //author[fn="jane" and ln="doe"]]
```




Double edges represent //



# Holistic twig join

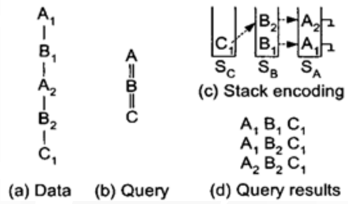
❖ Traditional approach: use a sequence of binary containment joins to process a twig

❖ Problem: intermediate results can get much larger than input and output sizes
  - Example?

❖ Idea: use a multi-way merge (since all joins are on the same attributes)
  - "Holistic" twig join (Bruno et al., *SIGMOD* 2002)

# Compact encoding using stacks

- One stack for each node in the query twig
  - Elements in a stack form a containment chain
- Each stack element points to one in the parent stack
  - Specifically, the top one that contains it





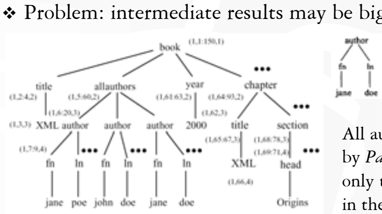

# *PathStack*

- Handles twigs with no branches $q1//q2//\ldots//qn$
- Input lists $T_{q1}, T_{q2}, \ldots, T_{qn}$ and stacks $S_{q1}, S_{q2}, \ldots, S_{qn}$
- While $T_{qn}$ is not empty:
  - Let $T_{qmin}$ be the list whose head has smallest *left*;
  - Clean all stacks: pop while top's *right* $<$ *head*($T_{qmin}$).*left*;
  - Push *head*($T_{qmin}$) on $S_{qmin}$, with pointer to *top*($S_{parent(q_{\min})}$);
  - If $q_{\min}$ is the leaf ($qn$), output results and pop $S_{qmin}$;
- Check properties
  - Elements in a stack form a containment chain
  - Each stack element points to the top one in the parent stack that contains it



# Extending *PathStack* to *TwigStack*

- A first cut
  - Decompose a twig into root-to-leaf paths
  - Process each path using *PathStack*
  - Merge solutions for all paths
- Problem: intermediate results may be big




All authors will be returned by *PathStack*, though only the last one should be in the final result

# *TwigStack*

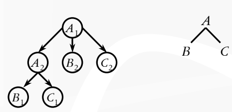
- Generate solutions for each root-to-leaf path
  - Do not use *PathStack*, which generates all solutions
  - Modify *PathStack* to generate only solutions that are parts of the final result (possible if twig contains only //)

    Specifically, when pushing $h_q$ onto stack $S_q$, ensure that
    - $h_q$ has a descendent $h_{q'}$ in the each input list $T_{q'}$ where $q'$ is a child of $q$
    - Each $h_{q'}$ recursively satisfies the above property
- Merge solutions for all paths



# *TwigStack* still suboptimal for /

- Example

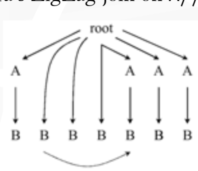



- Desired result: $(A_1, B_2, C_2)$, $(A_2, B_1, C_1)$
- Initial state: all three stacks empty; ready to push one of $A_1$, $B_1$, $C_1$ onto a stack
- If we want to ensure that non-contributing nodes are never pushed onto the stack, then
  - Cannot decide on $A_1$ unless we see $B_2$ and $C_2$
  - Cannot decide on $B_1$ or $C_1$ unless we see $A_2$



# Optimization using an index

- Idea: if there are indexes on input lists ordered by *left*, use these indexes to skip lists more efficiently
- Example: Niagara's ZigZag join on A//B




  - After advancing to the second A, use the index on B list to go directly to the first joining B, instead of scanning B list linearly
  - When processing a B, use the index on A list to skip

## Summary of structural approach

25

- What makes XML containment joins easier than joining lists of arbitrary intervals?
  - Intervals form either disjoint or containment relationships, but they cannot overlap
  - This property is heavily exploited by stack-based algorithms
- Most algorithms in literature assume that bindings must be produced for all nodes in a twig
  - Unnecessary requirement in practice
  - Leads to potentially much larger result sizes
  - Is it possible to have more efficient algorithms that produce bindings for only selected nodes in a twig?

## Navigational vs. structural approaches

26

- In the past some has argued that structural is preferable to navigational
- Niagara argues for a mixed-mode approach, using a cost-based analysis to pick which approach or combination of approaches is better
  - Just like one would implement both index nested-loop join and sort-merge join